**Program 1: Multidimensional array in c**

//multidimensional array in c

// C Program to store and print 12 values entered by the user

#include <stdio.h>

void main()

{

int integer[2][3][2];

printf("Enter 12 values: \n");

for (int i = 0; i < 2; ++i)

{

for (int j = 0; j < 3; ++j)

{

for (int k = 0; k < 2; ++k)

{

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

}

}

}

printf("\nDisplaying values:\n");

for (int i = 0; i < 2; ++i)

{

for (int j = 0; j < 3; ++j)

{

for (int k = 0; k < 2; ++k)

{

printf("Integer[%d][%d][%d] = %d\n", i, j, k, integer[i][j][k]);

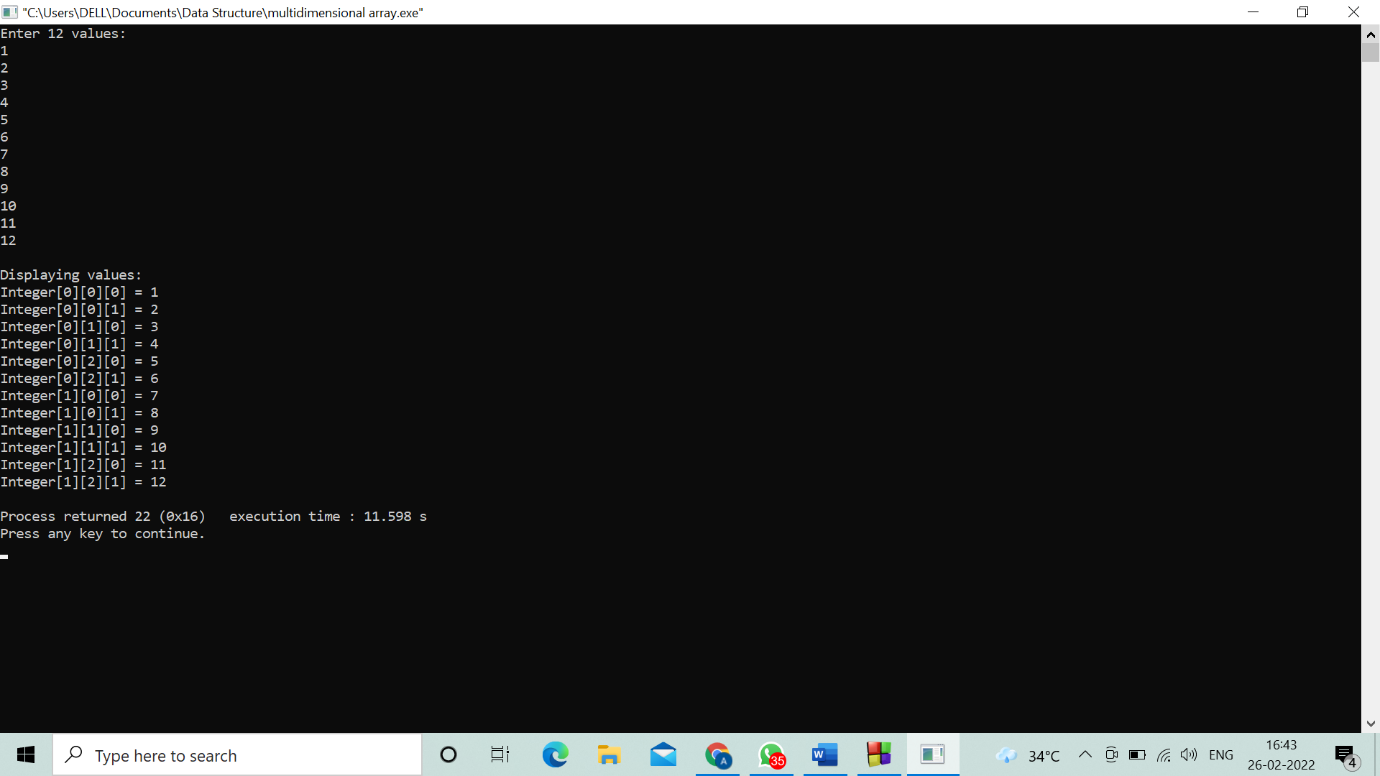
}

}

}

}

Output:



**Program 2. Passing array in function in c argument**

// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>

float calculateSum(float num[]);

void main() {

float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};

result = calculateSum(num);

printf("Result = %.2f", result);

}

float calculateSum(float num[]) {

float sum = 0.0;

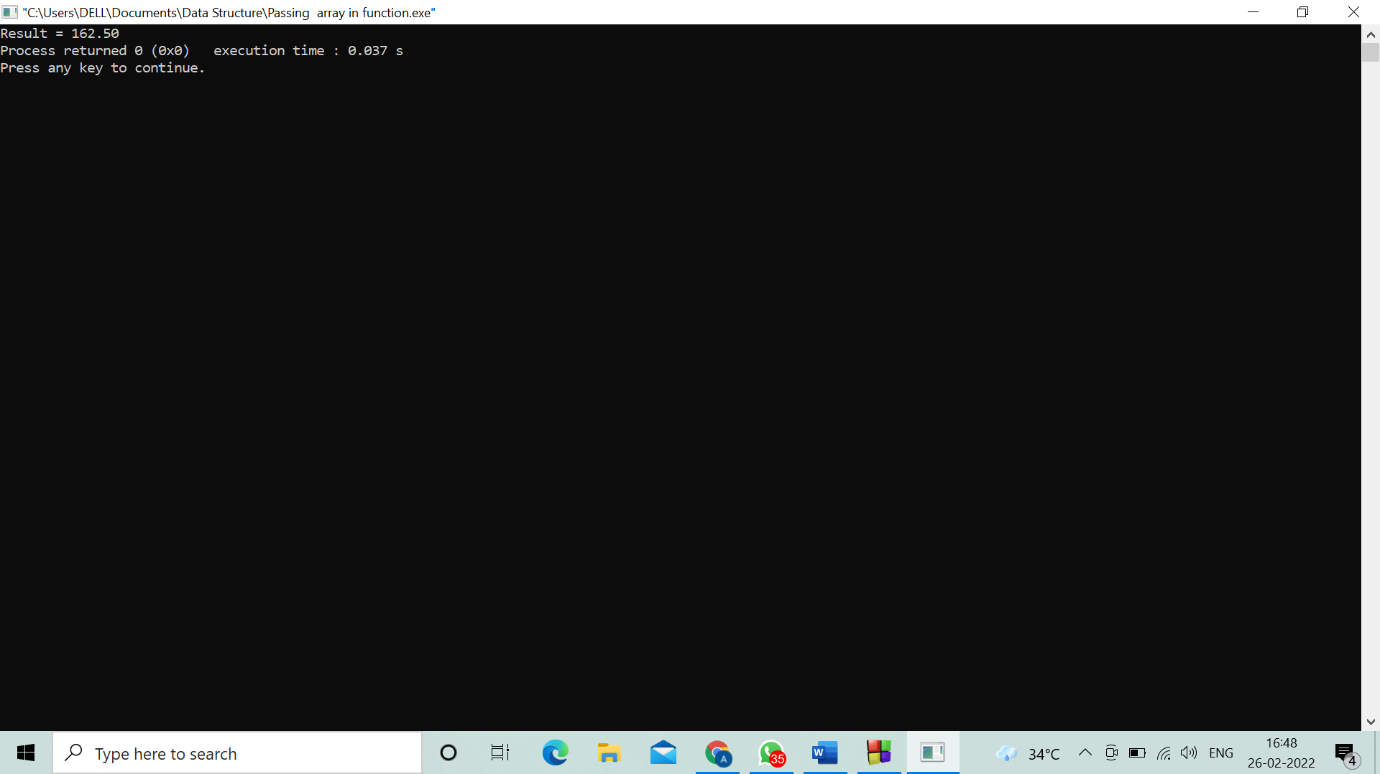
for (int i = 0; i < 6; ++i) {

sum += num[i];

}

return sum;

}

Output:

**Program 3. Return array in function from c**

#include <stdio.h>

#define MAX\_SIZE 10

void getArray(int arr[], int size);

int main()

{

int arr[MAX\_SIZE];

int i;

getArray(arr, MAX\_SIZE);

printf("Array outside function:");

for (i = 0; i < MAX\_SIZE; i++){

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

}

return 0;

}

void getArray(int arr[], int size)

{

int i;

printf("Enter elements in array: ");

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

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

}

printf("Array inside function:");

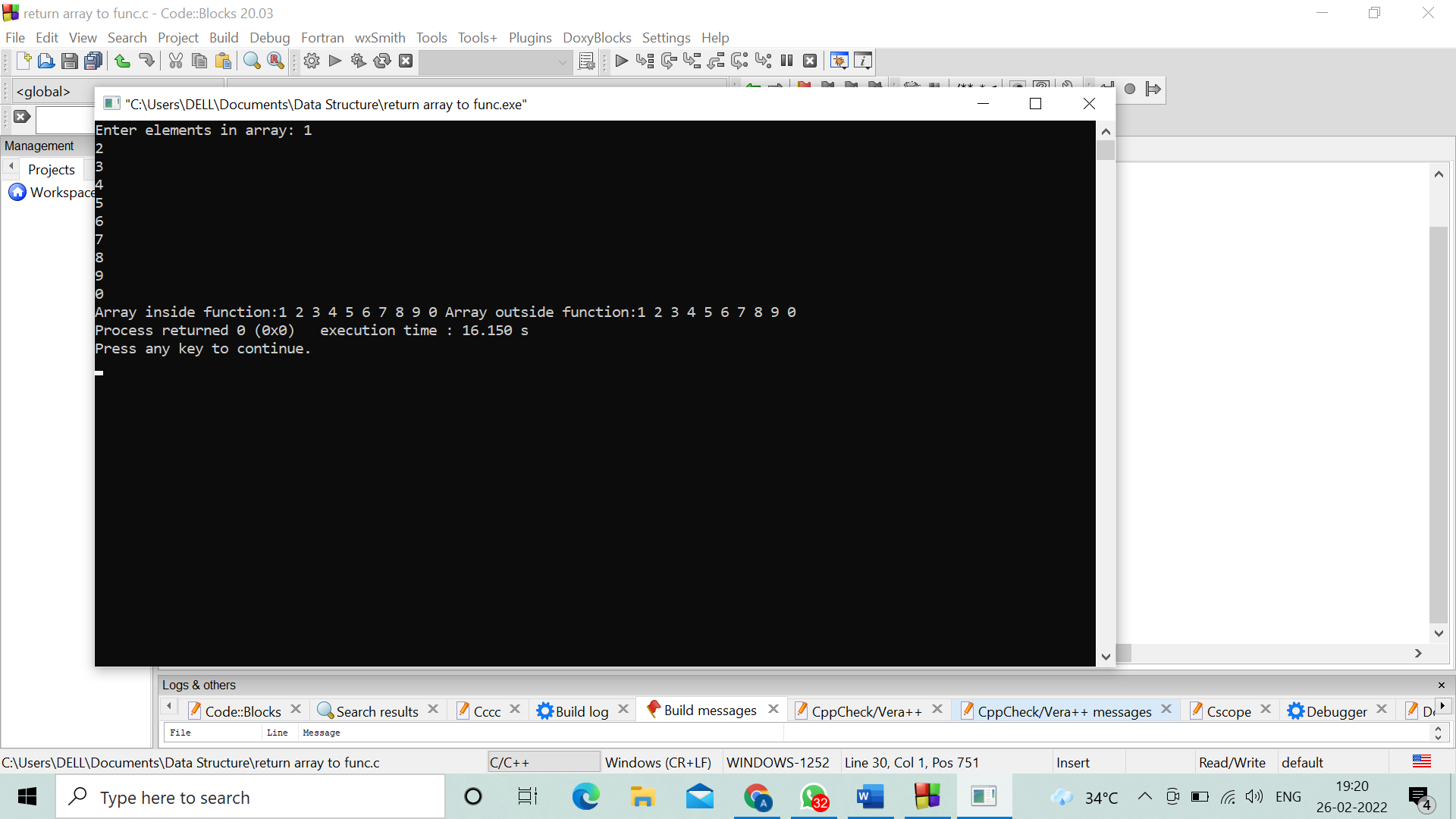
for (i = 0; i < size; i++){

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

}

}

Output:



**Program 4.Pointer to array in c.**

#include <stdio.h>

void main()

{

int i, x[6], sum = 0;

printf("Enter 6 numbers: ");

for(i = 0; i < 6; ++i) {

scanf("%d", x+i);

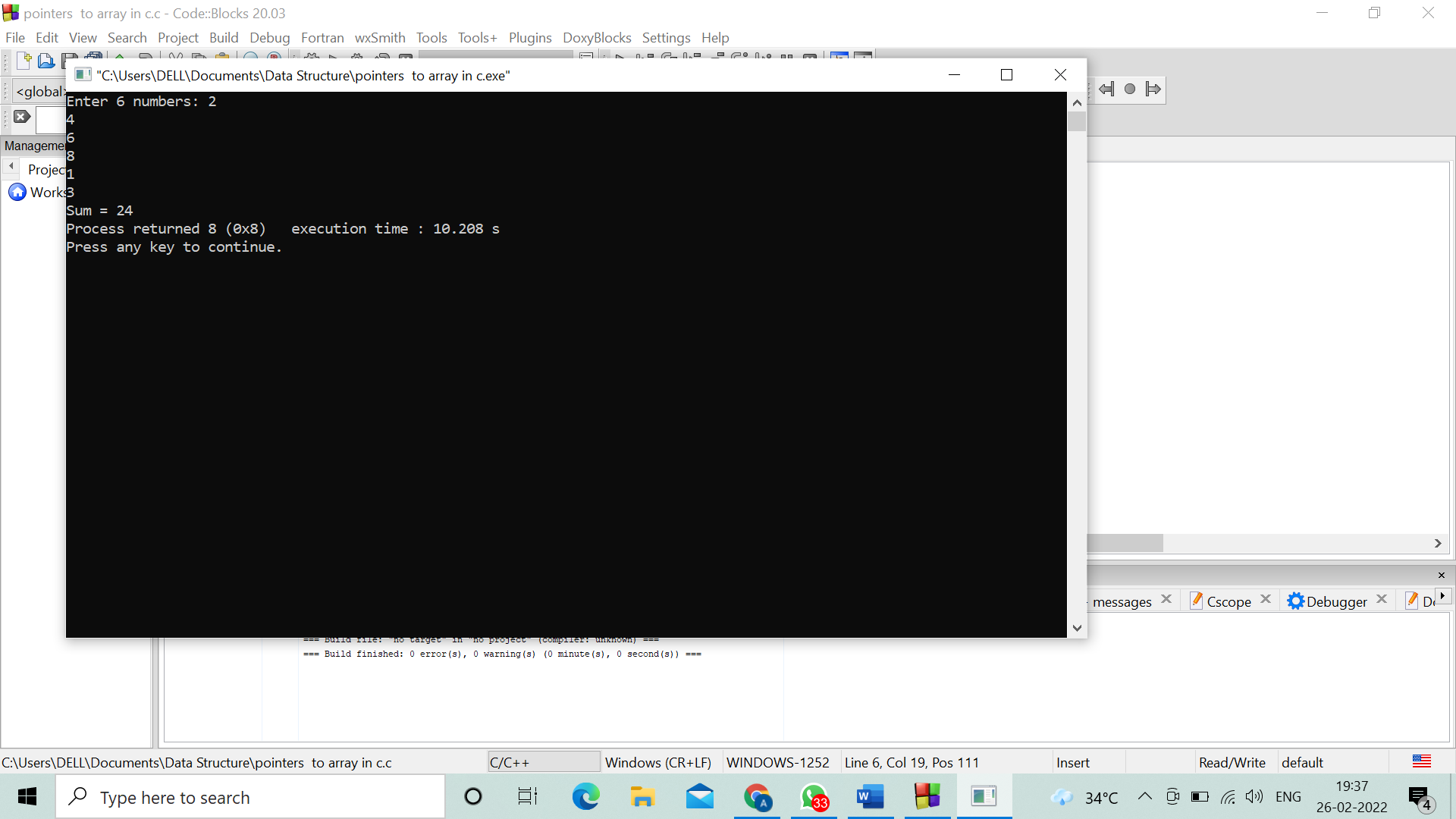
sum += \*(x+i);

}

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

}

Output:



**Program 5. Linear search in unsorted array**

//C program to implement linear search in unsorted array

#include <stdio.h>

#include <conio.h>

void main(){

int inputArray[100], elementCount, counter, num;

printf("Enter Number of Elements in Array\n");

scanf("%d", &elementCount);

printf("Enter %d numbers \n", elementCount);

for(counter = 0; counter < elementCount; counter++){

scanf("%d", &inputArray[counter]);

}

printf("Enter a number to serach in Array\n");

scanf("%d", &num);

for(counter = 0; counter < elementCount; counter++){

if(inputArray[counter] == num){

printf("Number %d found at index %d\n", num, counter);

break;

}

}

if(counter == elementCount){

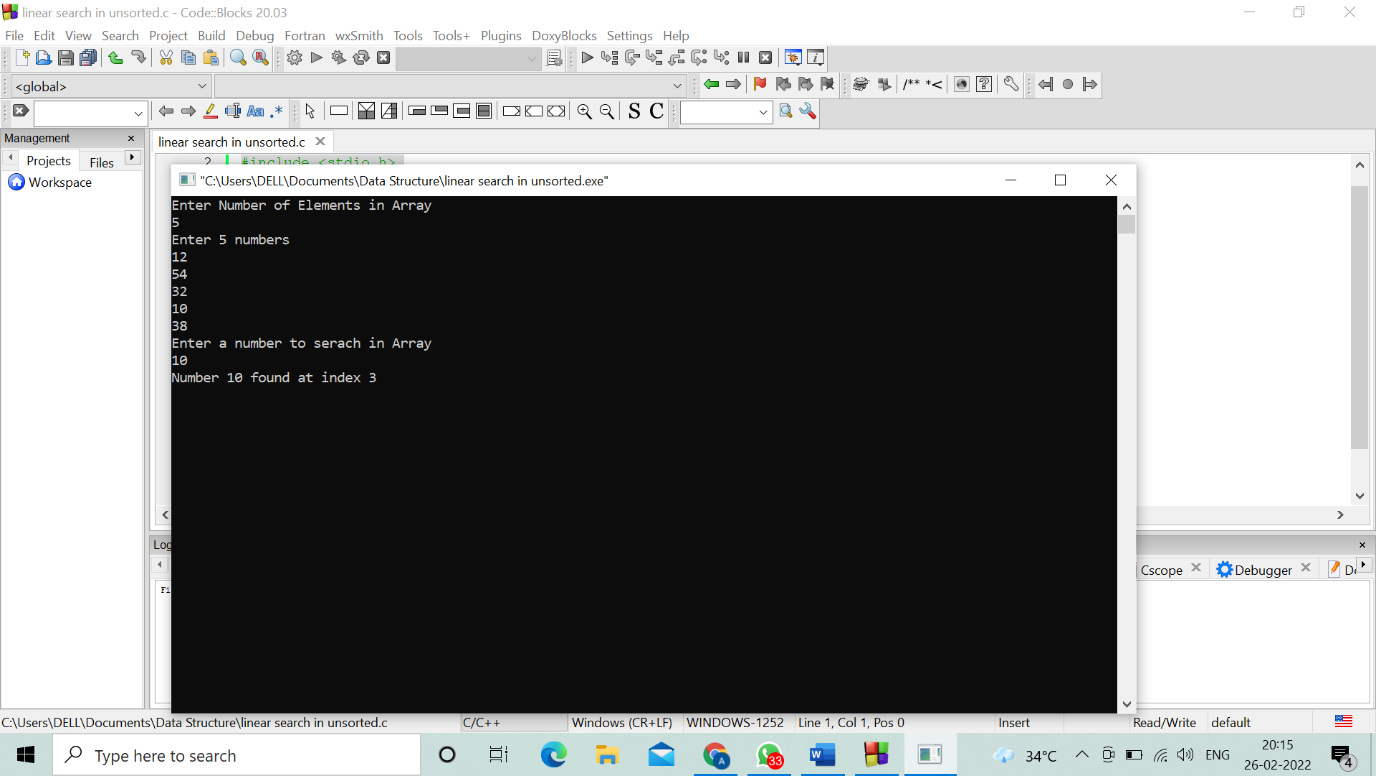
printf("Number %d Not Present in Input Array\n", num);

}

getch();

}

Output:



**Program 6. C program to insert an element in an array**

#include <stdio.h>

int main()

{

int array[100], position, c, n, value;

printf("Enter number of elements in array\n");

scanf("%d", &n);

printf("Enter %d elements\n", n);

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

scanf("%d", &array[c]);

printf("Enter the location where you wish to insert an element\n");

scanf("%d", &position);

printf("Enter the value to insert\n");

scanf("%d", &value);

for (c = n - 1; c >= position - 1; c--)

array[c+1] = array[c];

array[position-1] = value;

printf("Resultant array is\n");

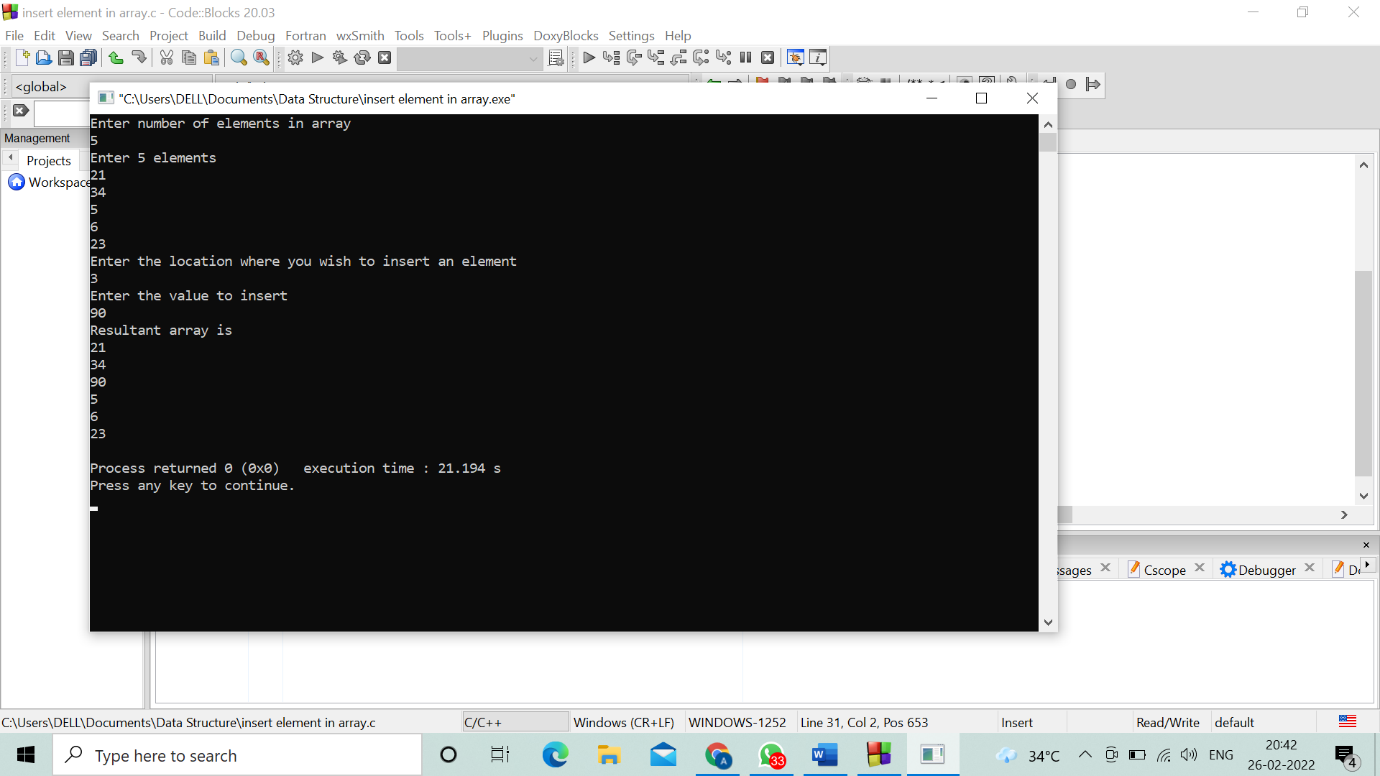
for (c = 0; c <= n; c++)

printf("%d\n", array[c]);

return 0;

}

Output:



**Program 7. C program to delete an element in an array**

#include <stdio.h>

void main()

{

int array[100], position, c, n;

printf("Enter number of elements in array\n");

scanf("%d", &n);

printf("Enter %d elements\n", n);

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

scanf("%d", &array[c]);

printf("Enter the location where you wish to delete element\n");

scanf("%d", &position);

if (position >= n+1)

printf("Deletion not possible.\n");

else

{

for (c = position - 1; c < n - 1; c++)

array[c] = array[c+1];

printf("Resultant array:\n");

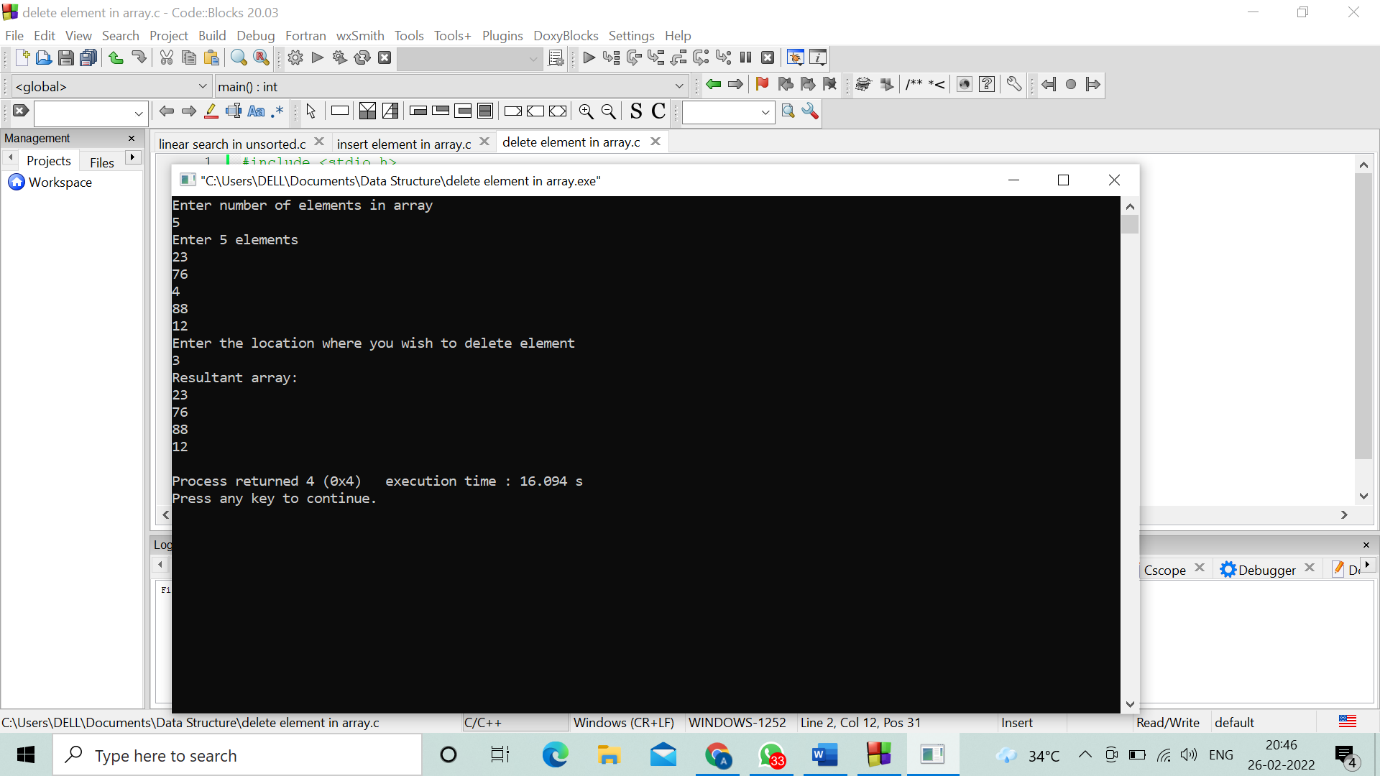
for (c = 0; c < n - 1; c++)

printf("%d\n", array[c]);

}

}

Output:



**Program 8. c program using array**

#include <stdio.h>

void main()

{

int marks[10], i, n, sum = 0, average;

printf("Enter number of elements: ");

scanf("%d", &n);

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

printf("Enter number%d: ",i+1);

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

sum += marks[i];

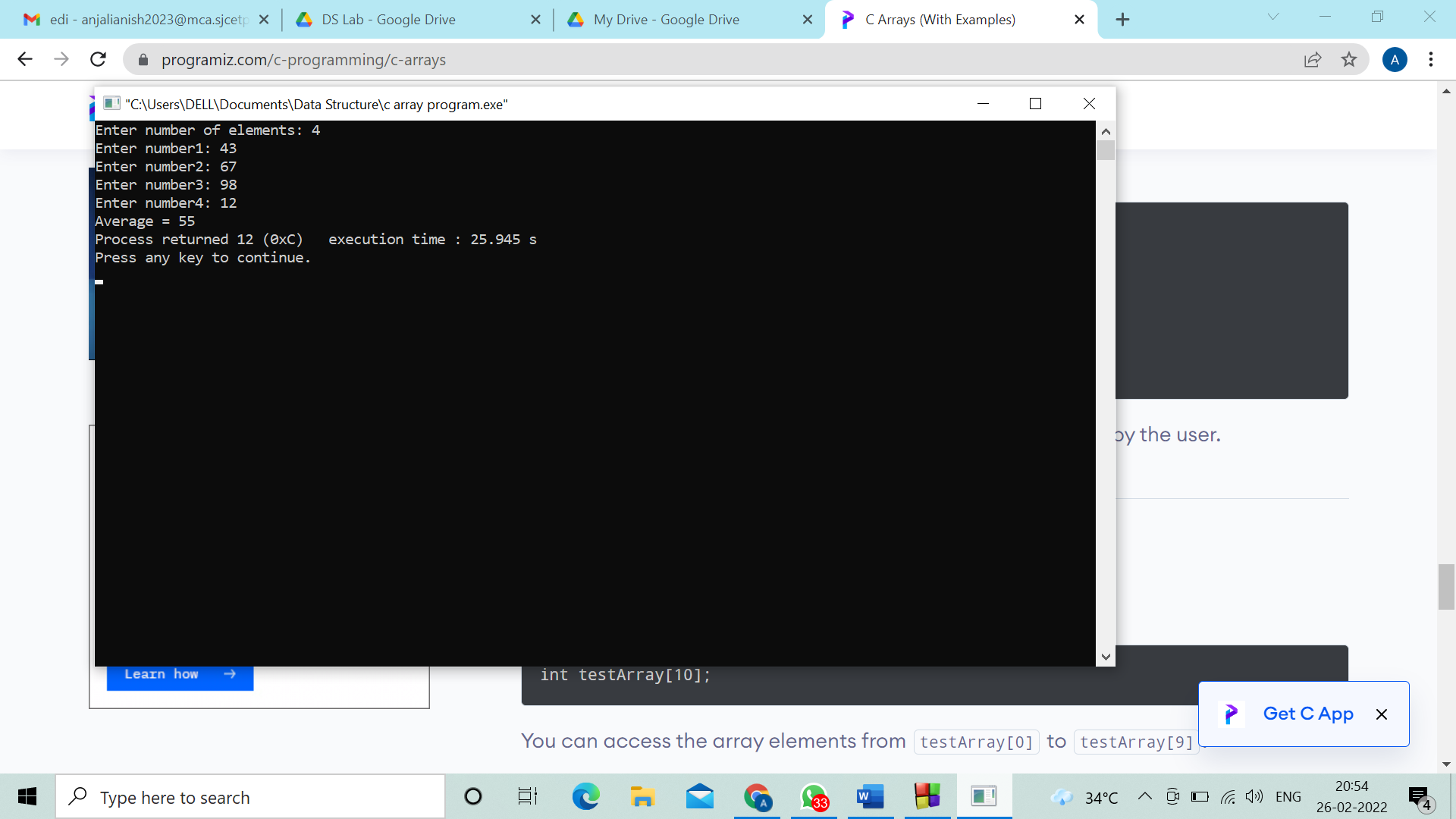
}

average = sum / n;

printf("Average = %d", average);

}

Output:



**Program 9. Write a program using structure**

#include<stdio.h>

struct Student

{

char name[10];

int roll;

};

void show(struct Student st);

void main(){

struct Student std;

printf("\nEnter Student record:\n");

printf("\nStudent name:\t");

scanf("%s", std.name);

printf("\nEnter Student rollno.:\t");

scanf("%d", &std.roll);

show(std);

}

void show(struct Student st)

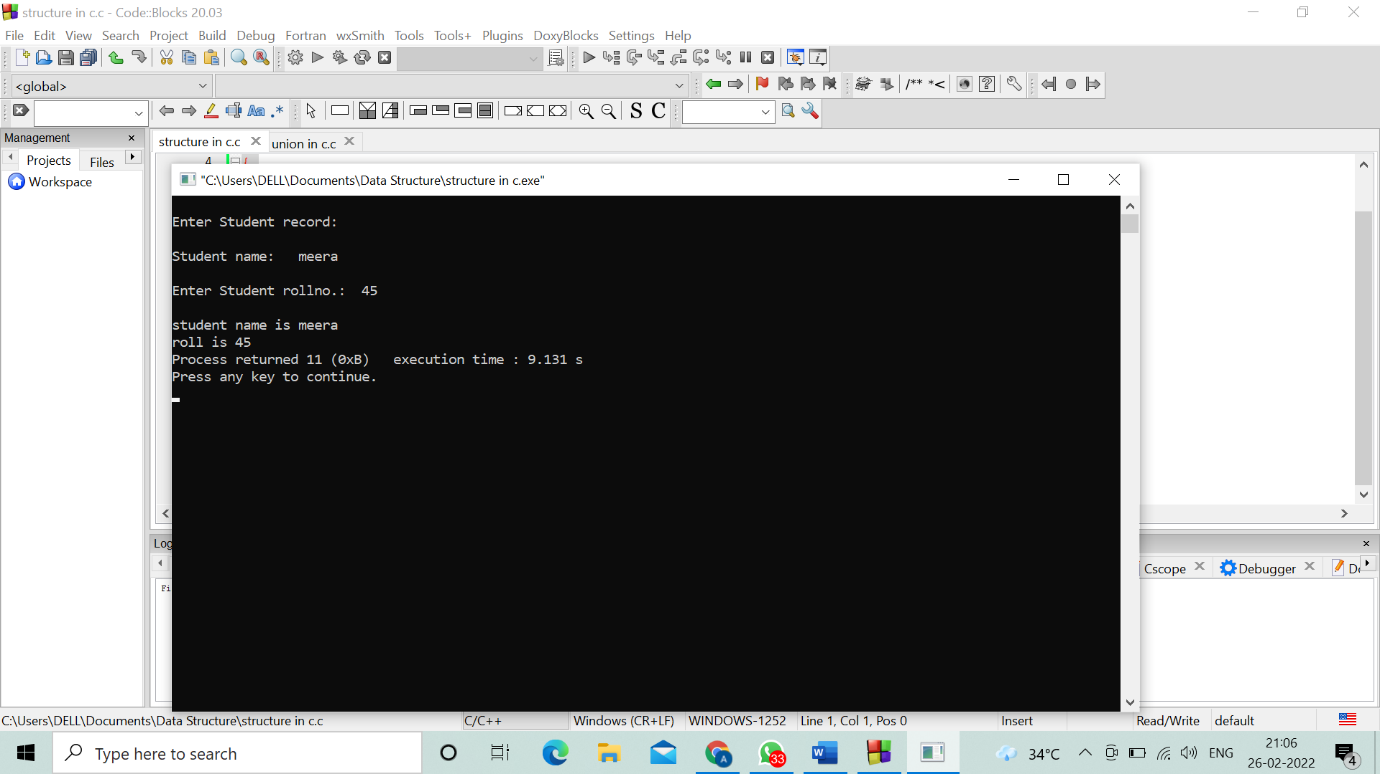
{

printf("\nstudent name is %s", st.name);

printf("\nroll is %d", st.roll);

}

Output:



**Program 10. write a program using union**

#include<stdio.h>

struct student

{

char name[20];

int rollno;

float marks;

};

struct student s1[3];

void main()

{

int i;

printf("Enter Name, RollNo, and Marks of Three Students:\n");

for(i=0; i<=2; i++)

scanf("%s %d %f",&s1[i].name,&s1[i].rollno,&s1[i].marks);

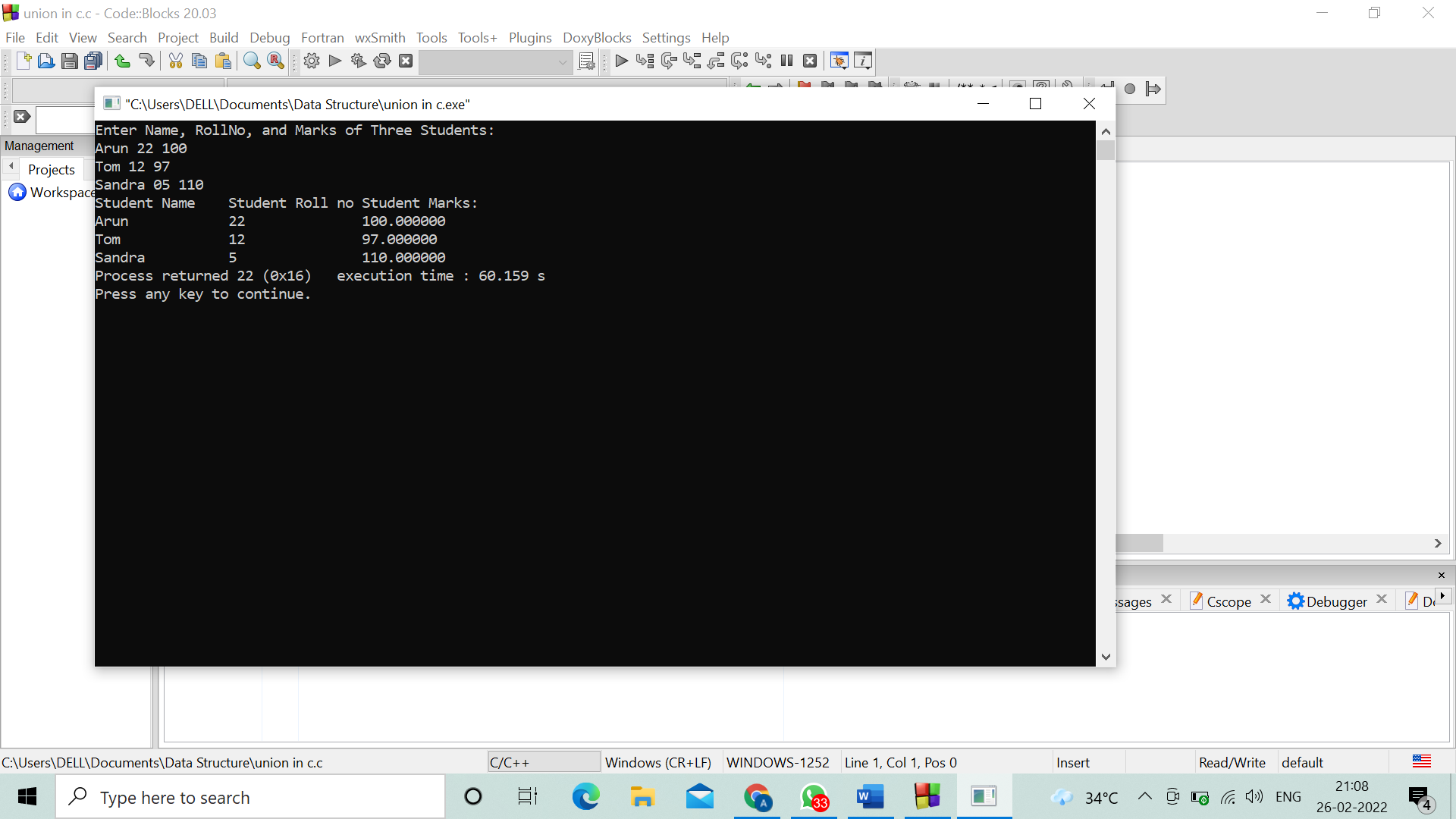
printf("Student Name\tStudent Roll no\tStudent Marks:");

for(i=0; i<=2; i++)

printf("\n%s\t\t%d\t\t%f", s1[i].name, s1[i].rollno, s1[i].marks);

}

Output:



**Program 11. program to implement stack operation**

// stack implementation:

#include<stdio.h>

#include<stdlib.h>

#define Size 5

int Top=-1, array[Size];

void Push();

void Pop();

void show();

int main()

{

int choice;

while(1)

{

printf("\nOperations performed by Stack");

printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");

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

scanf("%d",&choice);

switch(choice)

{

case 1: Push();

break;

case 2: Pop();

break;

case 3: show();

break;

case 4: exit(0);

default: printf("\nInvalid choice!!");

}

}

}

void Push()

{

int x;

if(Top==Size-1) {

printf("\n STACK FULL !!");

}

else{

printf("\nEnter element to be inserted to the stack:");

scanf("%d",&x);

Top=Top+1;

array[Top]=x;

}

}

void Pop()

{

if(Top==-1) {

printf("\n STACK NOT FULL!!");

}

else {

printf("\n Popped element: %d",array[Top]);

Top=Top-1;

}

}

void show()

{

if(Top==-1){

printf("\nSTACK NOT FULL!!");

}

else{

printf("\nElements present in the stack: \n");

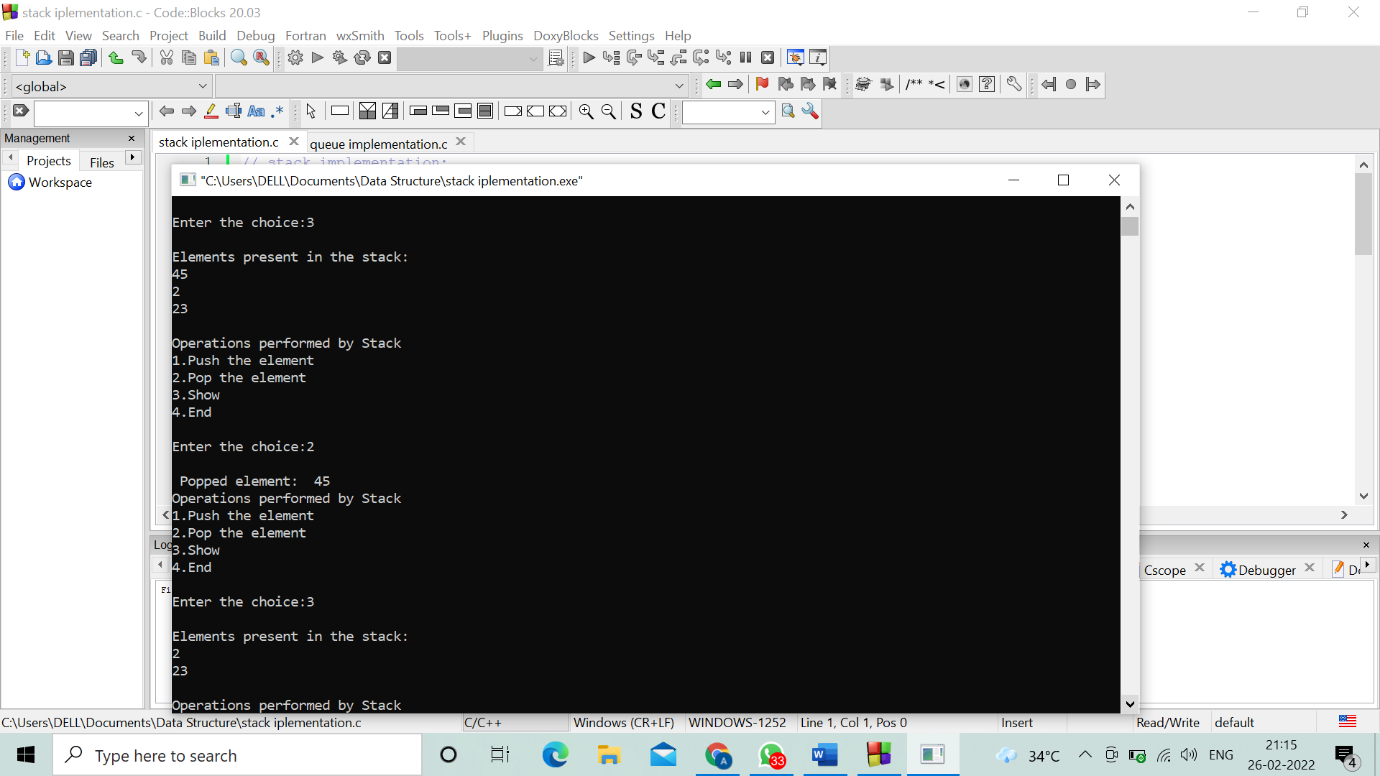
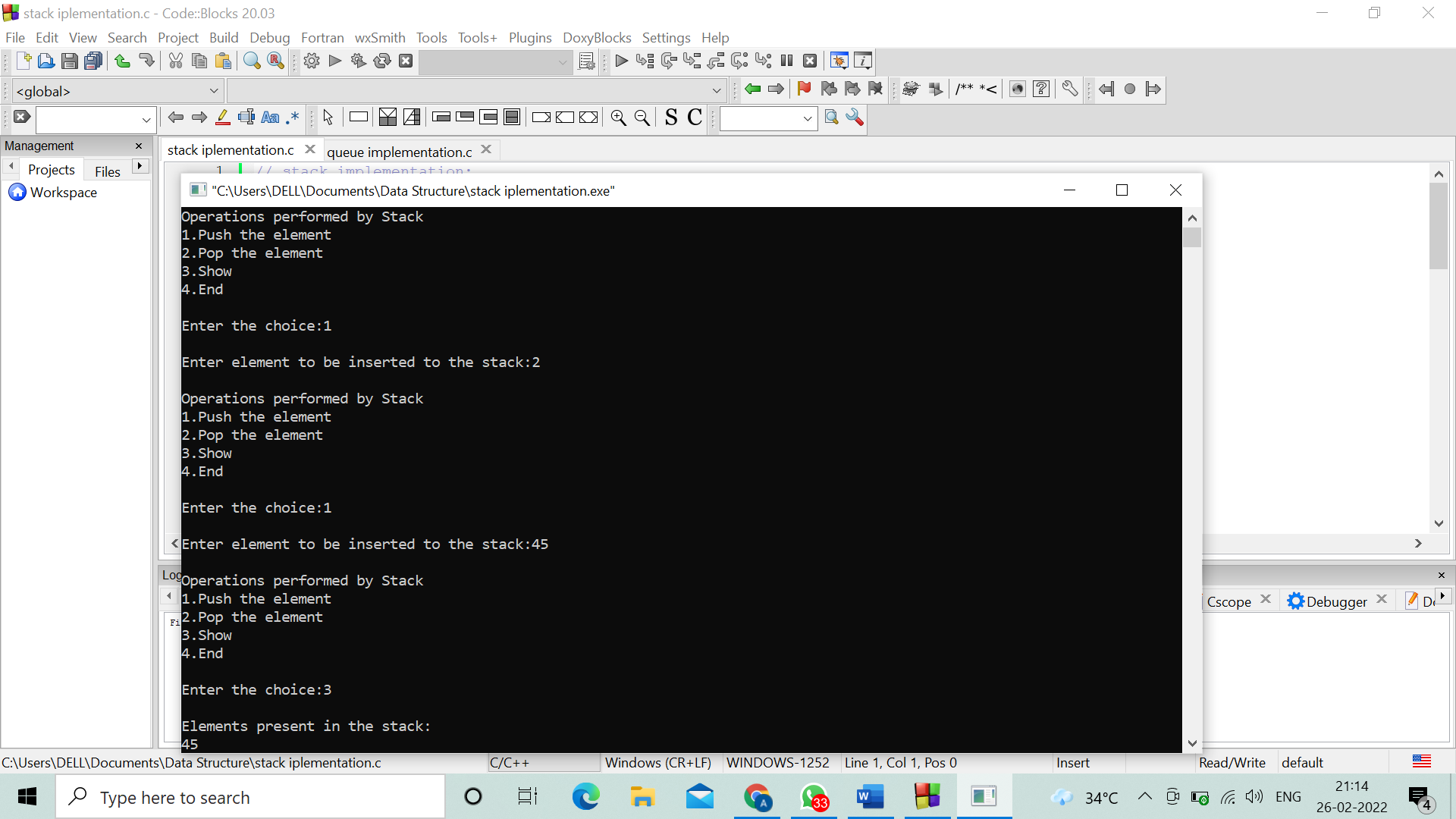
for(int i=Top;i>=0;--i)

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

}

}

Output:



**Program 12. Write a program for Queue implementation**

#include <stdio.h>

#define MAX 10

void insert();

void delete();

void display();

int queue\_array[MAX];

int rear = - 1;

int front = - 1;

main()

{

int choice;

while (1)

{

printf("1.Insert element to queue \n");

printf("2.Delete element from queue \n");

printf("3.Display all elements of queue \n");

printf("4.Quit \n");

printf("Enter your choice : ");

scanf("%d", &choice);

switch (choice)

{

case 1:

insert();

break;

case 2:

delete();

break;

case 3:

display();

break;

case 4:

exit(1);

default:

printf("Wrong choice \n");

}

}

}

void insert()

{

int add\_item;

if (rear == MAX - 1)

printf("Queue Overflow \n");

else

{

if (front == - 1)

front = 0;

printf("Inset the element in queue : ");

scanf("%d", &add\_item);

rear = rear + 1;

queue\_array[rear] = add\_item;

}

}

void delete()

{

if (front == - 1 || front > rear)

{

printf("Queue Underflow \n");

return ;

}

else

{

printf("Element deleted from queue is : %d\n", queue\_array[front]);

front = front + 1;

}

}

void display()

{

int i;

if (front == - 1)

printf("Queue is empty \n");

else

{

printf("Queue is : \n");

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

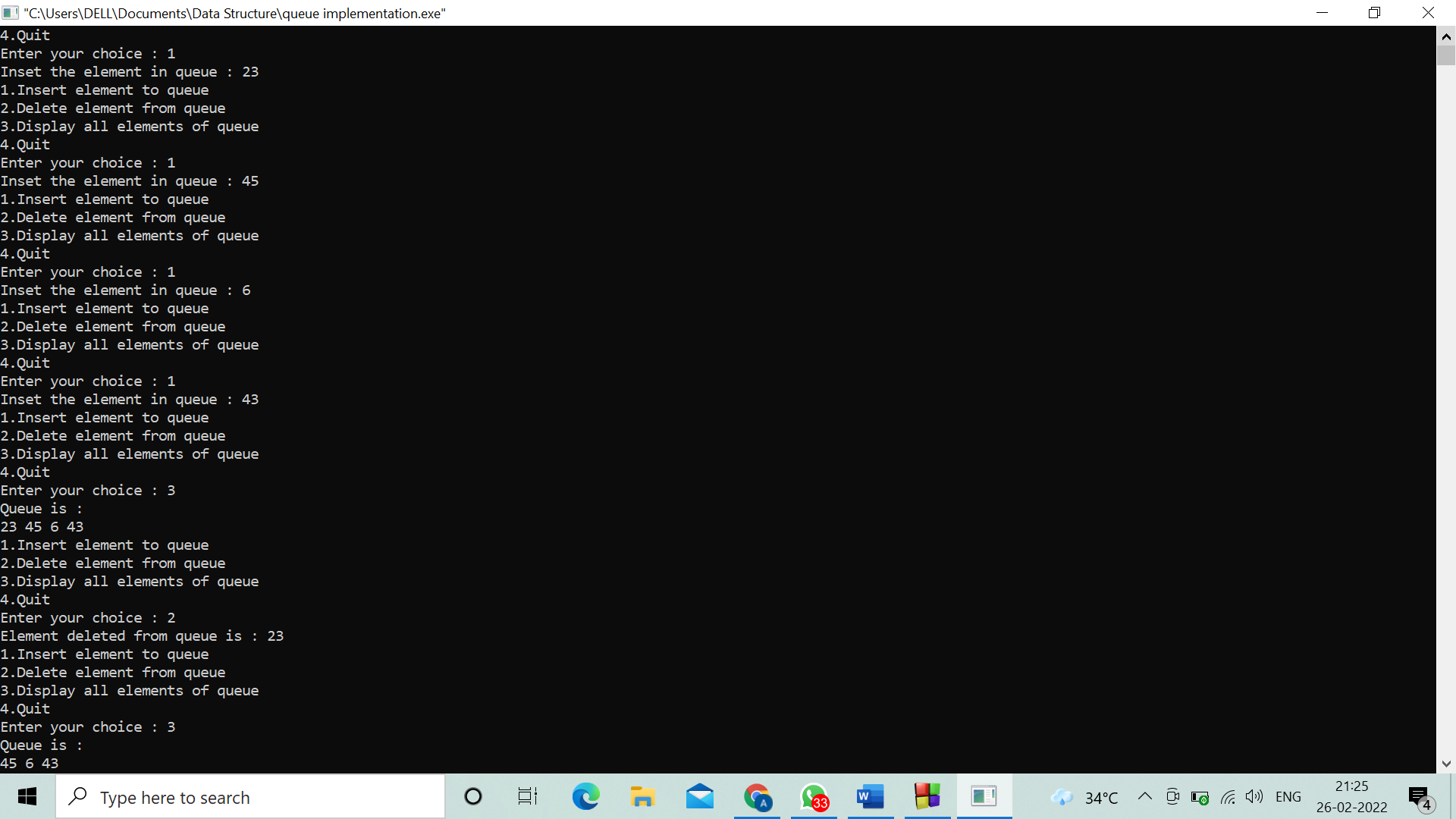
printf("%d ", queue\_array[i]);

printf("\n");

}

}

Output:



**Program 13. Write a program to implement circular queue**

//Circular Queue using Array

#include<conio.h>

#include<stdio.h>

#include<stdlib.h>

int cqueue[6];

int front = -1, rear = -1, n=6;

void enqueue(int val)

{

if ((front == 0 && rear == n-1) || (front == rear+1))

{

printf("Queue Overflow \n");

return;

}

if (front == -1)

{

front = 0;

rear = 0;

}

else

{

if (rear == n - 1)

rear = 0;

else

rear = rear + 1;

}

cqueue[rear] = val ;

}

void dequeue()

{

if (front == -1)

{

printf("Queue Underflow\n");

return ;

}

printf("Element deleted from queue is : %d ", cqueue[front]);

if (front == rear)

{

front = -1;

rear = -1;

}

else

{

if (front == n - 1)

front = 0;

else

front = front + 1;

}

}

void display()

{

int f = front, r = rear;

if (front == -1)

{

printf("Queue is empty");

return;

}

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

if (f <= r)

{

while (f <= r)

{

printf("%d ", cqueue[f]);

f++;

}

}

else

{

while (f <= n - 1)

{

printf("%d ",cqueue[f]);

f++;

}

f = 0;

while (f <= r) {

printf("%d ",cqueue[f]);

f++;

}

}

}

int menu()

{

int choice;

printf("\n 1.Add value to the list");

printf("\n 2. Delete value to the list");

printf("\n 3. View List");

printf("\n 4. Exit\n");

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

scanf("%d",&choice);

return(choice);

}

void main()

{

int value;

while(1)

{

switch(menu())

{

case 1:

printf("Input for insertion: ");

scanf("%d",&value);

enqueue(value);

break;

case 2:

dequeue();

break;

case 3:

display();

break;

case 4:

exit(0);

default:

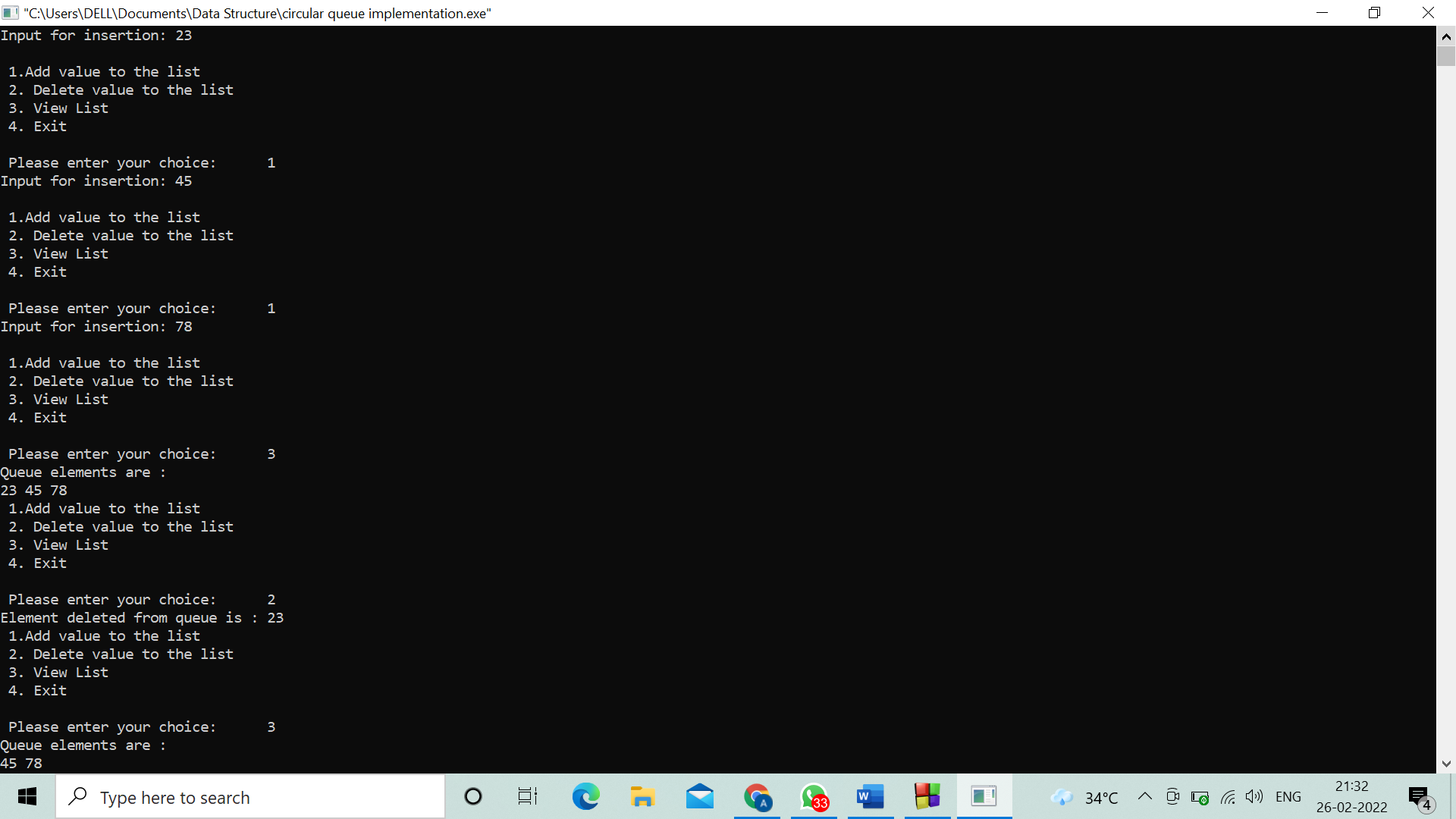
printf("invalid choice");

}

}

}

Output:



**Program 14. write a program to implement singly linked list**

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

struct node {

int value;

struct node \*next;

};

void insert();

void display();

void delete();

int count();

typedef struct node DATA\_NODE;

DATA\_NODE \*head\_node, \*first\_node, \*temp\_node = 0, \*prev\_node, next\_node;

int data;

int main()

{

int option = 0;

printf("Singly Linked List :\n");

while (option < 5)

{

printf("\nOptions\n");

printf("1 : Insert into Linked List \n");

printf("2 : Delete from Linked List \n");

printf("3 : Display Linked List\n");

printf("4 : Count Linked List\n");

printf("5: Exit\n");

printf("Enter your option:");

scanf("%d", &option);

switch (option) {

case 1:insert();

break;

case 2:delete();

break;

case 3:display();

break;

case 4:count();

break;

case 5:exit(1);

break;

default:printf("Incorrect Choice.\n");

break;

}

}

return 0;

}

void insert() {

printf("\n Enter Element for Inserting in Linked List : \n");

scanf("%d", &data);

temp\_node = (DATA\_NODE \*) malloc(sizeof (DATA\_NODE));

temp\_node->value = data;

if (first\_node == 0) {

first\_node = temp\_node;

} else {

head\_node->next = temp\_node;

}

temp\_node->next = 0;

head\_node = temp\_node;

}

void delete() {

int countvalue, pos, i = 0;

countvalue = count();

temp\_node = first\_node;

printf("\n Enter Position for Delete Element : \n");

scanf("%d", &pos);

if (pos > 0 && pos <= countvalue) {

if (pos == 1) {

temp\_node = temp\_node -> next;

first\_node = temp\_node;

printf("\n Deleted Successfully \n\n");

} else {

while (temp\_node != 0) {

if (i == (pos - 1)) {

prev\_node->next = temp\_node->next;

if(i == (countvalue - 1))

{

head\_node = prev\_node;

}

printf("\n Deleted Successfully \n\n");

break;

} else {

i++;

prev\_node = temp\_node;

temp\_node = temp\_node -> next;

}

}

}

} else

printf("\n Invalid Position \n\n");

}

void display() {

int count = 0;

temp\_node = first\_node;

printf("\n Display Linked List : \n");

while (temp\_node != 0) {

printf(" %d ", temp\_node->value);

count++;

temp\_node = temp\_node -> next;

}

}

int count() {

int count = 0;

temp\_node = first\_node;

while (temp\_node != 0) {

count++;

temp\_node = temp\_node -> next;

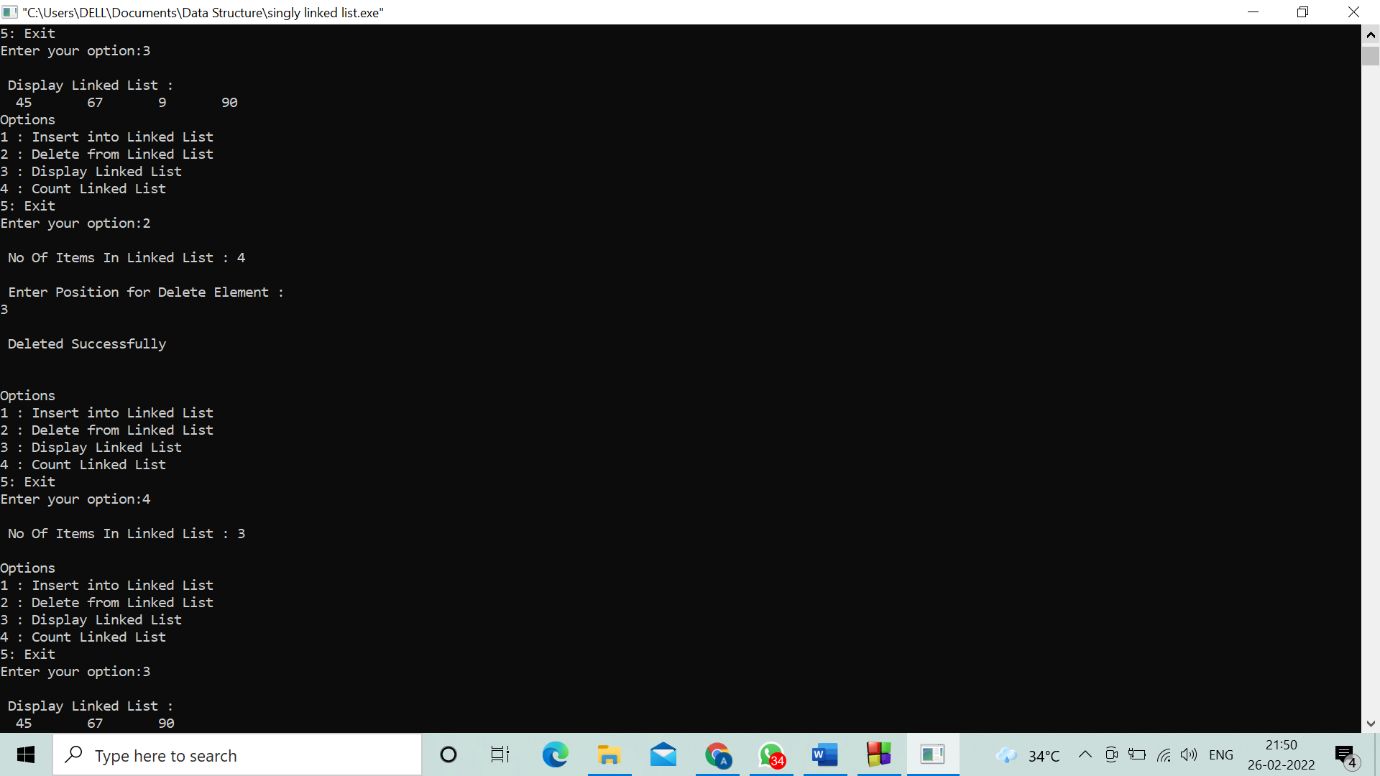
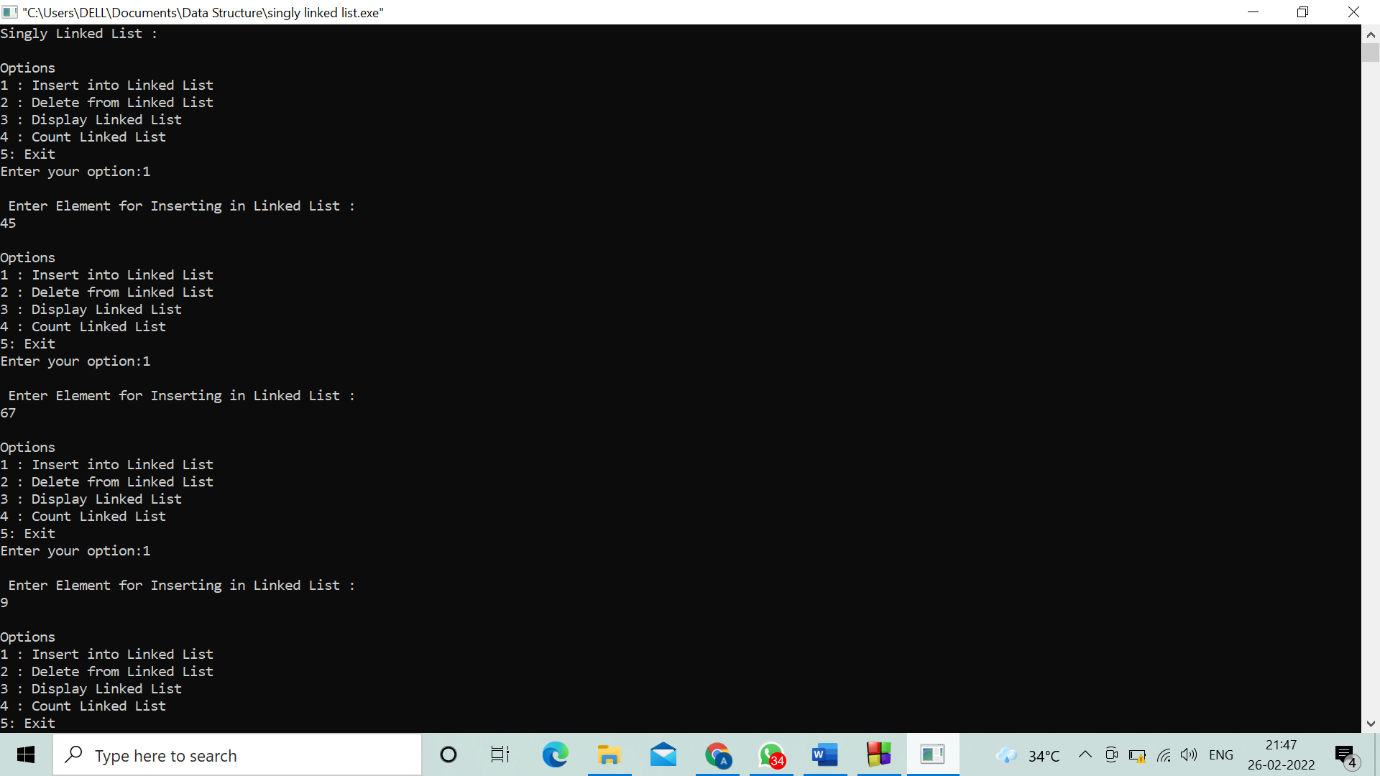
}

printf("\n No Of Items In Linked List : %d\n", count);

return count;

}

Output:



**Program 15. write a program to implement doubly linked list**

// Doubly Linked List

#include <stdio.h>

#include <stdlib.h>

struct node {

int info;

struct node \*prev, \*next;

};

struct node\* start = NULL;

void traverse()

{

if (start == NULL) {

printf("\nList is empty\n");

return;

}

struct node\* temp;

temp = start;

while (temp != NULL) {

printf("Data = %d \n", temp->info);

temp = temp->next;

}

}

void insertAtFront()

{

int data;

struct node\* temp;

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

printf("\nEnter number to be inserted: ");

scanf("%d", &data);

temp->info = data;

temp->prev = NULL;

temp->next = start;

start = temp;

}

void insertAtEnd()

{

int data;

struct node \*temp, \*trav;

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

temp->prev = NULL;

temp->next = NULL;

printf("\nEnter number to be inserted: ");

scanf("%d", &data);

temp->info = data;

temp->next = NULL;

trav = start;

if (start == NULL) {

start = temp;

}

else {

while (trav->next != NULL)

trav = trav->next;

temp->prev = trav;

trav->next = temp;

}

}

void insertAtPosition()

{

int data, pos, i = 1;

struct node \*temp, \*newnode;

newnode = malloc(sizeof(struct node));

newnode->next = NULL;

newnode->prev = NULL;

printf("\nEnter position : ");

scanf("%d", &pos);

if (start == NULL) {

start = newnode;

newnode->prev = NULL;

newnode->next = NULL;

}

else if (pos == 1) {

insertAtFront();

}

else {

printf("\nEnter number to be inserted: ");

scanf("%d", &data);

newnode->info = data;

temp = start;

while (i < pos - 1) {

temp = temp->next;

i++;

}

newnode->next = temp->next;

newnode->prev = temp;

temp->next = newnode;

temp->next->prev = newnode;

}

}

void deleteFirst()

{

struct node\* temp;

if (start == NULL)

printf("\nList is empty\n");

else {

temp = start;

start = start->next;

if (start != NULL)

start->prev = NULL;

free(temp);

}

}

void deleteEnd()

{

struct node\* temp;

if (start == NULL)

printf("\nList is empty\n");

temp = start;

while (temp->next != NULL)

temp = temp->next;

if (start->next == NULL)

start = NULL;

else {

temp->prev->next = NULL;

free(temp);

}

}

void deletePosition()

{

int pos, i = 1;

struct node \*temp, \*position;

temp = start;

if (start == NULL)

printf("\nList is empty\n");

else {

printf("\nEnter position : ");

scanf("%d", &pos);

deleteFirst();

if (start != NULL) {

start->prev = NULL;

}

free(position);

return;

}

while (i < pos - 1) {

temp = temp->next;

i++;

}

position = temp->next;

if (position->next != NULL)

position->next->prev = temp;

temp->next = position->next;

free(position);

}

int main()

{

int choice;

while (1) {

printf("\t1. Insert at The Beginning \n");

printf("\t2. Insert at The End \n");

printf("\t3. Insert at Any Position \n");

printf("\t4. Delete from Beginning \n");

printf("\t5. Delete from End \n");

printf("\t6.Delete from Any position \n");

printf("\t7. Display Linked List \n");

printf("\t8. To Exit \n");

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

scanf("%d", &choice);

switch (choice) {

case 1:insertAtFront();

break;

case 2:insertAtEnd();

break;

case 3:insertAtPosition();

break;

case 4:deleteFirst();

break;

case 5:deleteEnd();

break;

case 6:deletePosition();

break;

case 7:traverse();

break;

case 8:exit(1);

break;

default:printf("Incorrect Choice. Try Again \n");

continue;

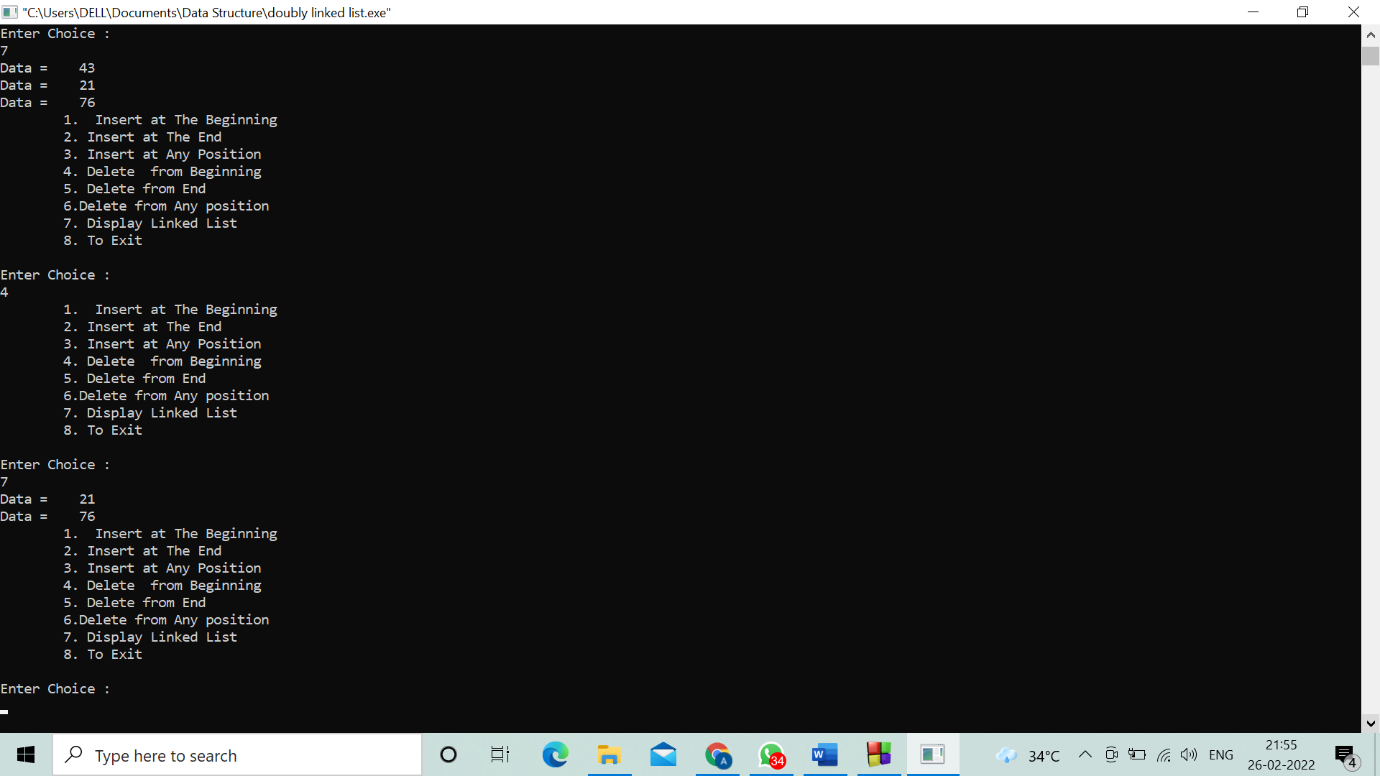
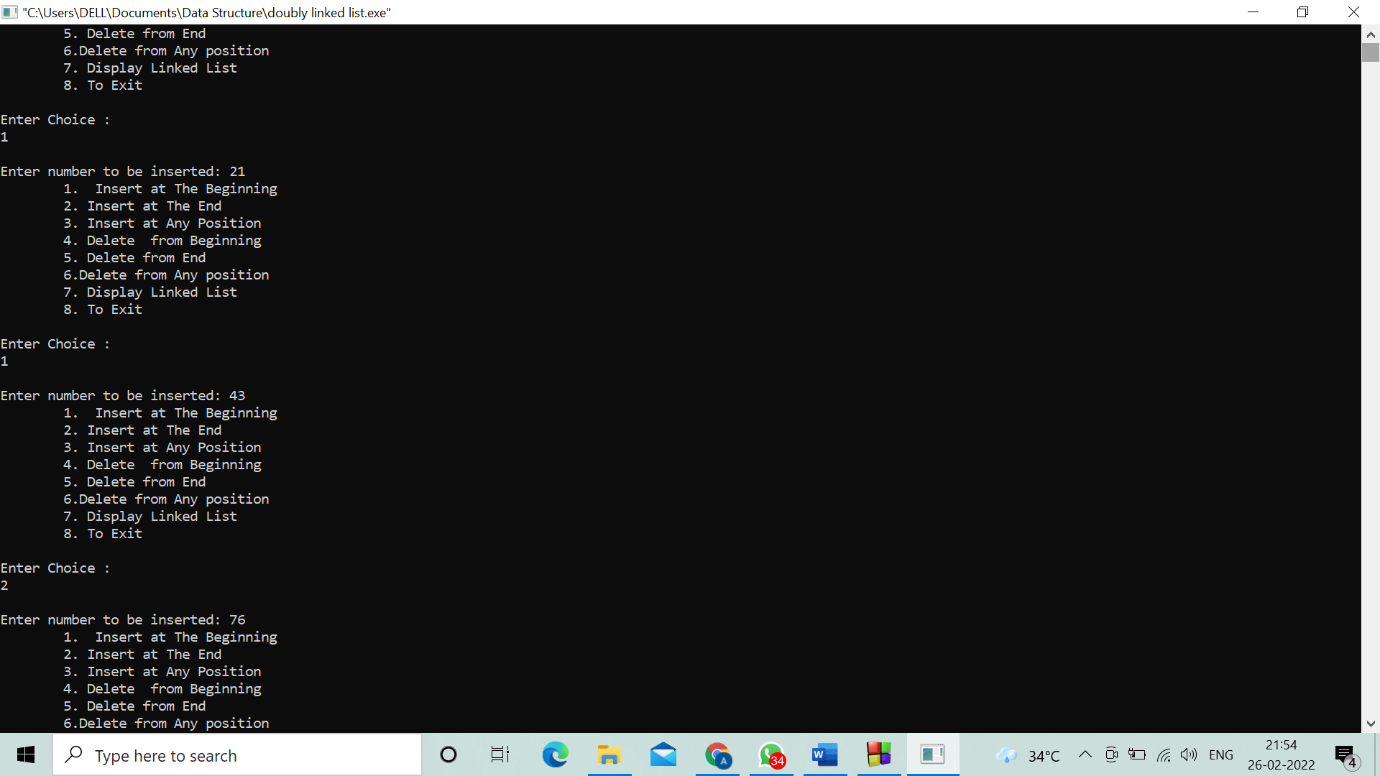
}

}

return 0;

}

Output:



**Program 16. Linear search of an array**

#include <stdio.h>

int main()

{

int array[10], search, c, number;

printf("Enter the number of elements in array\n");

scanf("%d",&number);

printf("Enter %d numbers\n", number);

for ( c = 0 ; c < number ; c++ )

scanf("%d",&array[c]);

printf("Enter the number to search\n");

scanf("%d",&search);

for ( c = 0 ; c < number ; c++ )

{

if ( array[c] == search )

{

printf("%d is present at location %d.\n", search, c+1);

break;

}

}

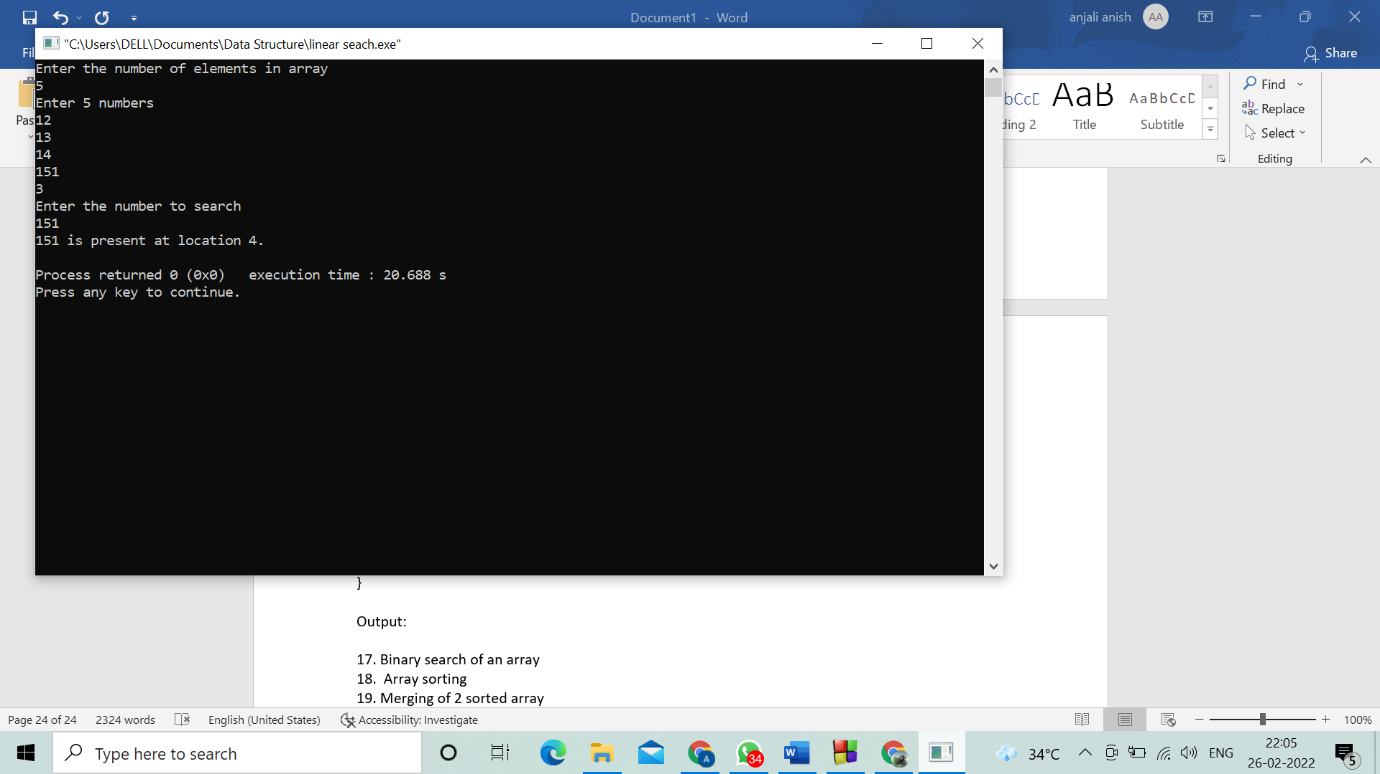
if ( c == number )

printf("%d is not present in array.\n", search);

return 0;

}

Output:



**Program 17. Binary search of an array**

#include <stdio.h>

int main()

{

int i, low, high, mid, n, key, array[100];

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

scanf("%d",&n);

printf("Enter %d integers: \n", n);

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

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

printf("Enter value to find: \n");

scanf("%d", &key);

low = 0;

high = n - 1;

mid = (low+high)/2;

while (low <= high) {

if(array[mid] < key)

low = mid + 1;

else if (array[mid] == key) {

printf("%d found at location %d. \n", key, mid+1);

break;

}

else

high = mid - 1;

mid = (low + high)/2;

}

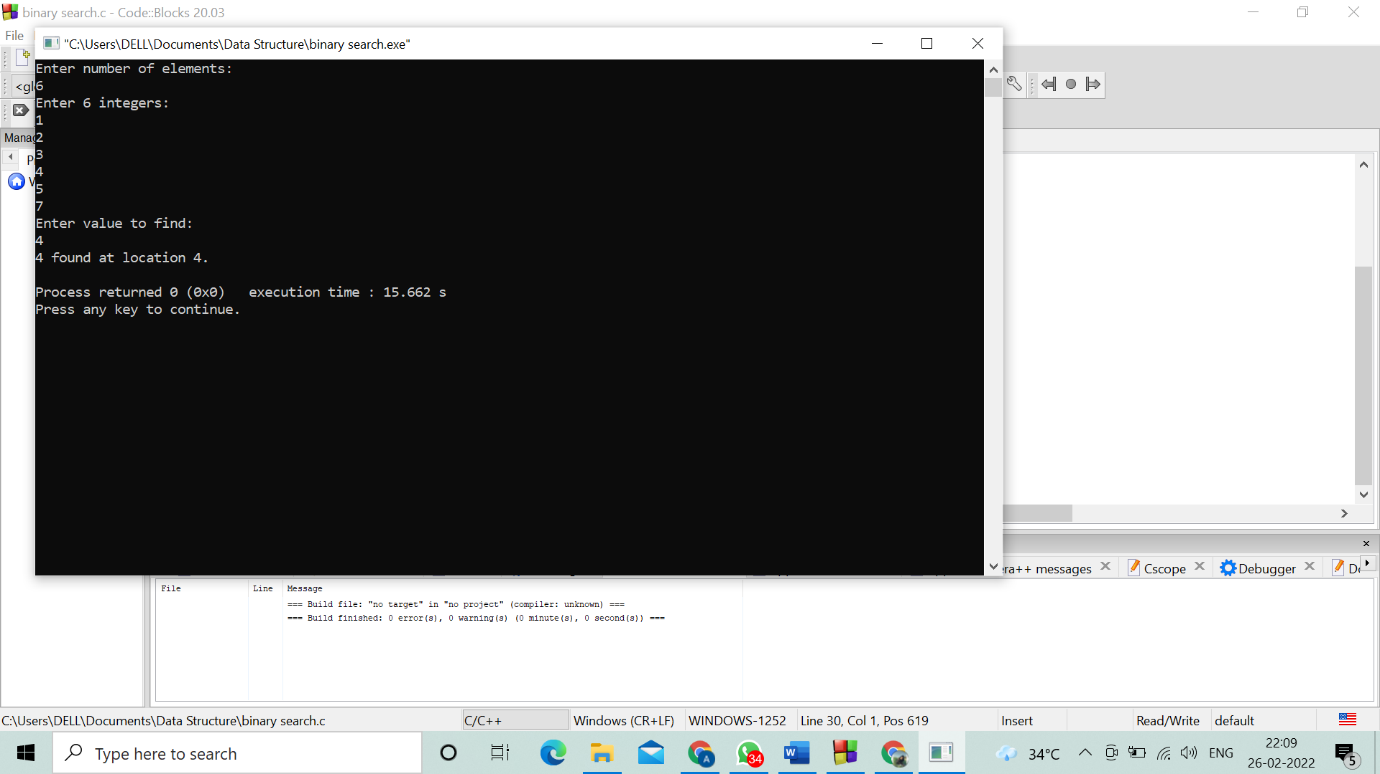
if(low > high)

printf("Not found! %d isn't present in the list. \n", key);

return 0;

}

Output:



**Program 18. Array sorting**

//C program to accept N numbers and arrange them in an ascending order

#include <stdio.h>

void main()

{

int i, j, a, n, number[30];

printf("Enter the value of N \n");

scanf("%d", &n);

printf("Enter the numbers \n");

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

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

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

{

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

{

if (number[i] > number[j])

{

a = number[i];

number[i] = number[j];

number[j] = a;

}

}

}

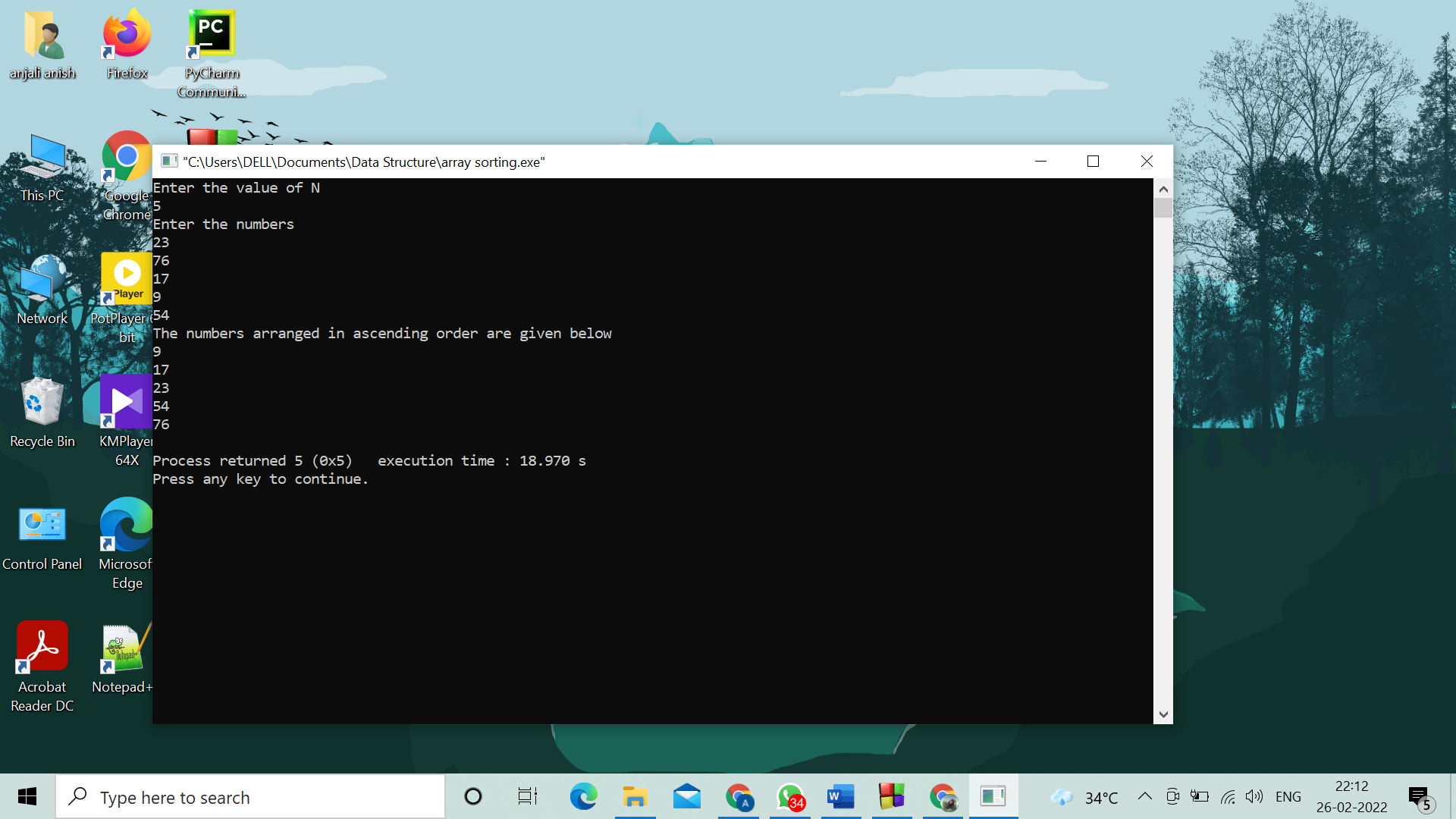
printf("The numbers arranged in ascending order are given below \n");

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

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

}

Output:



**Program 19. Merging of 2 sorted array**

#include<stdio.h>

#include<conio.h>

int main()

{

int a[10], b[10], c[20], i, j, limitC, temp;

printf("Enter 10 elements in array A:\n");

for(i=0; i<10; i++)

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

printf("Enter 10 elements in array B:\n");

for(i=0; i<10; i++)

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

printf("\nElements of Array A are:\n");

for(i=0; i<10; i++)

{

if(i==9)

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

else

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

}

printf("\n\nElements of Array B are:\n");

for(i=0; i<10; i++)

{

if(i==9)

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

else

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

}

// merging the two arrays

for(i=0; i<10; i++)

c[i] = a[i];

for(j=0; j<10; j++)

{

c[i] = b[j];

i++;

}

// sorting the merged array

for(j=0; j<19; j++)

{

for(i=0; i<19; i++)

{

if(c[i]>c[i+1])

{

temp = c[i];

c[i] = c[i+1];

c[i+1] = temp;

}

}

}

printf("\n\nElements of Array C are:\n");

for(i=0; i<20; i++)

{

if(i==19)

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

else

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

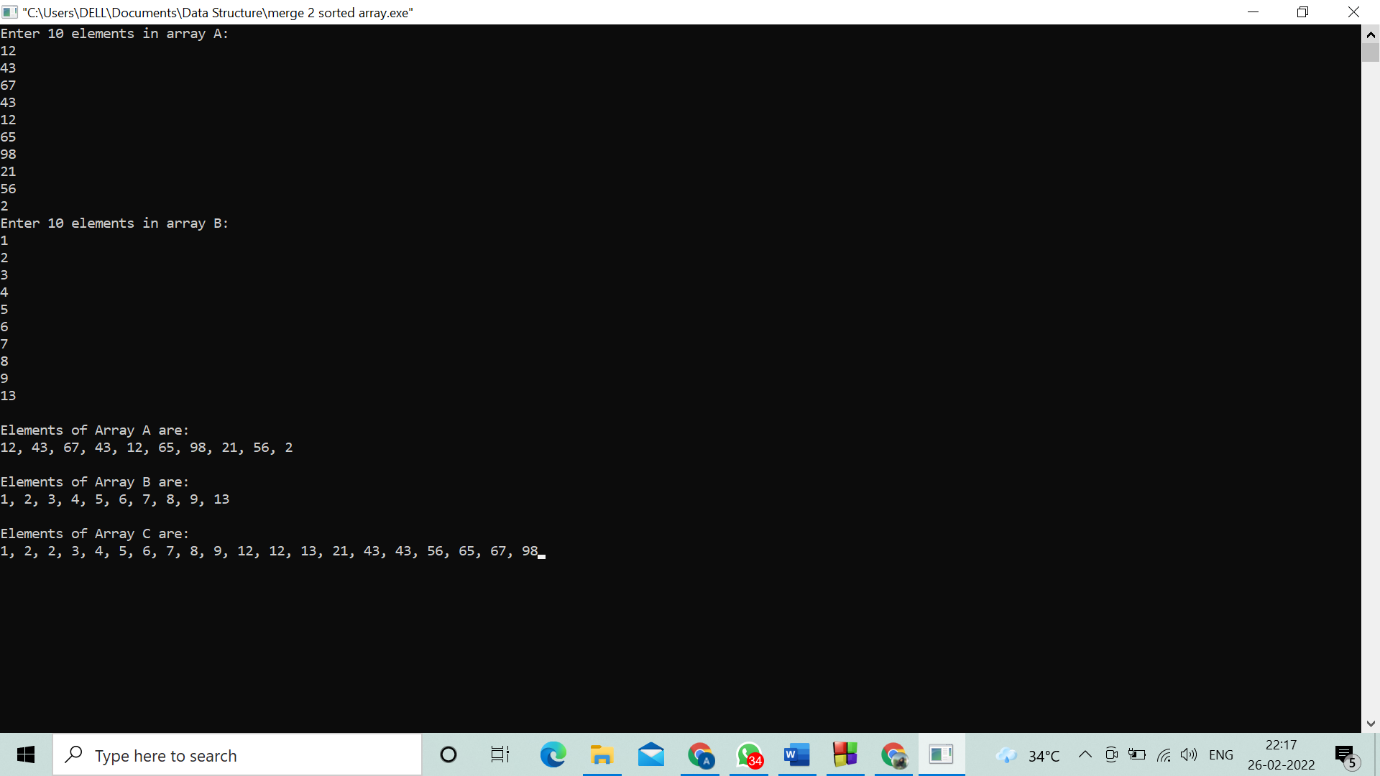
}

getch();

return 0;

}

Output:



**Program 20. Set implementation using bit string (union, insertion, difference)**

#include<stdio.h>

#include<conio.h>

main()

{

int i,j,k,p,ch,n1,n2,set1[10],set2[10], set3[20],flag;

int wish;

printf("\n Enter the size of sets1 \n");

scanf("%d",&n1);

printf("\n Enter the element of set1 \n" );

for(i=0;i<n1;i++)

scanf("%d",&set1[i]);

printf("\n Enter the size of sets2 \n");

scanf("%d",&n2);

printf("\n Enter the elements of set2 \n" );

for(i=0;i<n2;i++)

scanf("%d",&set2[i]);

do

{

printf("\n Menu for set operations");

printf("\n Press 1 for UNION");

printf("\n press 2 for INTERSECTION");

printf("\n press 3 for DIFFERENCE");

printf("\n Enter your Choice");

scanf("%d",&ch);

switch(ch)

{

case 1://for union

k=0;

for(i=0;i<n1;i++)

{

set3[k]=set1[i];

k++;

}

for(i=0;i<n2;i++)

{

flag=1;

for(j=0;j<n1;j++)

{

if(set2[i]==set1[j])

{

flag=0;

break;

}

}

if(flag==1)

{

set3[k]=set2[i];

k++;

}

}

p=k;

for(k=0;k <p;k++)

{

printf(" %d",set3[k]);

}

break;

case 2:

k=0;

for(i=0;i<n2;i++)

{

flag=1;

for(j=0;j<n1;j++)

{

if(set2[i]==set1[j])

{

flag=0;

break;

}

}

if(flag==0)

{

set3[k]=set2[i];

k++;

}

}

p=k;

for(k=0;k <p;k++)

{

printf("%d",set3[k]);

}

break;

case 3:

k=0;

for(i=0;i<n1;i++)

{

flag=1;

for(j=0;j<n2;j++)

{

if(set1[i]==set2[j])

{

flag=0;

break;

}

}

if(flag==1)

{

set3[k]=set1[i];

k++;

}

}

p=k;

for(k=0;k <p;k++)

{

printf(" %d",set3[k]);

}

break;

}

printf("\n Do you want to continue(0/1)? ");

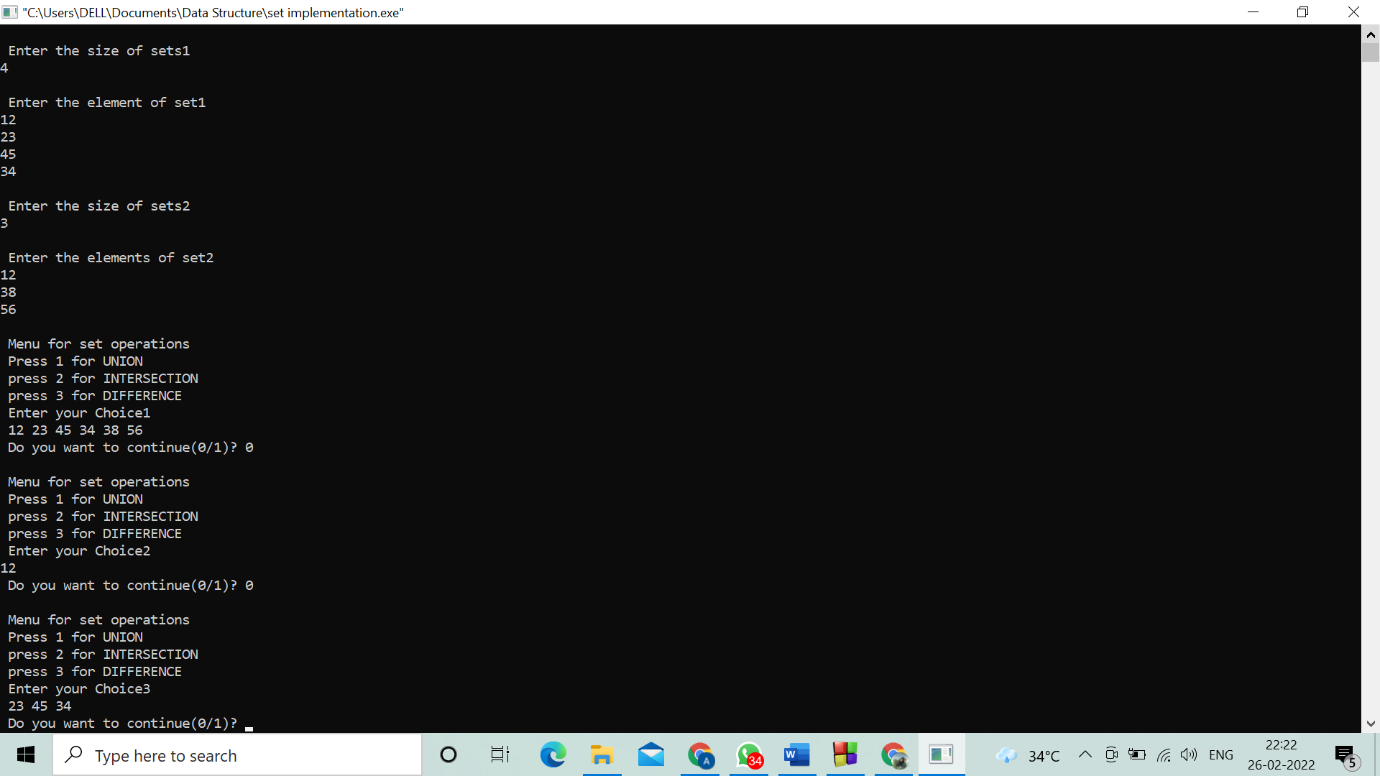
scanf("%d",&wish);

}

while(wish==0);

}

Output:



**Program 21. Disjoint set implementation (create, union, find)**

#include<stdio.h>

#include<conio.h>

struct disjointSet {

int parent[10];

int rank[10];

int n;

}

dis;

void makeset()

{

int i;

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

dis.parent[i]=i;

dis.rank[i]=0;

}

void displayset()

{

int i;

printf("\nparent array\n");

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

{

printf("%d",dis.parent[i]);

}

printf("\nrank of array\n");

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

{

printf("%d",dis.rank[i]);

}

printf("\n");

}

int find(int x)

{

if(dis.parent[x]!=x)

{

dis.parent[x]=find(dis.parent[x]);

}

return dis.parent[x];

}

void Union(int x,int y)

{

int xset=find(x) , yset=find(y);

if(xset==yset)

return;

if(dis.rank[xset]<dis.rank[yset])

{

dis.parent[xset]=yset;

dis.rank[xset]=-1;

}

else if(dis.rank[xset]>dis.rank[yset])

{

dis.parent[yset]=xset;

dis.rank[yset]=-1;

}

else

{

dis.parent[yset]=xset;

dis.rank[xset]=dis.rank[xset]+1;

dis.rank[yset]=-1;

}

}

int main()

{

int x,y,n;

printf("\nenter number of elements :\n");

scanf("%d",&dis.n);

makeset();

int ch,w;

do{

printf("\n1.UNION\n2.FIND \n3.DISPLAY");

printf("\nenter choice :");

scanf("%d",&ch);

switch(ch)

{

case 1:

printf("\nenter elements to perform union :");

scanf("%d%d",&x,&y);

Union(x,y);

break;

case 2:

printf("\nenter elements to check if connected components :");

scanf("%d%d",&x,&y);

if(find(x)==find(y))

printf("\nconnected components !");

else

printf("\n no connected components !");

break;

case 3:

displayset();

break;

}

printf("\n do you want to continue ?(1/0)");

scanf("%d",&w);

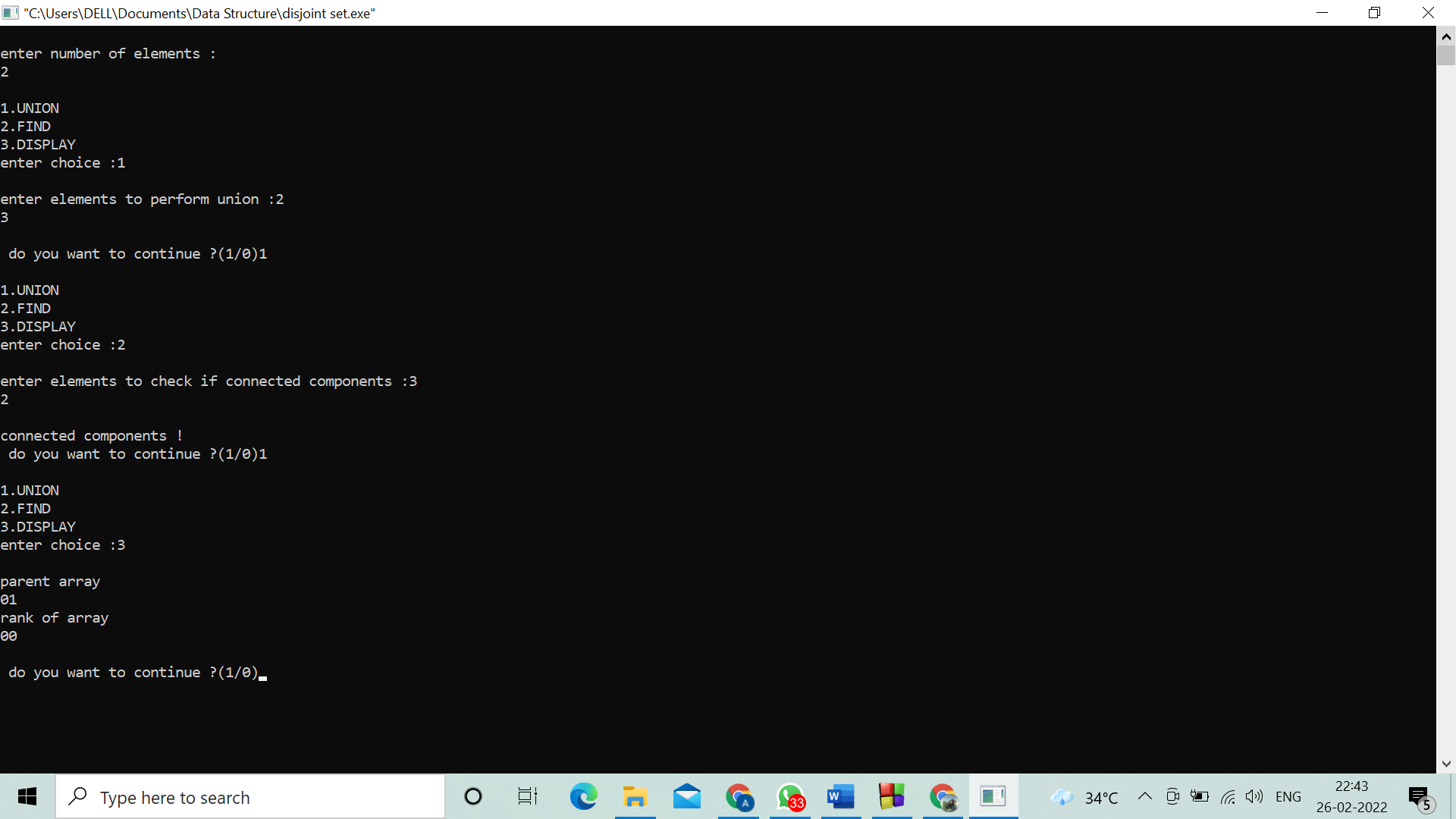
}

while(w==1);

return 0;

}

Output:



**Program 22. Implementation of binary search tree**

#include <stdio.h>

#include <stdlib.h>

struct node {

int key;

struct node \*left, \*right;

};

struct node \*newNode(int item) {

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

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

void inorder(struct node \*root) {

if (root != NULL) {

inorder(root->left);

printf("%d -> ", root->key);

inorder(root->right);

}

}

struct node \*insert(struct node \*node, int key) {

if (node == NULL) return newNode(key);

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

struct node \*minValueNode(struct node \*node) {

struct node \*current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

struct node \*deleteNode(struct node \*root, int key) {

if (root == NULL) return root;

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct node \*temp = root->left;

free(root);

return temp;

}

struct node \*temp = minValueNode(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

return root;

}

void main() {

struct node \*root = NULL;

int choice, n;

while(1){

printf(" 1.Insertion");

printf("\n 2.Deleteion");

printf("\n 3.Traversal");

printf("\n 4.Exit");

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

scanf("%d",&choice);

switch(choice)

{

case 1: printf("Enter the element to be Inserted:");

scanf("%d",&n);

root = insert(root, n);

break;

case 2: printf("Enter the element to be Deleted:");

scanf("%d",&n);

root = deleteNode(root, n);

break;

case 3: printf("Inorder traversal: ");

inorder(root);

break;

case 4:

exit(0);

break;

default:

printf("n Wrong Choice:n");

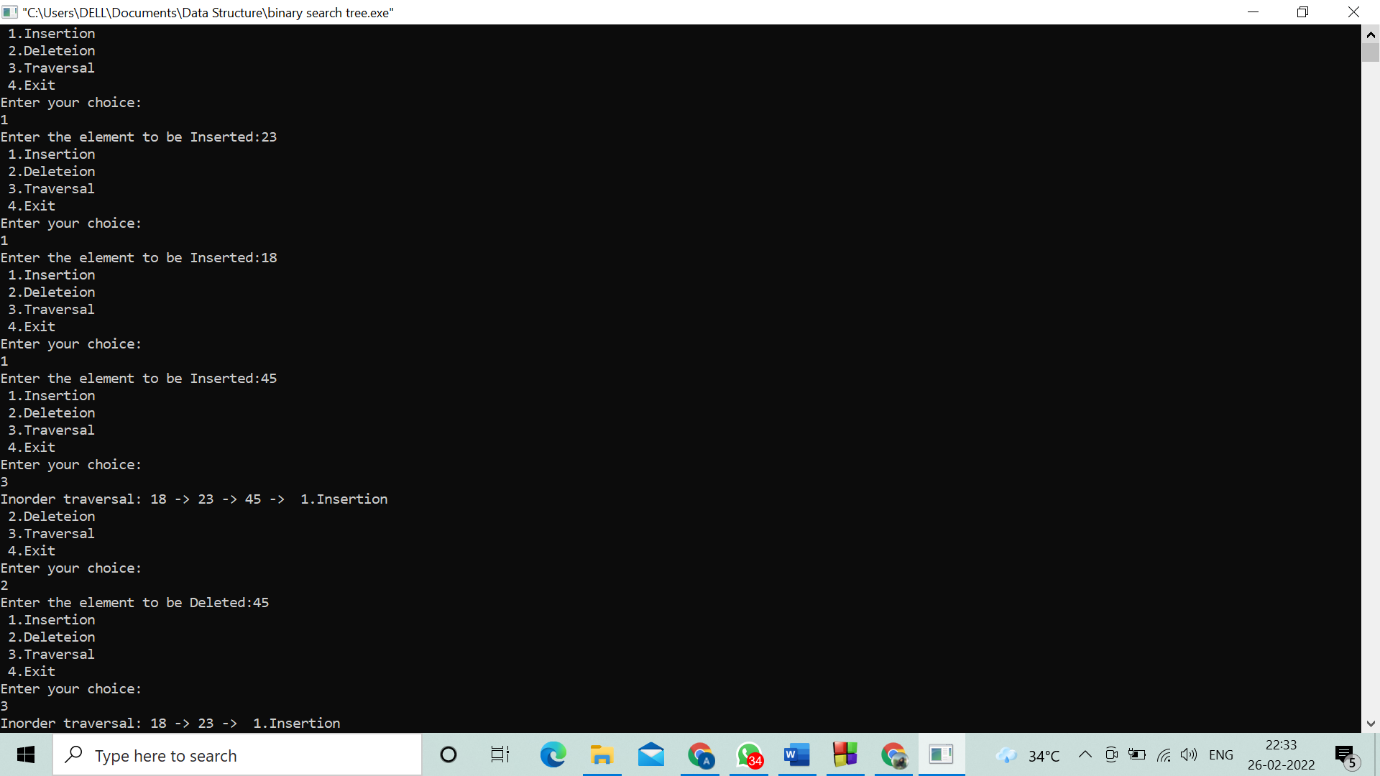
break;

}

}

}

Output:



**Program 23. Implementation of balanced binary search tree.**

// Checking if a binary tree is height balanced in C

#include <stdio.h>

#include <stdlib.h>

#define bool int

struct node {

int item;

struct node \*left;

struct node \*right;

};

struct node \*newNode(int item) {

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

node->item = item;

node->left = NULL;

node->right = NULL;

return (node);

}

bool checkHeightBalance(struct node \*root, int \*height) {

int leftHeight = 0, rightHeight = 0;

int l = 0, r = 0;

if (root == NULL) {

\*height = 0;

return 1;

}

l = checkHeightBalance(root->left, &leftHeight);

r = checkHeightBalance(root->right, &rightHeight);

\*height = (leftHeight > rightHeight ? leftHeight : rightHeight) + 1;

if ((leftHeight - rightHeight >= 2) || (rightHeight - leftHeight >= 2))

return 0;

else

return l && r;

}

int main() {

int height = 0;

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

root->left->right = newNode(5);

if (checkHeightBalance(root, &height))

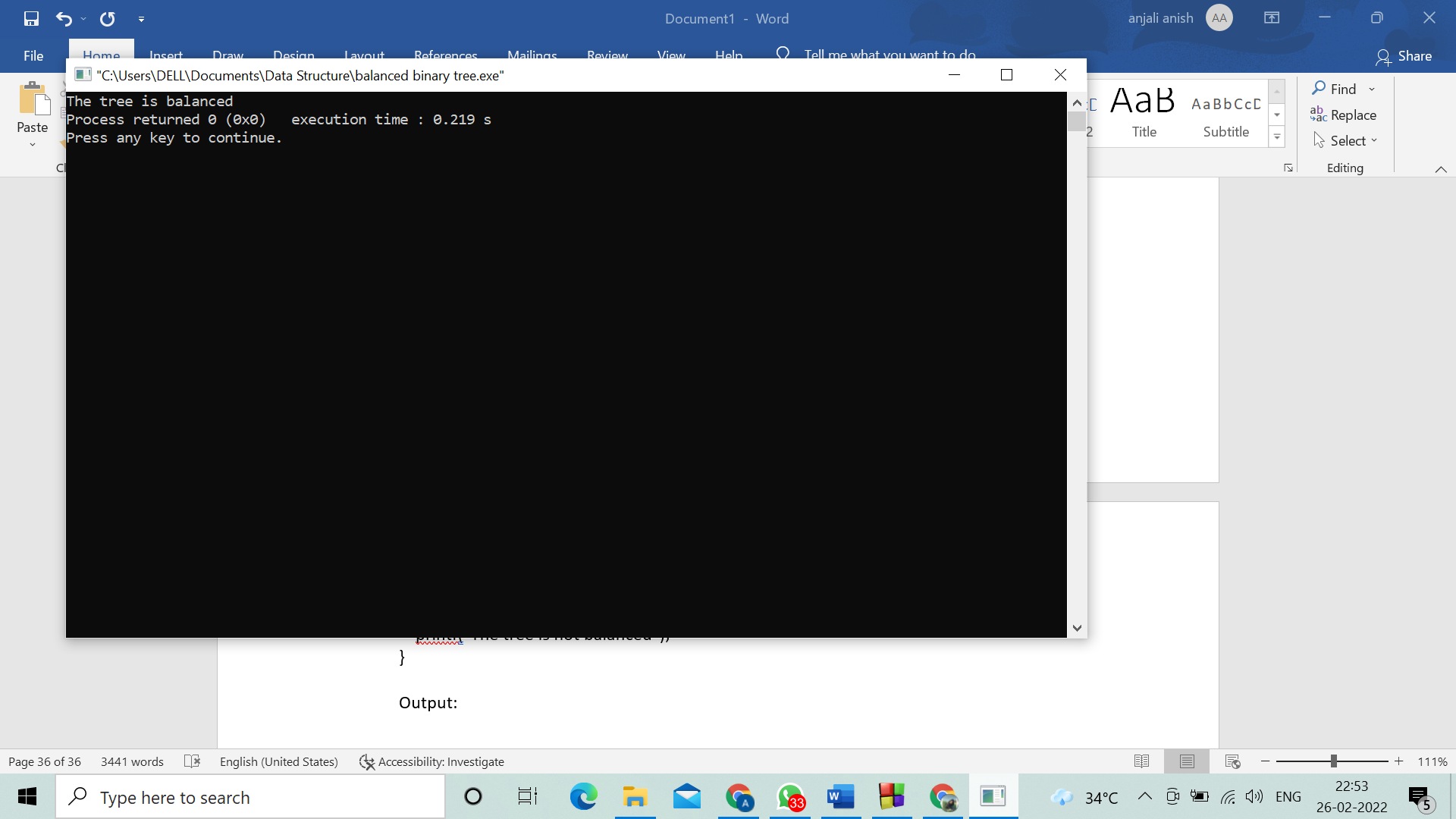
printf("The tree is balanced");

else

printf("The tree is not balanced");

}

Output:



**Program 24. Red black tree implementation**

// Implementing Red-Black Tree in C

#include <stdio.h>

#include <stdlib.h>

enum nodeColor {

RED,

BLACK

};

struct rbNode {

int data, color;

struct rbNode \*link[2];

};

struct rbNode \*root = NULL;

// Create a red-black tree

struct rbNode \*createNode(int data) {

struct rbNode \*newnode;

newnode = (struct rbNode \*)malloc(sizeof(struct rbNode));

newnode->data = data;

newnode->color = RED;

newnode->link[0] = newnode->link[1] = NULL;

return newnode;

}

// Insert an node

void insertion(int data) {

struct rbNode \*stack[98], \*ptr, \*newnode, \*xPtr, \*yPtr;

int dir[98], ht = 0, index;

ptr = root;

if (!root) {

root = createNode(data);

return;

}

stack[ht] = root;

dir[ht++] = 0;

while (ptr != NULL) {

if (ptr->data == data) {

printf("Duplicates Not Allowed!!\n");

return;

}

index = (data - ptr->data) > 0 ? 1 : 0;

stack[ht] = ptr;

ptr = ptr->link[index];

dir[ht++] = index;

}

stack[ht - 1]->link[index] = newnode = createNode(data);

while ((ht >= 3) && (stack[ht - 1]->color == RED)) {

if (dir[ht - 2] == 0) {

yPtr = stack[ht - 2]->link[1];

if (yPtr != NULL && yPtr->color == RED) {

stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK;

ht = ht - 2;

} else {

if (dir[ht - 1] == 0) {

yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[1];

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

yPtr->link[0] = xPtr;

stack[ht - 2]->link[0] = yPtr;

}

xPtr = stack[ht - 2];

xPtr->color = RED;

yPtr->color = BLACK;

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

yPtr->link[1] = xPtr;

if (xPtr == root) {

root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

} else {

yPtr = stack[ht - 2]->link[0];

if ((yPtr != NULL) && (yPtr->color == RED)) {

stack[ht - 2]->color = RED;

stack[ht - 1]->color = yPtr->color = BLACK;

ht = ht - 2;

} else {

if (dir[ht - 1] == 1) {

yPtr = stack[ht - 1];

} else {

xPtr = stack[ht - 1];

yPtr = xPtr->link[0];

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

yPtr->link[1] = xPtr;

stack[ht - 2]->link[1] = yPtr;

}

xPtr = stack[ht - 2];

yPtr->color = BLACK;

xPtr->color = RED;

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

yPtr->link[0] = xPtr;

if (xPtr == root) {

root = yPtr;

} else {

stack[ht - 3]->link[dir[ht - 3]] = yPtr;

}

break;

}

}

}

root->color = BLACK;

}

// Delete a node

void deletion(int data) {

struct rbNode \*stack[98], \*ptr, \*xPtr, \*yPtr;

struct rbNode \*pPtr, \*qPtr, \*rPtr;

int dir[98], ht = 0, diff, i;

enum nodeColor color;

if (!root) {

printf("Tree not available\n");

return;

}

ptr = root;

while (ptr != NULL) {

if ((data - ptr->data) == 0)

break;

diff = (data - ptr->data) > 0 ? 1 : 0;

stack[ht] = ptr;

dir[ht++] = diff;

ptr = ptr->link[diff];

}

if (ptr->link[1] == NULL) {

if ((ptr == root) && (ptr->link[0] == NULL)) {

free(ptr);

root = NULL;

} else if (ptr == root) {

root = ptr->link[0];

free(ptr);

} else {

stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];

}

} else {

xPtr = ptr->link[1];

if (xPtr->link[0] == NULL) {

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

color = xPtr->color;

xPtr->color = ptr->color;

ptr->color = color;

if (ptr == root) {

root = xPtr;

} else {

stack[ht - 1]->link[dir[ht - 1]] = xPtr;

}

dir[ht] = 1;

stack[ht++] = xPtr;

} else {

i = ht++;

while (1) {

dir[ht] = 0;

stack[ht++] = xPtr;

yPtr = xPtr->link[0];

if (!yPtr->link[0])

break;

xPtr = yPtr;

}

dir[i] = 1;

stack[i] = yPtr;

if (i > 0)

stack[i - 1]->link[dir[i - 1]] = yPtr;

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

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

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

if (ptr == root) {

root = yPtr;

}

color = yPtr->color;

yPtr->color = ptr->color;

ptr->color = color;

}

}

if (ht < 1)

return;

if (ptr->color == BLACK) {

while (1) {

pPtr = stack[ht - 1]->link[dir[ht - 1]];

if (pPtr && pPtr->color == RED) {

pPtr->color = BLACK;

break;

}

if (ht < 2)

break;

if (dir[ht - 2] == 0) {

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

if (!rPtr)

break;

if (rPtr->color == RED) {

stack[ht - 1]->color = RED;

rPtr->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0];

rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 0;

stack[ht] = stack[ht - 1];

stack[ht - 1] = rPtr;

ht++;

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

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&

(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {

rPtr->color = RED;

} else {

if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {

qPtr = rPtr->link[0];

rPtr->color = RED;

qPtr->color = BLACK;

rPtr->link[0] = qPtr->link[1];

qPtr->link[1] = rPtr;

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

}

rPtr->color = stack[ht - 1]->color;

stack[ht - 1]->color = BLACK;

rPtr->link[1]->color = BLACK;

stack[ht - 1]->link[1] = rPtr->link[0];

rPtr->link[0] = stack[ht - 1];

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

} else {

rPtr = stack[ht - 1]->link[0];

if (!rPtr)

break;

if (rPtr->color == RED) {

stack[ht - 1]->color = RED;

rPtr->color = BLACK;

stack[ht - 1]->link[0] = rPtr->link[1];

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

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

dir[ht] = 1;

stack[ht] = stack[ht - 1];

stack[ht - 1] = rPtr;

ht++;

rPtr = stack[ht - 1]->link[0];

}

if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&

(!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {

rPtr->color = RED;

} else {

if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {

qPtr = rPtr->link[1];

rPtr->color = RED;

qPtr->color = BLACK;

rPtr->link[1] = qPtr->link[0];

qPtr->link[0] = rPtr;

rPtr = stack[ht - 1]->link[0] = qPtr;

}

rPtr->color = stack[ht - 1]->color;

stack[ht - 1]->color = BLACK;

rPtr->link[0]->color = BLACK;

stack[ht - 1]->link[0] = rPtr->link[1];

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

if (stack[ht - 1] == root) {

root = rPtr;

} else {

stack[ht - 2]->link[dir[ht - 2]] = rPtr;

}

break;

}

}

ht--;

}

}

}

// Print the inorder traversal of the tree

void inorderTraversal(struct rbNode \*node) {

if (node) {

inorderTraversal(node->link[0]);

printf("%d ", node->data);

inorderTraversal(node->link[1]);

}

return;

}

// Driver code

int main() {

int ch, data;

while (1) {

printf("1. Insertion\t2. Deletion\n");

printf("3. Traverse\t4. Exit");

printf("\nEnter your choice:");

scanf("%d", &ch);

switch (ch) {

case 1:

printf("Enter the element to insert:");

scanf("%d", &data);

insertion(data);

break;

case 2:

printf("Enter the element to delete:");

scanf("%d", &data);

deletion(data);

break;

case 3:

inorderTraversal(root);

printf("\n");

break;

case 4:

exit(0);

default:

printf("Not available\n");

break;

}

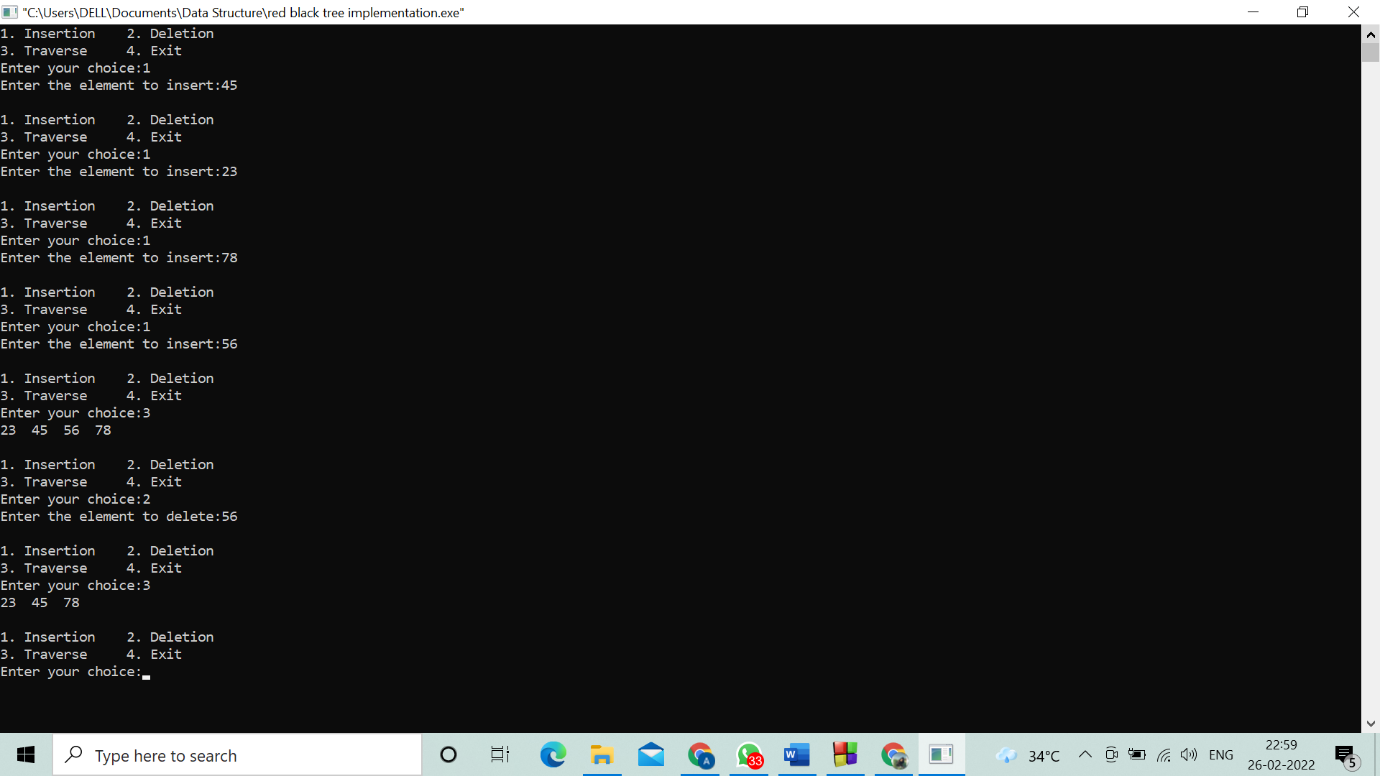
printf("\n");

}

return 0;

}

Output:



**Program 25. B-tree implementation**

// Searching a key on a B-tree in C

#include <stdio.h>

#include <stdlib.h>

#define MAX 3

#define MIN 2

struct BTreeNode {

int val[MAX + 1], count;

struct BTreeNode \*link[MAX + 1];

};

struct BTreeNode \*root;

// Create a node

struct BTreeNode \*createNode(int val, struct BTreeNode \*child) {

struct BTreeNode \*newNode;

newNode = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode));

newNode->val[1] = val;

newNode->count = 1;

newNode->link[0] = root;

newNode->link[1] = child;

return newNode;

}

// Insert node

void insertNode(int val, int pos, struct BTreeNode \*node,

struct BTreeNode \*child) {

int j = node->count;

while (j > pos) {

node->val[j + 1] = node->val[j];

node->link[j + 1] = node->link[j];

j--;

}

node->val[j + 1] = val;

node->link[j + 1] = child;

node->count++;

}

// Split node

void splitNode(int val, int \*pval, int pos, struct BTreeNode \*node,

struct BTreeNode \*child, struct BTreeNode \*\*newNode) {

int median, j;

if (pos > MIN)

median = MIN + 1;

else

median = MIN;

\*newNode = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode));

j = median + 1;

while (j <= MAX) {

(\*newNode)->val[j - median] = node->val[j];

(\*newNode)->link[j - median] = node->link[j];

j++;

}

node->count = median;

(\*newNode)->count = MAX - median;

if (pos <= MIN) {

insertNode(val, pos, node, child);

} else {

insertNode(val, pos - median, \*newNode, child);

}

\*pval = node->val[node->count];

(\*newNode)->link[0] = node->link[node->count];

node->count--;

}

// Set the value

int setValue(int val, int \*pval,

struct BTreeNode \*node, struct BTreeNode \*\*child) {

int pos;

if (!node) {

\*pval = val;

\*child = NULL;

return 1;

}

if (val < node->val[1]) {

pos = 0;

} else {

for (pos = node->count;

(val < node->val[pos] && pos > 1); pos--)

;

if (val == node->val[pos]) {

printf("Duplicates are not permitted\n");

return 0;

}

}

if (setValue(val, pval, node->link[pos], child)) {

if (node->count < MAX) {

insertNode(\*pval, pos, node, \*child);

} else {

splitNode(\*pval, pval, pos, node, \*child, child);

return 1;

}

}

return 0;

}

// Insert the value

void insert(int val) {

int flag, i;

struct BTreeNode \*child;

flag = setValue(val, &i, root, &child);

if (flag)

root = createNode(i, child);

}

// Search node

void search(int val, int \*pos, struct BTreeNode \*myNode) {

if (!myNode) {

return;

}

if (val < myNode->val[1]) {

\*pos = 0;

} else {

for (\*pos = myNode->count;

(val < myNode->val[\*pos] && \*pos > 1); (\*pos)--)

;

if (val == myNode->val[\*pos]) {

printf("%d is found", val);

return;

}

}

search(val, pos, myNode->link[\*pos]);

return;

}

// Traverse then nodes

void traversal(struct BTreeNode \*myNode) {

int i;

if (myNode) {

for (i = 0; i < myNode->count; i++) {

traversal(myNode->link[i]);

printf("%d ", myNode->val[i + 1]);

}

traversal(myNode->link[i]);

}

}

int main() {

int val, ch;

insert(8);

insert(9);

insert(10);

insert(11);

insert(15);

insert(16);

insert(17);

insert(18);

insert(20);

insert(23);

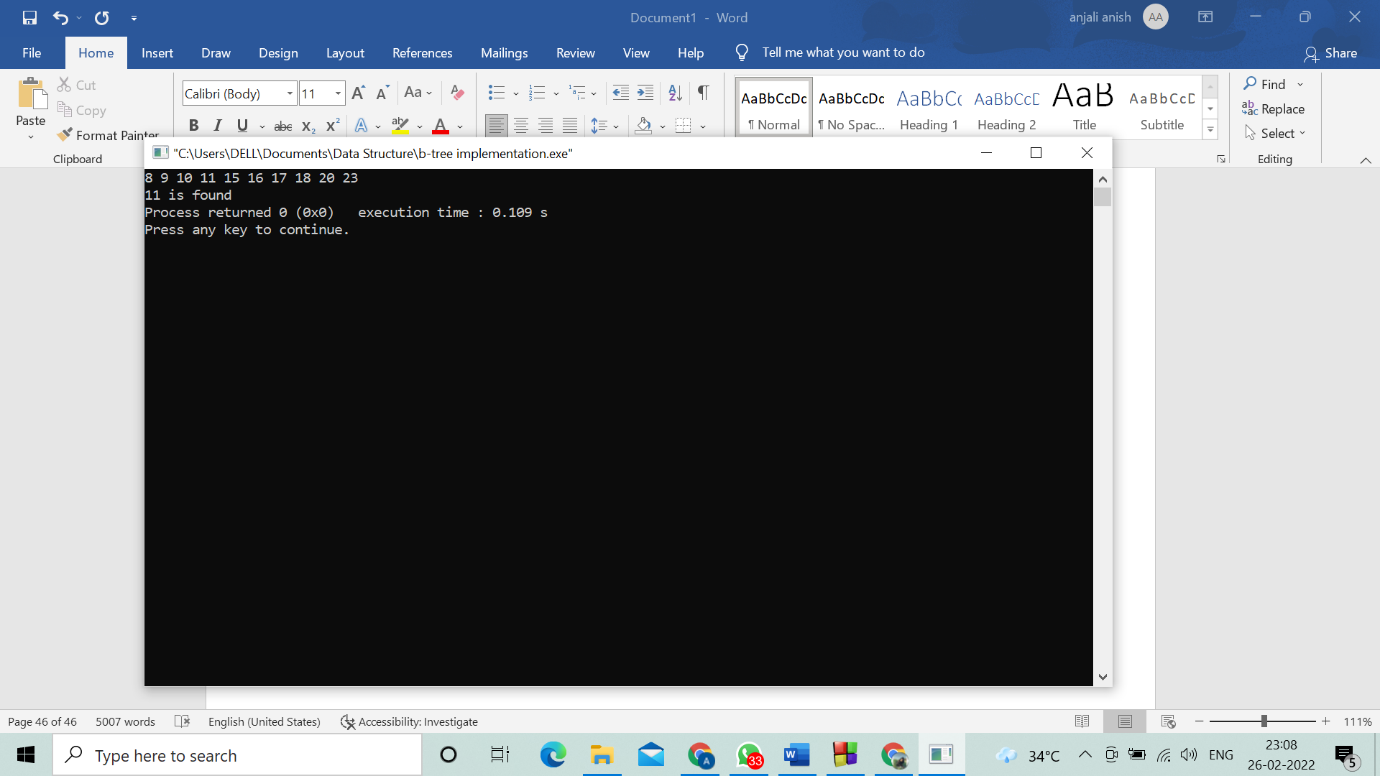
traversal(root);

printf("\n");

search(11, &ch, root);

}

Output:



**Program 26. Binary heap max implementation**

// Max-Heap data structure in C

#include <stdio.h>

int size = 0;

void swap(int \*a, int \*b)

{

int temp = \*b;

\*b = \*a;

\*a = temp;

}

void heapify(int array[], int size, int i)

{

if (size == 1)

{

printf("Single element in the heap");

}

else

{

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < size && array[l] > array[largest])

largest = l;

if (r < size && array[r] > array[largest])

largest = r;

if (largest != i)

{

swap(&array[i], &array[largest]);

heapify(array, size, largest);

}

}

}

void insert(int array[], int newNum)

{

if (size == 0)

{

array[0] = newNum;

size += 1;

}

else

{

array[size] = newNum;

size += 1;

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(array, size, i);

}

}

}

void deleteRoot(int array[], int num)

{

int i;

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

{

if (num == array[i])

break;

}

swap(&array[i], &array[size - 1]);

size -= 1;

for (int i = size / 2 - 1; i >= 0; i--)

{

heapify(array, size, i);

}

}

void printArray(int array[], int size)

{

for (int i = 0; i < size; ++i)

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

printf("\n");

}

int main()

{

int array[10];

insert(array, 3);

insert(array, 4);

insert(array, 9);

insert(array, 5);

insert(array, 2);

printf("Max-Heap array: ");

printArray(array, size);

deleteRoot(array, 4);

printf("After deleting an element: ");

printArray(array, size);

}

Output:



**27. Fibonacci heap implementation**

// Operations on a Fibonacci heap in C

#include <math.h>

#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>

typedef struct \_NODE {

int key;

int degree;

struct \_NODE \*left\_sibling;

struct \_NODE \*right\_sibling;

struct \_NODE \*parent;

struct \_NODE \*child;

bool mark;

bool visited;

} NODE;

typedef struct fibanocci\_heap {

int n;

NODE \*min;

int phi;

int degree;

} FIB\_HEAP;

FIB\_HEAP \*make\_fib\_heap();

void insertion(FIB\_HEAP \*H, NODE \*new, int val);

NODE \*extract\_min(FIB\_HEAP \*H);

void consolidate(FIB\_HEAP \*H);

void fib\_heap\_link(FIB\_HEAP \*H, NODE \*y, NODE \*x);

NODE \*find\_min\_node(FIB\_HEAP \*H);

void decrease\_key(FIB\_HEAP \*H, NODE \*node, int key);

void cut(FIB\_HEAP \*H, NODE \*node\_to\_be\_decrease, NODE \*parent\_node);

void cascading\_cut(FIB\_HEAP \*H, NODE \*parent\_node);

void Delete\_Node(FIB\_HEAP \*H, int dec\_key);

FIB\_HEAP \*make\_fib\_heap() {

FIB\_HEAP \*H;

H = (FIB\_HEAP \*)malloc(sizeof(FIB\_HEAP));

H->n = 0;

H->min = NULL;

H->phi = 0;

H->degree = 0;

return H;

}

// Printing the heap

void print\_heap(NODE \*n) {

NODE \*x;

for (x = n;; x = x->right\_sibling) {

if (x->child == NULL) {

printf("node with no child (%d) \n", x->key);

} else {

printf("NODE(%d) with child (%d)\n", x->key, x->child->key);

print\_heap(x->child);

}

if (x->right\_sibling == n) {

break;

}

}

}

// Inserting nodes

void insertion(FIB\_HEAP \*H, NODE \*new, int val) {

new = (NODE \*)malloc(sizeof(NODE));

new->key = val;

new->degree = 0;

new->mark = false;

new->parent = NULL;

new->child = NULL;

new->visited = false;

new->left\_sibling = new;

new->right\_sibling = new;

if (H->min == NULL) {

H->min = new;

} else {

H->min->left\_sibling->right\_sibling = new;

new->right\_sibling = H->min;

new->left\_sibling = H->min->left\_sibling;

H->min->left\_sibling = new;

if (new->key < H->min->key) {

H->min = new;

}

}

(H->n)++;

}

// Find min node

NODE \*find\_min\_node(FIB\_HEAP \*H) {

if (H == NULL) {

printf(" \n Fibonacci heap not yet created \n");

return NULL;

} else

return H->min;

}

// Union operation

FIB\_HEAP \*unionHeap(FIB\_HEAP \*H1, FIB\_HEAP \*H2) {

FIB\_HEAP \*Hnew;

Hnew = make\_fib\_heap();

Hnew->min = H1->min;

NODE \*temp1, \*temp2;

temp1 = Hnew->min->right\_sibling;

temp2 = H2->min->left\_sibling;

Hnew->min->right\_sibling->left\_sibling = H2->min->left\_sibling;

Hnew->min->right\_sibling = H2->min;

H2->min->left\_sibling = Hnew->min;

temp2->right\_sibling = temp1;

if ((H1->min == NULL) || (H2->min != NULL && H2->min->key < H1->min->key))

Hnew->min = H2->min;

Hnew->n = H1->n + H2->n;

return Hnew;

}

// Calculate the degree

int cal\_degree(int n) {

int count = 0;

while (n > 0) {

n = n / 2;

count++;

}

return count;

}

// Consolidate function

void consolidate(FIB\_HEAP \*H) {

int degree, i, d;

degree = cal\_degree(H->n);

NODE \*A[degree], \*x, \*y, \*z;

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

A[i] = NULL;

}

x = H->min;

do {

d = x->degree;

while (A[d] != NULL) {

y = A[d];

if (x->key > y->key) {

NODE \*exchange\_help;

exchange\_help = x;

x = y;

y = exchange\_help;

}

if (y == H->min)

H->min = x;

fib\_heap\_link(H, y, x);

if (y->right\_sibling == x)

H->min = x;

A[d] = NULL;

d++;

}

A[d] = x;

x = x->right\_sibling;

} while (x != H->min);

H->min = NULL;

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

if (A[i] != NULL) {

A[i]->left\_sibling = A[i];

A[i]->right\_sibling = A[i];

if (H->min == NULL) {

H->min = A[i];

} else {

H->min->left\_sibling->right\_sibling = A[i];

A[i]->right\_sibling = H->min;

A[i]->left\_sibling = H->min->left\_sibling;

H->min->left\_sibling = A[i];

if (A[i]->key < H->min->key) {

H->min = A[i];

}

}

if (H->min == NULL) {

H->min = A[i];

} else if (A[i]->key < H->min->key) {

H->min = A[i];

}

}

}

}

// Linking

void fib\_heap\_link(FIB\_HEAP \*H, NODE \*y, NODE \*x) {

y->right\_sibling->left\_sibling = y->left\_sibling;

y->left\_sibling->right\_sibling = y->right\_sibling;

if (x->right\_sibling == x)

H->min = x;

y->left\_sibling = y;

y->right\_sibling = y;

y->parent = x;

if (x->child == NULL) {

x->child = y;

}

y->right\_sibling = x->child;

y->left\_sibling = x->child->left\_sibling;

x->child->left\_sibling->right\_sibling = y;

x->child->left\_sibling = y;

if ((y->key) < (x->child->key))

x->child = y;

(x->degree)++;

}

// Extract min

NODE \*extract\_min(FIB\_HEAP \*H) {

if (H->min == NULL)

printf("\n The heap is empty");

else {

NODE \*temp = H->min;

NODE \*pntr;

pntr = temp;

NODE \*x = NULL;

if (temp->child != NULL) {

x = temp->child;

do {

pntr = x->right\_sibling;

(H->min->left\_sibling)->right\_sibling = x;

x->right\_sibling = H->min;

x->left\_sibling = H->min->left\_sibling;

H->min->left\_sibling = x;

if (x->key < H->min->key)

H->min = x;

x->parent = NULL;

x = pntr;

} while (pntr != temp->child);

}

(temp->left\_sibling)->right\_sibling = temp->right\_sibling;

(temp->right\_sibling)->left\_sibling = temp->left\_sibling;

H->min = temp->right\_sibling;

if (temp == temp->right\_sibling && temp->child == NULL)

H->min = NULL;

else {

H->min = temp->right\_sibling;

consolidate(H);

}

H->n = H->n - 1;

return temp;

}

return H->min;

}

void cut(FIB\_HEAP \*H, NODE \*node\_to\_be\_decrease, NODE \*parent\_node) {

NODE \*temp\_parent\_check;

if (node\_to\_be\_decrease == node\_to\_be\_decrease->right\_sibling)

parent\_node->child = NULL;

node\_to\_be\_decrease->left\_sibling->right\_sibling = node\_to\_be\_decrease->right\_sibling;

node\_to\_be\_decrease->right\_sibling->left\_sibling = node\_to\_be\_decrease->left\_sibling;

if (node\_to\_be\_decrease == parent\_node->child)

parent\_node->child = node\_to\_be\_decrease->right\_sibling;

(parent\_node->degree)--;

node\_to\_be\_decrease->left\_sibling = node\_to\_be\_decrease;

node\_to\_be\_decrease->right\_sibling = node\_to\_be\_decrease;

H->min->left\_sibling->right\_sibling = node\_to\_be\_decrease;

node\_to\_be\_decrease->right\_sibling = H->min;

node\_to\_be\_decrease->left\_sibling = H->min->left\_sibling;

H->min->left\_sibling = node\_to\_be\_decrease;

node\_to\_be\_decrease->parent = NULL;

node\_to\_be\_decrease->mark = false;

}

void cascading\_cut(FIB\_HEAP \*H, NODE \*parent\_node) {

NODE \*aux;

aux = parent\_node->parent;

if (aux != NULL) {

if (parent\_node->mark == false) {

parent\_node->mark = true;

} else {

cut(H, parent\_node, aux);

cascading\_cut(H, aux);

}

}

}

void decrease\_key(FIB\_HEAP \*H, NODE \*node\_to\_be\_decrease, int new\_key) {

NODE \*parent\_node;

if (H == NULL) {

printf("\n FIbonacci heap not created ");

return;

}

if (node\_to\_be\_decrease == NULL) {

printf("Node is not in the heap");

}

else {

if (node\_to\_be\_decrease->key < new\_key) {

printf("\n Invalid new key for decrease key operation \n ");

} else {

node\_to\_be\_decrease->key = new\_key;

parent\_node = node\_to\_be\_decrease->parent;

if ((parent\_node != NULL) && (node\_to\_be\_decrease->key < parent\_node->key)) {

printf("\n cut called");

cut(H, node\_to\_be\_decrease, parent\_node);

printf("\n cascading cut called");

cascading\_cut(H, parent\_node);

}

if (node\_to\_be\_decrease->key < H->min->key) {

H->min = node\_to\_be\_decrease;

}

}

}

}

void \*find\_node(FIB\_HEAP \*H, NODE \*n, int key, int new\_key) {

NODE \*find\_use = n;

NODE \*f = NULL;

find\_use->visited = true;

if (find\_use->key == key) {

find\_use->visited = false;

f = find\_use;

decrease\_key(H, f, new\_key);

}

if (find\_use->child != NULL) {

find\_node(H, find\_use->child, key, new\_key);

}

if ((find\_use->right\_sibling->visited != true)) {

find\_node(H, find\_use->right\_sibling, key, new\_key);

}

find\_use->visited = false;

}

FIB\_HEAP \*insertion\_procedure() {

FIB\_HEAP \*temp;

int no\_of\_nodes, ele, i;

NODE \*new\_node;

temp = (FIB\_HEAP \*)malloc(sizeof(FIB\_HEAP));

temp = NULL;

if (temp == NULL) {

temp = make\_fib\_heap();

}

printf(" \n enter number of nodes to be insert = ");

scanf("%d", &no\_of\_nodes);

for (i = 1; i <= no\_of\_nodes; i++) {

printf("\n node %d and its key value = ", i);

scanf("%d", &ele);

insertion(temp, new\_node, ele);

}

return temp;

}

void Delete\_Node(FIB\_HEAP \*H, int dec\_key) {

NODE \*p = NULL;

find\_node(H, H->min, dec\_key, -5000);

p = extract\_min(H);

if (p != NULL)

printf("\n Node deleted");

else

printf("\n Node not deleted:some error");

}

int main(int argc, char \*\*argv) {

NODE \*new\_node, \*min\_node, \*extracted\_min, \*node\_to\_be\_decrease, \*find\_use;

FIB\_HEAP \*heap, \*h1, \*h2;

int operation\_no, new\_key, dec\_key, ele, i, no\_of\_nodes;

heap = (FIB\_HEAP \*)malloc(sizeof(FIB\_HEAP));

heap = NULL;

while (1) {

printf(" \n Operations \n 1. Create Fibonacci heap \n 2. Insert nodes into fibonacci heap \n 3. Find min \n 4. Union \n 5. Extract min \n 6. Decrease key \n 7.Delete node \n 8. print heap \n 9. exit \n enter operation\_no = ");

scanf("%d", &operation\_no);

switch (operation\_no) {

case 1:

heap = make\_fib\_heap();

break;

case 2:

if (heap == NULL) {

heap = make\_fib\_heap();

}

printf(" enter number of nodes to be insert = ");

scanf("%d", &no\_of\_nodes);

for (i = 1; i <= no\_of\_nodes; i++) {

printf("\n node %d and its key value = ", i);

scanf("%d", &ele);

insertion(heap, new\_node, ele);

}

break;

case 3:

min\_node = find\_min\_node(heap);

if (min\_node == NULL)

printf("No minimum value");

else

printf("\n min value = %d", min\_node->key);

break;

case 4:

if (heap == NULL) {

printf("\n no FIbonacci heap created \n ");

break;

}

h1 = insertion\_procedure();

heap = unionHeap(heap, h1);

printf("Unified Heap:\n");

print\_heap(heap->min);

break;

case 5:

if (heap == NULL)

printf("Empty Fibonacci heap");

else {

extracted\_min = extract\_min(heap);

printf("\n min value = %d", extracted\_min->key);

printf("\n Updated heap: \n");

print\_heap(heap->min);

}

break;

case 6:

if (heap == NULL)

printf("Fibonacci heap is empty");

else {

printf(" \n node to be decreased = ");

scanf("%d", &dec\_key);

printf(" \n enter the new key = ");

scanf("%d", &new\_key);

find\_use = heap->min;

find\_node(heap, find\_use, dec\_key, new\_key);

printf("\n Key decreased- Corresponding heap:\n");

print\_heap(heap->min);

}

break;

case 7:

if (heap == NULL)

printf("Fibonacci heap is empty");

else {

printf(" \n Enter node key to be deleted = ");

scanf("%d", &dec\_key);

Delete\_Node(heap, dec\_key);

printf("\n Node Deleted- Corresponding heap:\n");

print\_heap(heap->min);

break;

}

case 8:

print\_heap(heap->min);

break;

case 9:

free(new\_node);

free(heap);

exit(0);

default:

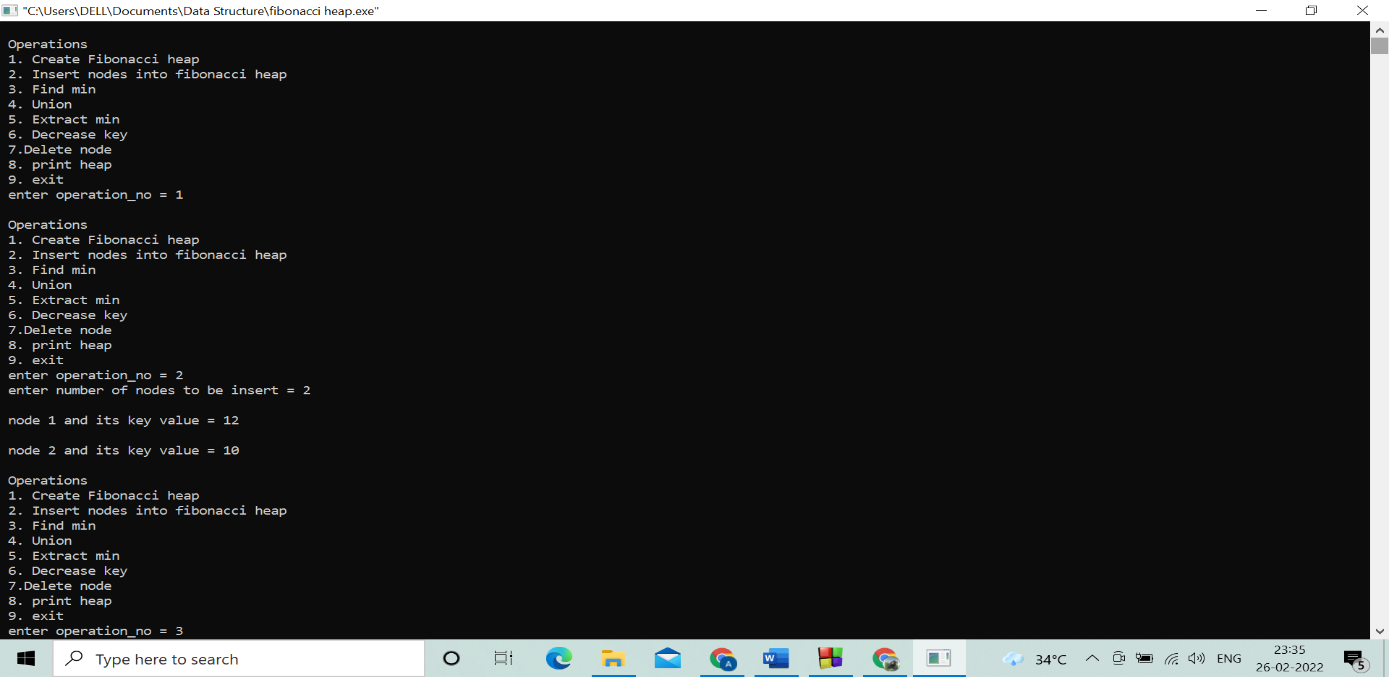
printf("Invalid choice ");

}

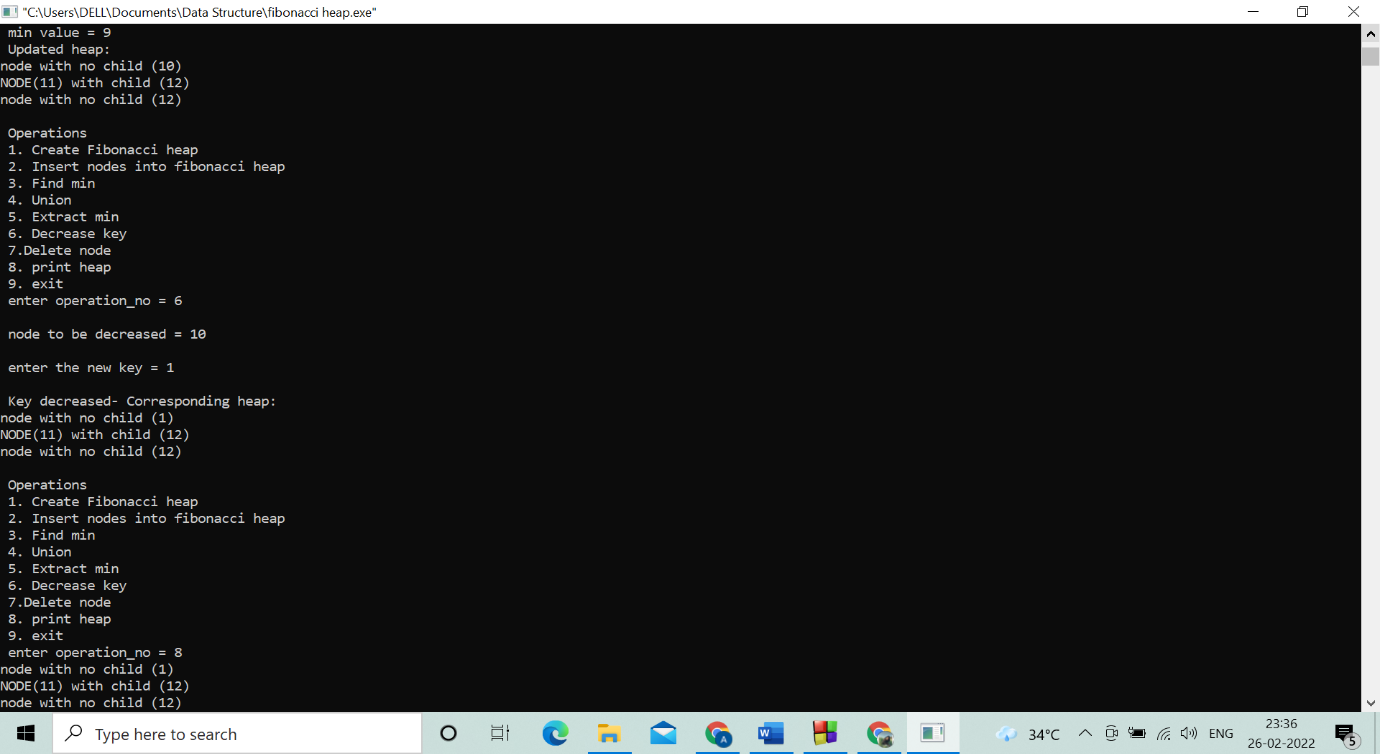
}

}

Output:







**Program 28. Implementation of Prim’s algorithm**

//prims algorithm

#include <limits.h>

#include <stdbool.h>

#include <stdio.h>

#define V 5

int minKey(int key[], bool mstSet[])

{

int min = INT\_MAX, min\_index;

for (int v = 0; v < V; v++)

if (mstSet[v] == false && key[v] < min)

min = key[v], min\_index = v;

return min\_index;

}

int printMST(int parent[], int graph[V][V])

{

printf("Edge \tWeight\n");

for (int i = 1; i < V; i++)

printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);

}

void primMST(int graph[V][V])

{

int parent[V];

int key[V];

bool mstSet[V];

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

key[i] = INT\_MAX, mstSet[i] = false;

key[0] = 0;

parent[0] = -1;

for (int count = 0; count < V - 1; count++) {

int u = minKey(key, mstSet);

mstSet[u] = true;

for (int v = 0; v < V; v++)

if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])

parent[v] = u, key[v] = graph[u][v];

}

printMST(parent, graph);

}

int main()

{

int graph[V][V] = { { 0, 2, 0, 6, 0 },

{ 2, 0, 3, 8, 5 },

{ 0, 3, 0, 0, 7 },

{ 6, 8, 0, 0, 9 },

{ 0, 5, 7, 9, 0 } };

primMST(graph);

return 0;

}

Output:

