|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl no.** | **Topic** | **Date** | **Page no.** | **Teacher’s Signature** |
| 1. | Write a program in C to implement base conversion from  any base to any other base (among decimal, binary, octal and hexadecimal). | 26-03-2022 | 3 - 9 |  |
| 2. | Write a program in C to search an element from a list. Give option to user to perform Linear or Binary Search | 15-05-2022 | 10-16 |  |
| 3. | Write a program in C to implement Bubble Sort. | 17-05-2022 | 17-20 |  |
| 4. | Write a program in C to implement Insertion Sort. | 17-05-2022 | 21-24 |  |
| 5. | Write a program in C to implement Selection Sort. | 17-05-2022 | 25-29 |  |
| 6. | Write a program in C to implement Quick Sort. | 29-06-2022 | 30-34 |  |
| 7. | Write a program in C to implement Merge Sort. | 28-06-2022 | 35-38 |  |
| 8. | Write a program in C to transpose a matrix. | 22-04-2022 | 39-42 |  |
| 9. | Write a program in C to implement Diagonal Matrix using one-dimensional array. | 01-06-2022 | 43-47 |  |
| 10. | Write a program in C to implement Lower and Upper  Triangular Matrix using one-dimensional array. | 02-06-2022 | 48-56 |  |
| 11. | Write a program in C to implement Symmetric Matrix using one-dimensional array. | 31-05-2022 | 57-60 |  |
| 12. | Write a program in C to implement Sparse Matrix using one- dimensional array. | 04-06-2022 | 61-65 |  |
| 13. | Write a program in C to implement matrix multiplication. | 04-06-2022 | 66-69 |  |
| 14. | Write a program in C to create a 2D array dynamically. | 30-06-2022 | 70-72 |  |
| 15. | Write a program in C to implement Stack using array. | 03-04-2022 | 73-76 |  |
| 16. | Write a program in C to implement Linear Queue using array. | 10-04-2022 | 77-80 |  |
| 17. | Write a program in C to implement Circular Queue using  array. | 25-04-2022 | 81-86 |  |
| 18. | Write a program in C to scan polynomial using array. Implement addition, subtraction, multiplication of two polynomials. | 10-07-2022 | 87-95 |  |
| 19. | Write a program in C to implement evaluation of postfix notations. | 30-06-2022 | 96-100 |  |
| 20. | Write a program in C to implement Singly Linked List. | 05-06-2022 | 101-113 |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl no.** | **Topic** | **Date** | **Page no.** | **Teacher’s Signature** |
| 21. | Write a program in C to implement Doubly Linked List. | 09-06-2022 | 114-128 |  |
| 22. | Write a program in C to merge two sorted list into a Single List. | 19-06-2022 | 129-133 |  |
| 23. | Write a program in C to implement a Binary Search Tree and including following operations in tree:   1. Insertion (Recursive Implementation). 2. Search a no. in BST. 3. Display it preorder, postorder, inorder traversals recursively. | 17-07-2022 | 134-140 |  |

**ASSIGNMENT – 1**

**1.PROBLEM STATEMENT**

Write a program in C to implement base conversion from any base to any other base (among binary,decimal,octal and hexadecimal).

**2.ALGORITHMS**

Algorithm **Decimal\_To\_Other**

**Input:** A decimal number ‘num’ and the base ‘dbase’ in which it is to be converted.

**Output:** The decimal number ‘num’ converted to base ‘dbase’ and store in an array whose address is returned.

**Remarks:** The input decimal number must be valid.

**Steps:**

1. k=0
2. temp=num
3. **While**(temp≠0) **do**
4. Find the value of temp modulo dbase and store it in ‘rem’
5. Store rem in an array of integers ‘result’ at k
6. Divide temp by 10
7. Increment k by 1
8. **EndWhile**
9. **Return** result

Algorithm **Other\_To\_Decimal**

**Input:** A number ‘num’ and it’s base ‘sbase’.

**Output:** The number ‘num’ converted to a decimal number and returned.

**Remarks:** The input number must be valid.

**Steps:**

1. **Reverse**(num) //reverse the number
2. temp=num
3. sum=0
4. i=0
5. **While**(temp≠0) **do**
6. digit = temp modulo 10 //obtain last digit
7. sum=sum+(digit \* sbasei) //find value
8. temp = temp/10 //eliminate last digit from temp
9. i=i+1
10. **EndWhile**
11. **Return** sum
12. **Stop**

Algorithm **Reverse**

**Input:** The address of the integer array ‘arr’ to be reversed and the number of elements ‘len’.

**Output:** The contents of ‘arr’ reversed.

**Remarks:** The array must be passed as pointer.

**Steps:**

1. **For**(i=1 to len) **do**
2. **arr[i]=arr[i]+arr[j]**
3. **arr[j]=arr[i]-arr[j]**
4. **arr[i]=arr[i]+arr[j]**
5. **EndFor**
6. **Stop**

Algorithm **Other\_To\_Other**

**Input:** A number ‘num’ with its base ‘sbase’ and the base in which it is to be converted ‘dbase’.

**Output:** The number ‘num’ converted to base ‘dbase’ and returned.

**Remarks:** The input number must be valid.

**Steps**

1. res1=**Other\_To\_Decimal**(num,sbase) //find decimal equivalent of num
2. **res2=Decimal\_To\_Others**(res1,dbase) //convert res1 to required base.
3. **Return** res1
4. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<string.h>

#include<math.h>

#include<stdlib.h>

//function to reverse an array

void reverse(int \*arr,int len)

{

    int i,j;

    for(i=0,j=len-1;i<j;i++,j--)

    {

        arr[i]=arr[i]+arr[j];

        arr[j]=arr[i]-arr[j];

        arr[i]=arr[i]-arr[j];

    }

}

//function to convert a number from a given base to decimal

int othertodec(char \*num,int sbase)

{

    int i,x,sum=0;

    strrev(num);

    //Mapping Each Element Of String

    for(i=0;num[i]!='\0';i++)

    {

        if(sbase==16 && num[i]>=65&&num[i]<=70)

            num[i]=num[i]-7; // Joining the number and character sequences together

        num[i]=num[i]-'0';

        sum = sum + num[i]\*pow(sbase,i);

    }

    return sum;

}

//function to convert a decimal number into required base

void dectoother(char \*num,int dbase)

{

    int remarr[20];

    int i,j,k,val=0,tempval,len,rem;

    sscanf(num,"%d",&val); //extract integer from string

    tempval=val;

    i=0;

    while(tempval!=0)

    {

        remarr[i]=tempval%dbase;

        if(remarr[i]<10) //if remainder is less than 10

            remarr[i]+'0'; //add ASCII value of 0

        else if(remarr[i]>9) //if remainder is greater than 9

            remarr[i]+'A'; //add ASCII value of A

        tempval=tempval/dbase;

        i++;

    }

    reverse(remarr,i);

    for(j=0;j<i;j++) //display the result char by char

    {

        if(remarr[j]>9)

            printf("%c",remarr[j]+55);

        else

            printf("%d",remarr[j]);

    }

}

//function to convert a number from one base to another base

void othertoother(char\*num,int sbase,int dbase)

{

    int res1,temp,i=0,j;

    char arr[20];

    res1=othertodec(num,sbase); //find decimal equivalent of num

    temp=res1;

    //converting the decimal equivalent to a string

    while(temp!=0)

    {

        arr[i]=temp%10+'0';

        temp = temp/10;

        i++;

    }

    arr[i]='\0';

    strrev(arr);

    dectoother(arr,dbase);

}

int main(void)

{

    int sbase,dbase,res1;

    char num[20];

printf(“For base conversion of a number: \n);

    printf("Enter Input Base: ");

    scanf("%d",&sbase);

    printf("Enter Output Base: ");

    scanf("%d",&dbase);

    printf("Enter the number: ");

    fflush(stdin); //empty input buffer

    gets(num);

    if(dbase==10)

    {    res1=othertodec(num,sbase);

        printf("DECIMAL:%d",res1);

    }

    else if(sbase==10)

        dectoother(num,dbase);

    else if(sbase!=10 && dbase!=10)

    {

        othertoother(num,sbase,dbase);

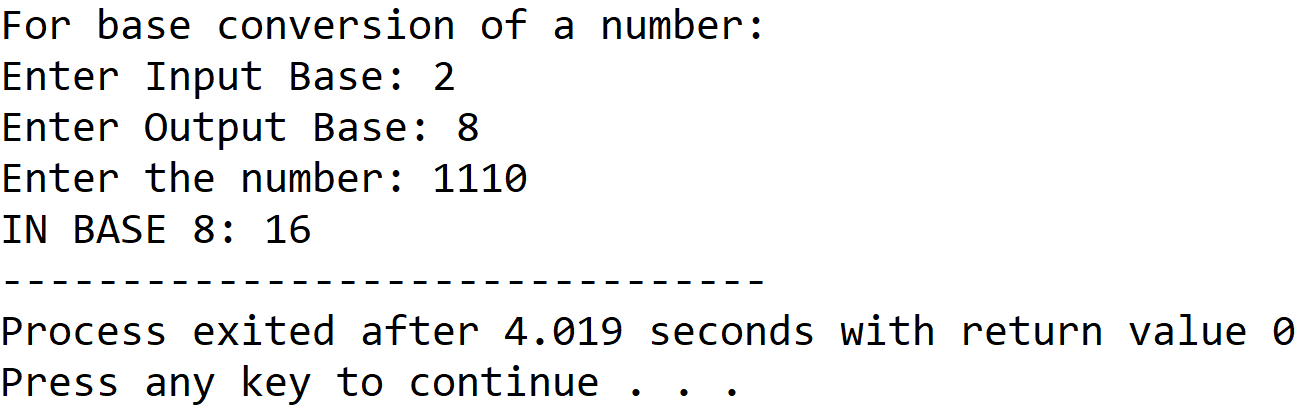
    }

    return 0;

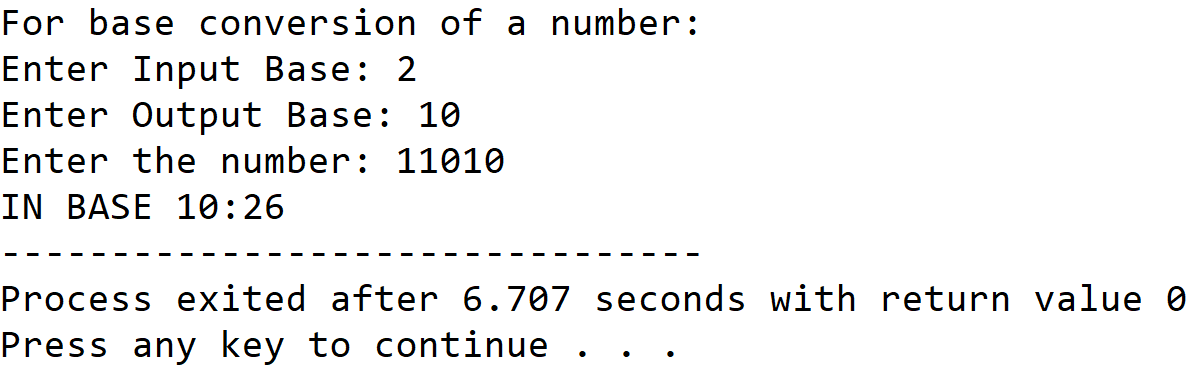
}

**4.OUTPUT**

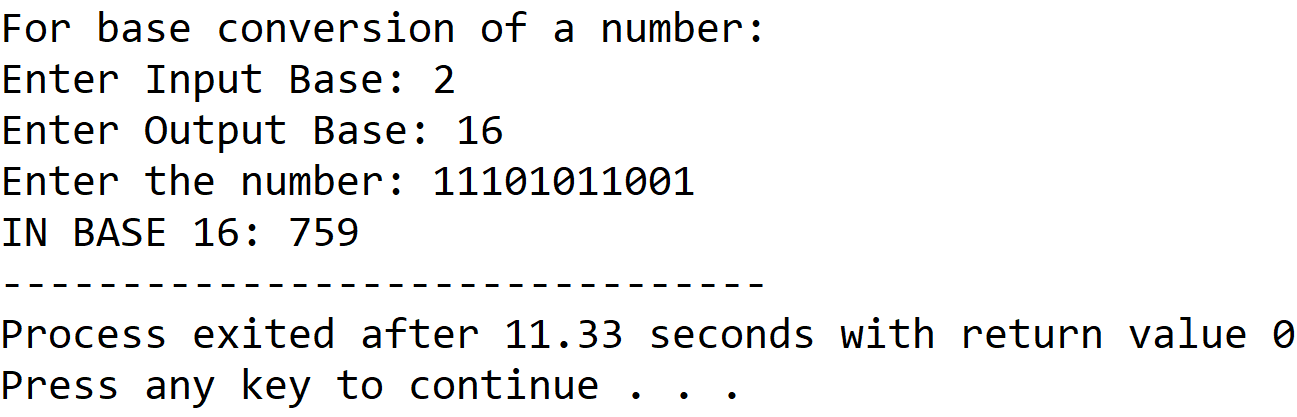
**SET 1:** Binary to octal



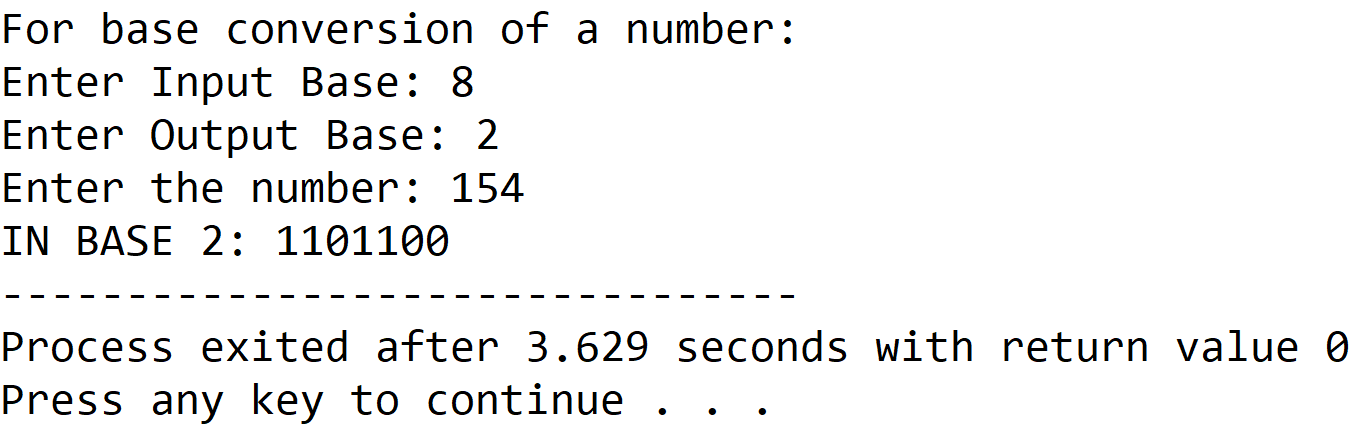
**SET 2:** Binary to decimal



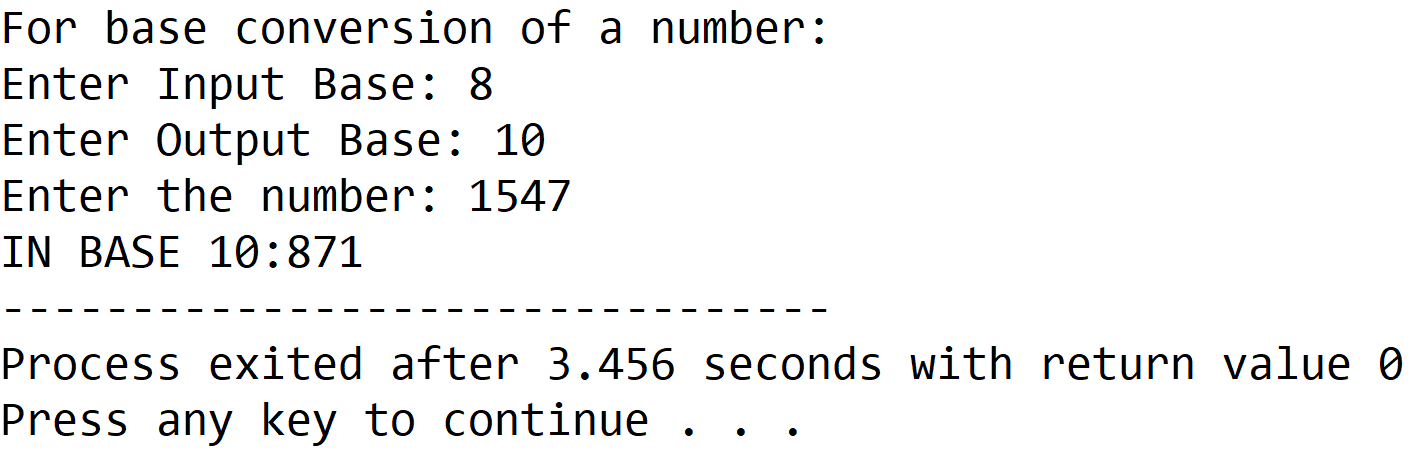
**SET 3:** Binary to hexadecimal



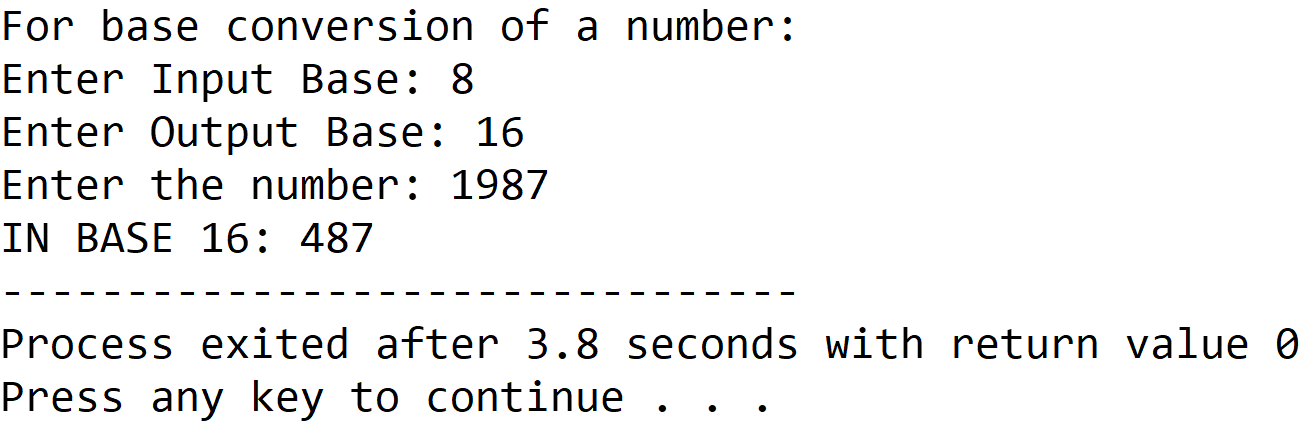
**SET 4:** Octal to binary



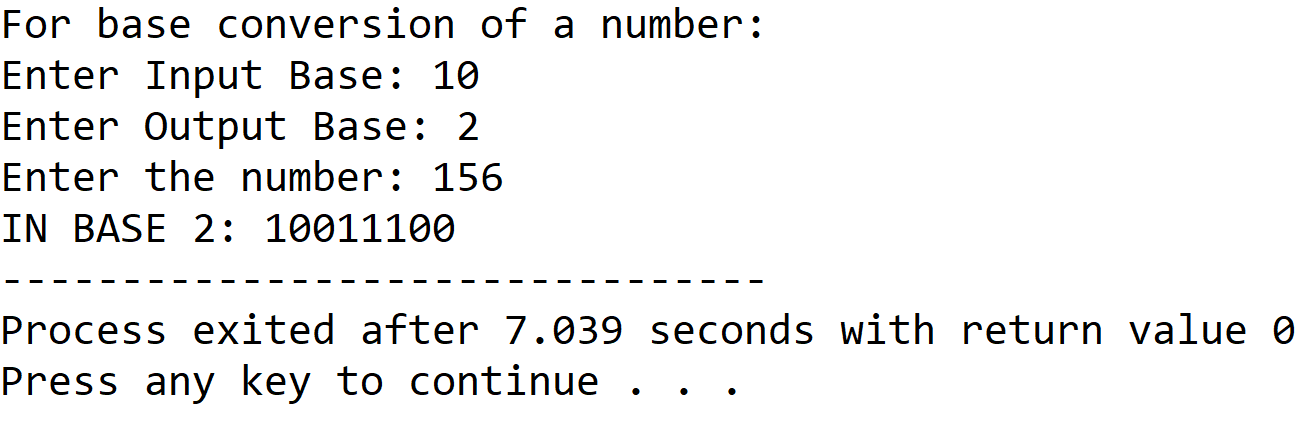
**SET 5:** Octal to decimal



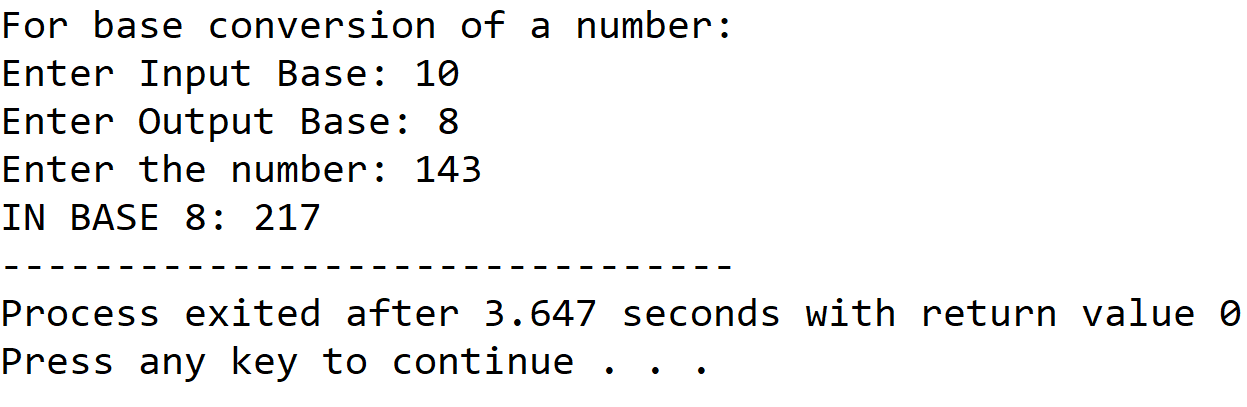
**SET 6:** Octal to hexadecimal



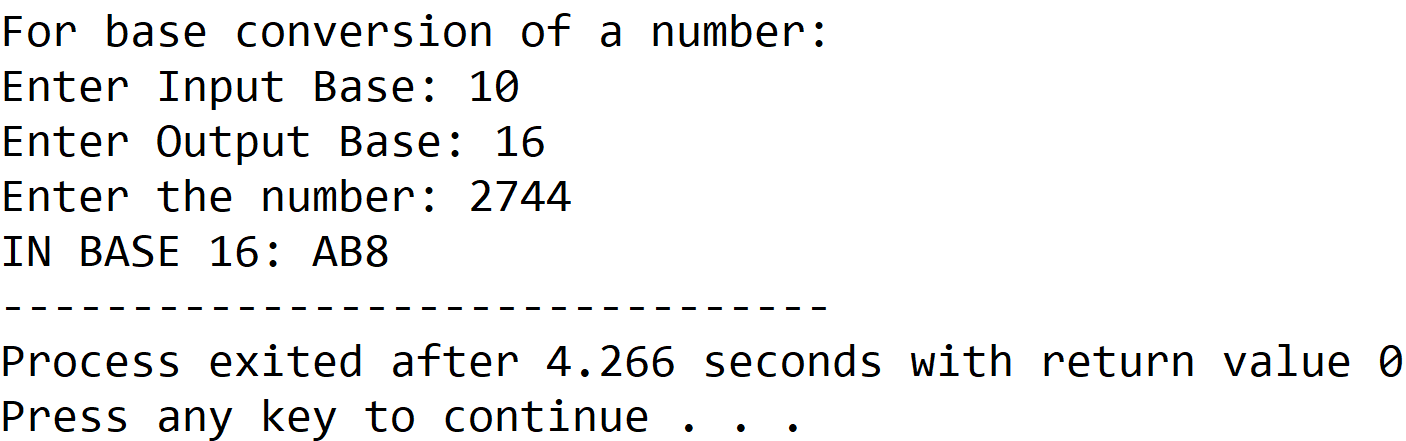
**SET 7:** Decimal to binary



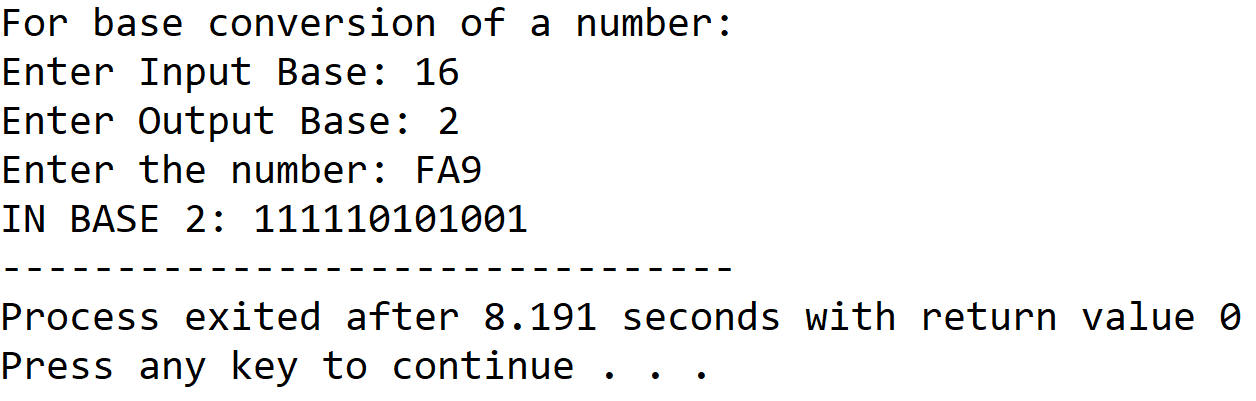
**SET 8:** Decimal to octal



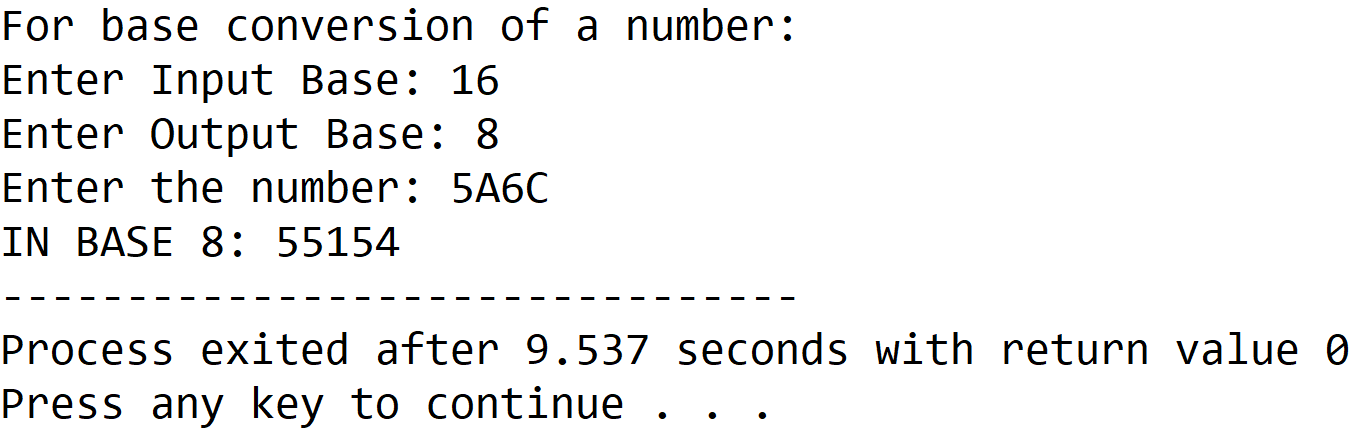
**SET 9:** Decimal to hexadecimal



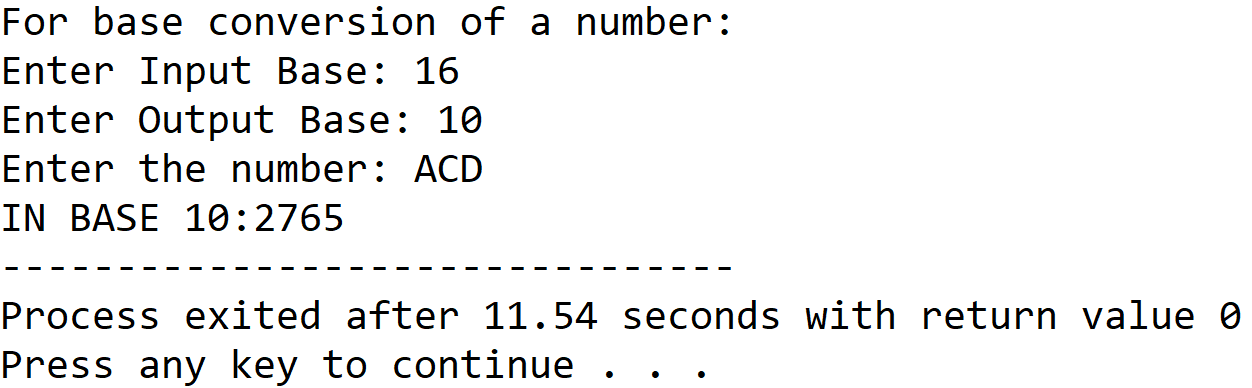
**SET 10:** Hexadecimal to binary



**SET 11:** Hexadecimal to octal



**SET 12:** Hexadecimal to decimal



**5.DISCUSSIONS**

**Variable Description**

* **Sbase:** input base
* **Dbase:** output base
* **Num:** string to hold the input number
* **Remarr:** integer array to hold remainders in dectoother procedure
* **I,j,k:** loop counters

**Limitations**

* There is no procedure to check the user input, so the input must be valid otherwise the program will produce undesired output.

**Uses**

* The program can be used to convert any number among the bases: binary,decimal,octal and hexadecimal. It can find implementation in a calculator program.

**Future Scope**

* A provision for input validation may be added.
* The program can be modified to support more bases.

**Teacher’s Signature**

**ASSIGNMENT - 2**

**1.PROBLEM STATEMENT**

Write a program in C to search an element from a list. Give option to the user to perform linear search or binary search

**2.ALGORITHMS**

The program presents an option to the user to perform either linear search or binary search.

**Algorithm Linear\_Search**

**Input:** A pointer to the array named arr[1…n] with size n containing the list of integers and the integer to be searched is called item.

**Output:** If the given integer is found , the index of the position of the element is returned with a successful message , else an unsuccessful message is shown.

**Remarks:** It is assumed that the array is not empty.

**Steps:**

1. NOTFOUND=-1 *// Invalid index to indicate unsuccessful search*
2. flag = 0 *//* *To indicate whether element was found or not*
3. **For** i=1 to n **do** *//**Traversing the whole list*
4. **If**(arr[i]=item) **then** *//**if value at index equals item*
5. flag=1 *// change value of flag to 1*
6. return i *// index of the element is returned*
7. **EndIf**
8. **If**(flag=0) **then** *// if flag value is not changed*
9. return NOTFOUND*//**return invalid index to indicate absent item*
10. **EndIf**
11. **Stop**

**Algorithm Binary\_Search**

**Input:** A pointer to the integer array named arr[1...n] with size n containing the list of integers and the integer to be searched is called item.

**Output:** If the given integer is found , the index of the position of the element is returned with a successful message , else an unsuccessful message is shown.

**Remarks:** It is assumed that the array is not empty and is sorted in ascending order.

**Steps:**

1. low=1 , high=n
2. flag=FALSE
3. **While**(low≤high) **do**
4. mid = (low+high)/2
5. **If**(arr[mid]=item) **then**
6. flag=TRUE
7. return mid
8. **EndIf**
9. **If**(arr[mid]<item) **then**
10. low=mid+1
11. **EndIf**
12. **If**(arr[mid]>item) **then**
13. high=mid-1
14. **EndIf**
15. **EndWhile**
16. **If**(flag=FALSE) **then**
17. return NOTFOUND
18. **EndIf**
19. **Stop**

**3.SOURCE CODE**

​#​include​<​stdio.h​>

​#​include​<​stdlib.h​>

​#​define​ ​NOTFOUND​ -​1

// Function to ask the user whether to continue or to exit

​int​ ​prompt​(​void​)

​{

​        ​int​ num;

​        ​printf​(​"​\n​Press:​\n​0 To Exit​\n​1 to continue​\n​"​);

​        ​scanf​(​"​%d​"​,&num);

​        ​return​ num;

​}

// Function to take input in an array from the user

​void​ ​getarr​(​int​\*arr,​int​ num)

​{

​        ​int​ i;

​        ​for​(i=​0​;i<num;i++)

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

​}

// sorting algorithm for binarysearch function

void insertionsort(int\* arr,int num)

{

int i,j,temp;

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

{

temp=arr[i];

for(j=i-1; j>=0 && arr[j]>temp;j--)

arr[j+1]=arr[j];

arr[j+1]=temp;

}

}

// Function to display the contents of an array

​void​ ​disparr​(​int​\*arr,​int​ num)

​{

​        ​int​ i;

​        ​for​(i=​0​;i<num;i++)

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

​}

​int​ ​binarysearch​(​int​\*arr,​int​ num,​int​ item)

​{

​        ​int​ low=​0​,high=num-​1​,mid; // setting initial lower and upper index variables

insertionsort(arr,num); // sorting the list to enable binarysearch

​        ​while​(low<=high)

​        {

​                mid=(low+high)/​2​; // calculating the middle element

​                ​if​(arr[mid]==item) // if mid element equals item

​                        ​return​ mid; // return the middle index

​                ​else​ ​if​(item<arr[mid]) //if mid element is greater than item

​                        high=mid-​1​; // set the value of high than mid-1

​                ​else​ ​if​(item>arr[mid]) // if mid element is less than item

​                        low=mid+​1​; // set the value of low to mid+1

​        }

​        ​return​ NOTFOUND; // if the above does not return,item is not found

​}

​int​ ​linearsearch​(​int​\*arr,​int​ num,​int​ item)

​{

​        ​int​ i;

​        ​for​(i=​0​;i<num;i++) //traversing the array

​                ​if​(arr[i]==item) // if an element equals item

​                        ​return​ i; // return the corresponding index

​        ​return​ NOTFOUND;// if the above does not return , item is not found

​}

​int​ ​main​(​void​)

​{

​        ​int​ num,i,ch,item,found,val;

​        ​int​\*arr; // pointer to hold an array

​while​(​1​)

​        {

printf(“To search a given list of integers: \n“);

​        ​printf​(​"​Enter the number of elements needed: ​"​);

​        ​scanf​(​"​%d​"​,&num);

​        ​if​(num<​2​) // Checking if at least two elements are present

​        {

​                ​printf​(​"​Invalid Array Length​\n​Please Retry​"​);

​                ​return​ ​0​;

​        }

​        arr=(​int​\*)​calloc​(num,​sizeof​(​int​)); // creating the array

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

​             ​getarr​(arr,num);

​

​                ​printf​(​"​\n​Enter the element to be searched: ​"​);

​                ​scanf​(​"​%d​"​,&item);

​

​                ​printf​(​"​\n\n​1.Binary Search\n​2.Linear Search​"​);

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

​                ​scanf​(​"​%d​"​,&ch);

​                ​switch​(ch)

​                {

​                        ​case​ ​1​:

​                        found=​binarysearch​(arr,num,item);

​                        if​(found==NOTFOUND)

​                        printf​(​"​\n​%d​ is not present in the list​"​,item);

​                        ​else

​                        ​printf​(​"​\n​%d​ Found At ​%d​"​,item,found);

​                        val=​prompt​(); //Asking user whether to continue

​                        if​(val==​0​)

​                        ​return​ ​0​; // Program terminates if user enters

​                        ​break​;

​                        ​case​ ​2​:

​                        found=​linearsearch​(arr,num,item);

​                        ​if​(found!=NOTFOUND)

​                        ​printf​(​"​\n​%d​ found at index ​%d​"​,item,found);

​                        ​else

​                              printf​(​"​\n​%d​ not found in the array​"​,item);

​                        val=​prompt​(); // Asking user whether to continue

​                        ​if​(val==​0​)

​                     ​return​ ​0​; // Program terminates if user enters 0

​                        ​break​;

​                        ​default​:

​              printf​(​"​Wrong Input-Please Enter Valid Choice​"​);

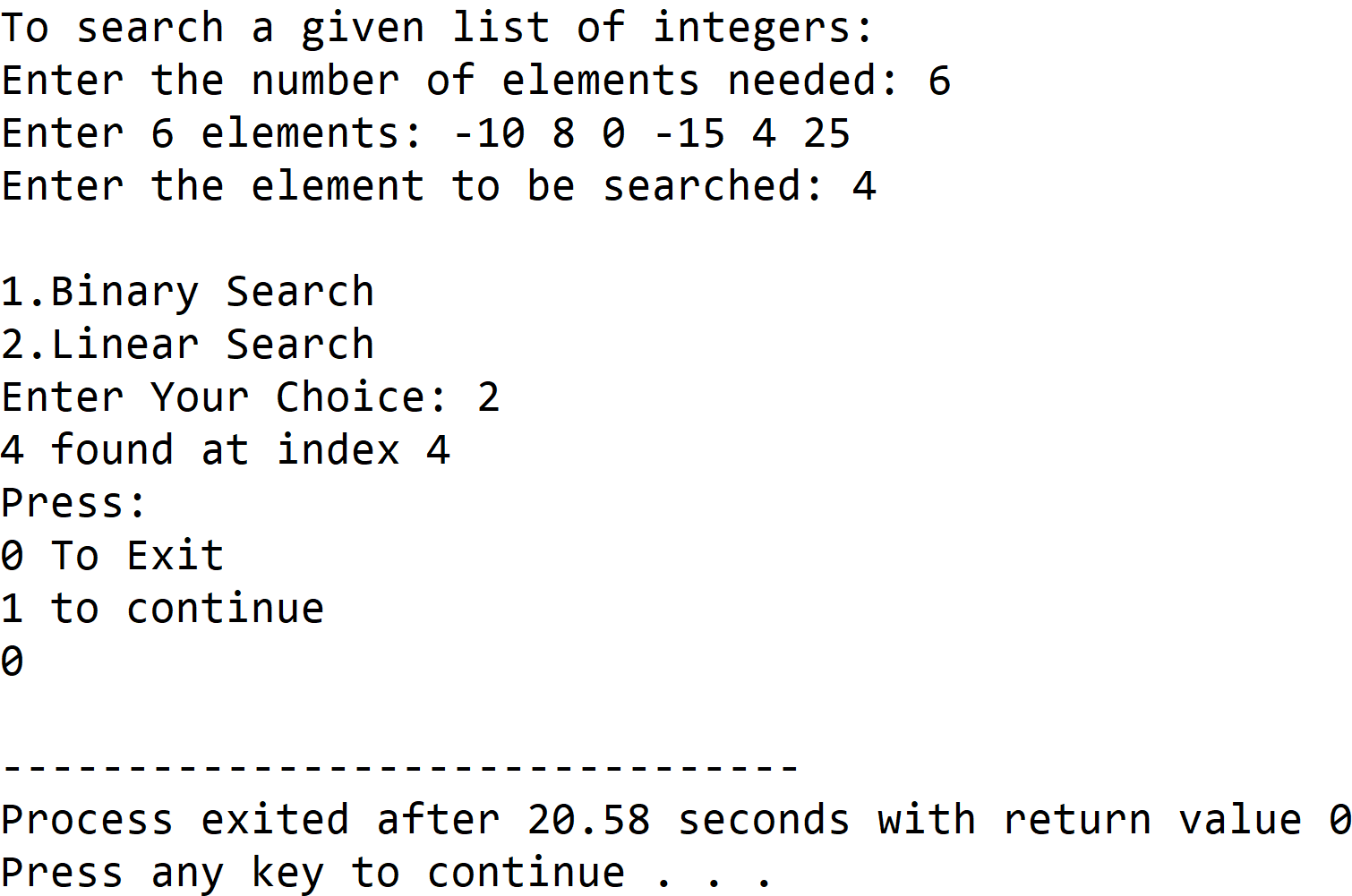
​                }

​        }

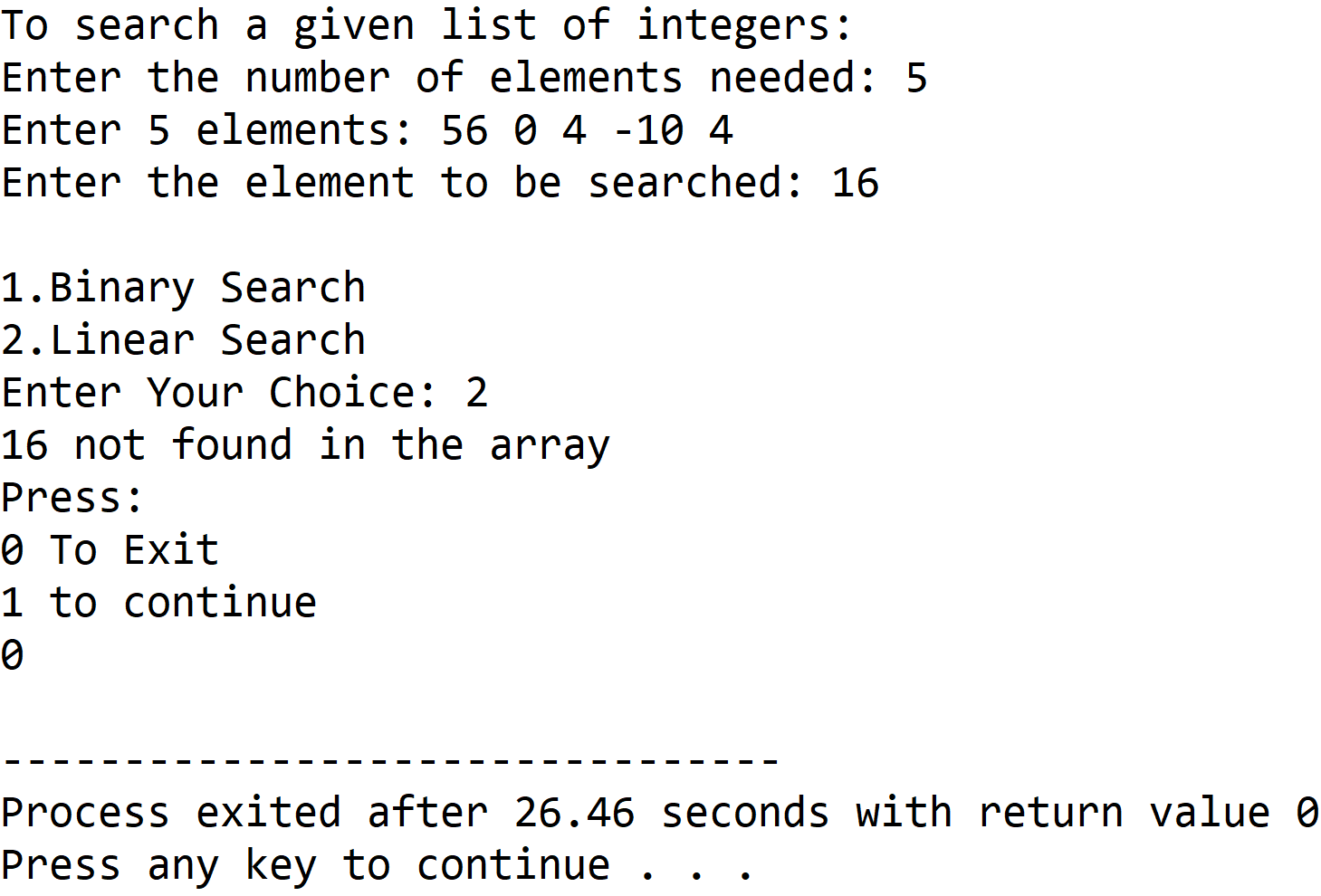
​}

**4. OUTPUT**

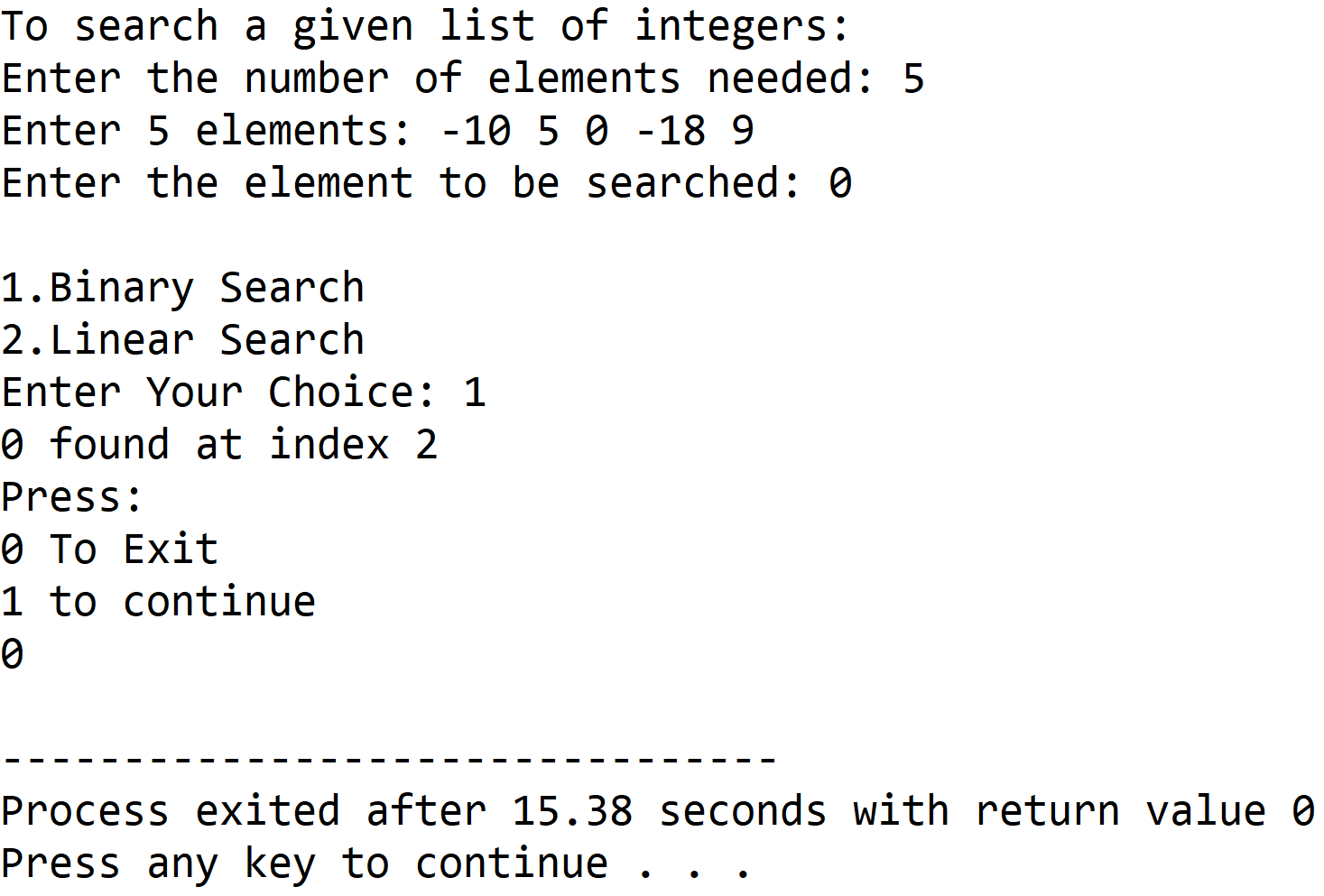
**Set 1:** Item Found in Linear Search



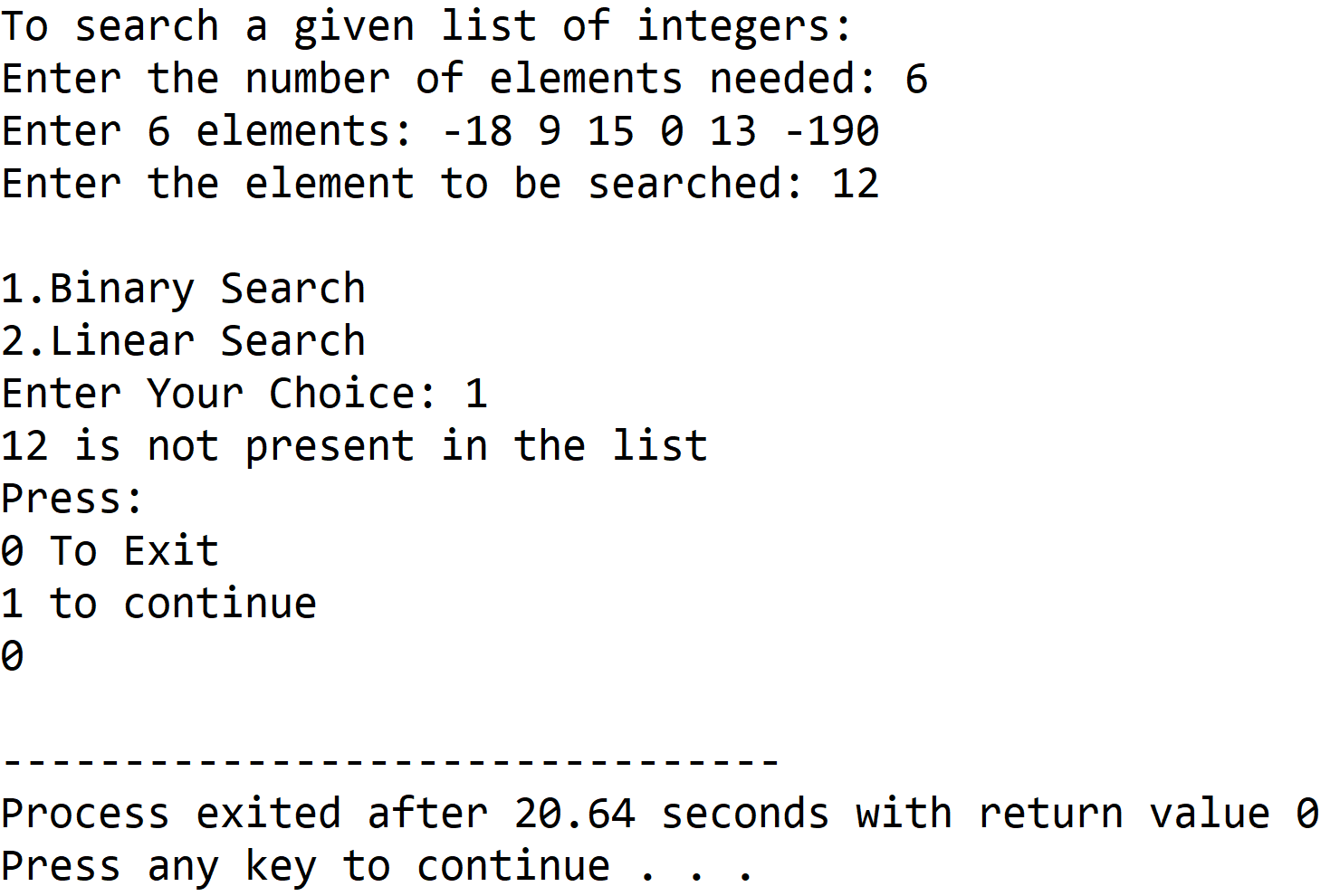
**Set 2:** Item Not Found in Linear Search



**Set 3:** Item Found in Binary Search



**Set 4:** Item Not Found in Binary Search



**5. DISCUSSIONS**

1. **Variable Description:**

**In Binary Search Function:**

* low : lower index value of the array under consideration
* high : upper index value of the array under consideration
* mid : index of the middle element with respect to low and high

**In Linear Search Function:**

* i : loop counter used to access the array

**In Main Function:**

* num : size of the array required by the user
* i : loop counter
* ch : holds the choice entered by the user in switch-case construct
* item: element to be found in the list
* val: stores return value of prompt function

**Macros Used:**

* NOTFOUND: holds a constant value -1 which is an invalid index value indicative of the absence of item in the given list

1. **Limitations:**

* The binarysearch function relies on having the array in sorted in ascending order, so the array entered by the user has to be sorted by using a sorting algorithm,in the above program insertion sort is used , if the data is unsorted , the binarysearch function will not work .
* The program uses an integer array to hold the list of integers entered by the user, since arrays are static data structures ,their size cannot be manipulated once it is allocated in the memory.

1. **Uses:**

* The above program can be used to search an integer value from any database consisting of a list of integers. For example, it can be used by educational institutions to enable automation of searching of roll numbers of students.

1. **Future Scope:**

* The list of integers can be stored in a linked list , enabling more elasticity in manipulation of size of the list.
* A sorting algorithm with better time and space complexity can to used for a more efficient program.

**Teacher’s Signature**

**ASSIGNMENT - 3**

**1.PROBLEM STATEMENT**

Write a program in C to implement bubble sort

**2.ALGORITHMS**

**Algorithm Bubble\_Sort**

**Input:**  A pointer to an integer array named arr[1…n] with size n

**Output:** The same input array in sorted order

**Remarks:** Elements are sorted in ascending order and it is assumed that the array is not empty

**Steps:**

1. **For** i=1 to (n-1) **do** // bubble sort needs n-1 iterations to complete
2. **For** j=1 to (n-i) **do** // traversing the unsorted array
3. **If**(arr[j]>arr[j+1] **then** // if previous is greater
4. **Swap**(arr[j],arr[j+1]) // swap the two elements
5. **EndIf**
6. **EndFor**
7. **Endfor**
8. **Stop**

**Algorithm Swap**

**Input:** The two variables named a and b whose data is to be swapped

**Output:** The two variables a and b with interchanged data

**Remarks:** The variables must be passed as pointers

**Steps:**

1. a = a + b
2. b = a - b
3. a = a - b
4. **Stop**

**3. Source Code**

#include<stdio.h>

#include<stdlib.h>

// function to swap the contents of two variables

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

{

\*a = \*a+\*b;

\*b = \*a-\*b;

\*a = \*a-\*b;

}

// function to display an array

void disparr(int\* arr,int num)

{

int i;

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

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

}

// function to take input in an array

void getarr(int \*arr, int num)

{

int i;

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

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

}

void bubblesort(int \*arr,int arrlen)

{

int i,j;

for(i=0;i<arrlen-1;i++)// loop to perform n-iterations

{

for(j=0;j<arrlen-1-i;j++) // access unsorted part

{

if(arr[j]>arr[j+1]) // if the order is wrong

swap(&arr[j],&arr[j+1]); //interchange the elements

}

printf("\n\nPASS %d: ",i+1);

disparr(arr,arrlen);

}

}

int main(void)

{

int \*arr,num,i;

printf("To Sort An Array");

printf("\nEnter The Length Of The Array: ");

scanf("%d",&num);

//checking if at least two elements are present

if(num<2)

{

printf("Invalid Array Length\nPlease Retry");

return 0;

}

arr = (int\*)calloc(num,sizeof(int));//creating the array in heap

printf("\nEnter %d Elements Of The Array: ",num);

getarr(arr,num); // taking input in array

printf("\nEntered Array: ");

disparr(arr,num); // displaying the entered array

bubblesort(arr,num);

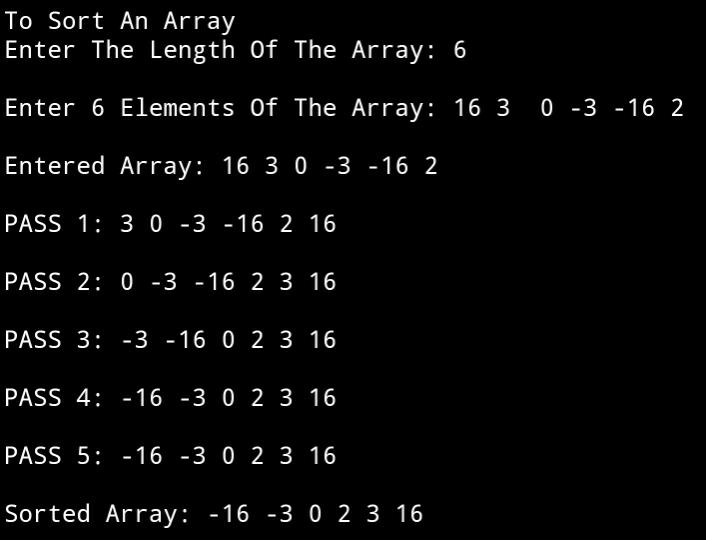
printf("\n\nSorted Array: ");

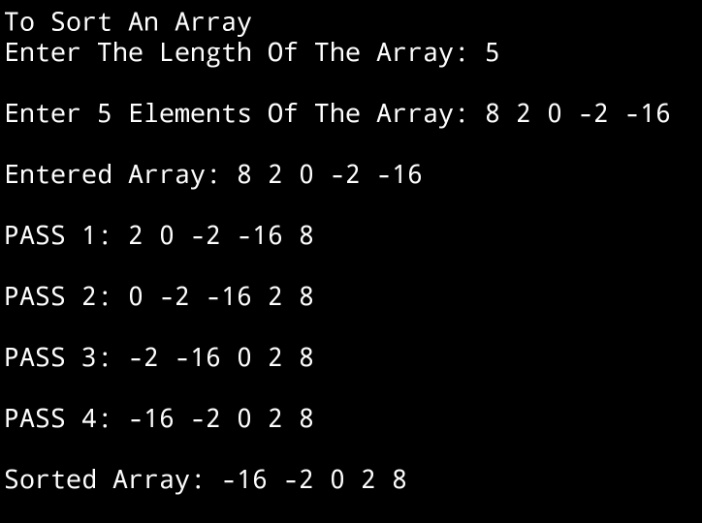
disparr(arr,num);

return 0;

}

**4. OUTPUT**

**SET 1:** Even Sized Input Array

**SET 2:** Odd Sized Input Array

1. **DISCUSSIONS**
2. **Variable Description**

**In main function**

* **num:** size of the array entered by the user
* **i:** loop counter to access the array
* **\*arr:** pointer to an array

**In bubblesort function**

* **i,j:** loop counters to access the array

**In disparr and getarr functions**

* **i**: loop counter

1. **Limitations**

* The time complexity for bubble sort algorithm is O(n2) , therefore, for a very large list of integers , the algorithm is less efficient
* The program uses an integer array to hold the list of integers entered by the user, since arrays are static data structures ,their size cannot be manipulated once it is allocated in the memory.

1. **Uses**

* The above program can be used to sort any list of integers in ascending order. It can be used by population survey groups to sort a list of people in ascending order of their age.

1. **Future Scope**

* The list of integers can be stored in a linked list , enabling more elasticity in manipulation of size of the list.

**Teacher’s Signature**

**ASSIGNMENT - 4**

**1. PROBLEM STATEMENT**

Write a program in C to implement insertion sort\

**2. ALGORITHMS**

**Algorithm Insertion\_Sort**

**Input:** A pointer to an integer array named arr[1…n] with size n

**Output:** The array arr in sorted order

**Remarks:** The array is sorted in ascending order

**Steps:**

1. **For**(j=2 to n) **do** // traversing the unsorted list
2. temp = arr[i] // storing the leftmost element of unsorted list
3. **For** j = i-1 to 1 **AND** arr[j]>temp **do** // finding valid position
4. arr[j+1] = arr[j] // shifting elements to the right
5. **EndFor**
6. arr[j+1] = temp // placing temp at valid position
7. **Stop**

**3. SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

// function to display an array

void disp(int\* arr,int num)

{

int i;

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

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

}

// function to take input in an array

void getarr(int\*arr,int num)

{

int i;

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

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

}

void insertionsort(int\* arr,int num)

{

int i,j,temp;

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

{

temp=arr[i]; // storing first element of unsorted list

for(j=i-1; j>=0 && arr[j]>temp;j--) // finding valid position

arr[j+1]=arr[j]; // shifting elements to the right

arr[j+1]=temp; // inserting temp at valid position

printf("\n\nPASS %d: ",i);

disp(arr,num);

}

}

int main(void)

{

int \*arr,num,i,j;

//taking the length of the array from the user

printf("Enter the number of elements needed: ");

scanf("%d",&num);

arr = (int\*)calloc(num,sizeof(int)); // creating array in heap

// checking if at least 2 elements are present

if(num<2)

{

printf("Invalid Array Length\nPlease Retry");

return 0;

}

//taking the elements of the array from user

printf("Enter %d elements of the Array: ",num);

getarr(arr,num);

//displaying the user entered array

printf("Entered Array: ");

disp(arr,num);

//calling insertion sort function

insertionsort(arr,num);

//displaying the sorted array

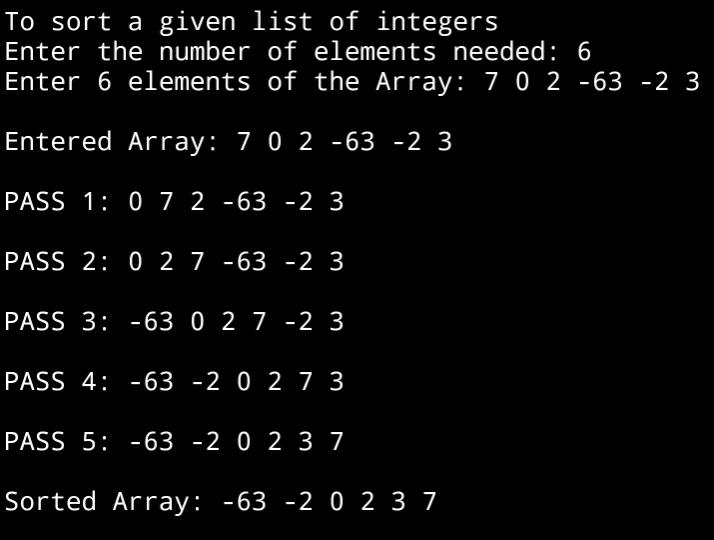
printf("\n\nSorted Array: ");

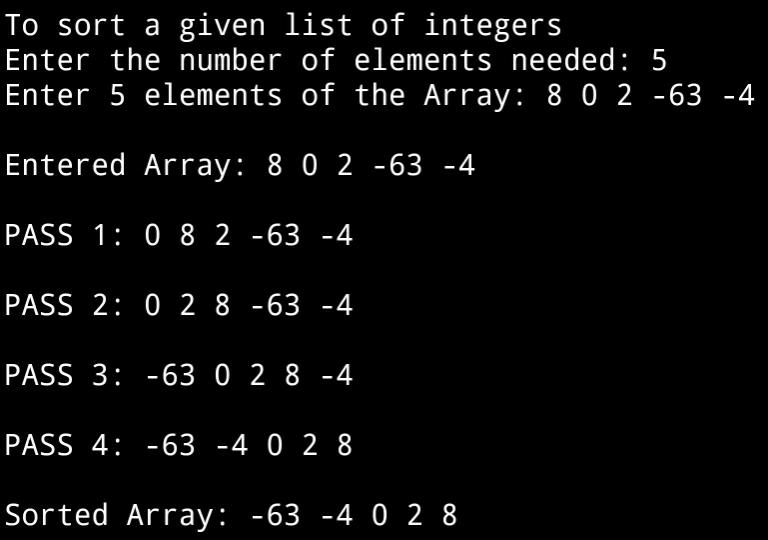
disp(arr,num);

return 0 ;

}

**4. OUTPUT**

**SET 1 :** Even Sized Input Array

**SET 2 :** Odd Sized Input Array

**5. DISCUSSIONS**

1. **Variable Description:**

**In insertionsort function**

* **i,j:** loop counters to access the array
* **temp:** to store the leftmost element of unsorted sublist

**In main function**

* **i,j:** loop counters to access the array
* **num:** size of the array entered by the user

**In getarr and disp function**

* **i:** loop counter

1. **Limitations**

* The program uses an integer array to hold the list of integers entered by the user, since arrays are static data structures ,their size cannot be manipulated once it is allocated in the memory.

1. **Uses**

* The above program can be used to sort any list of integers in ascending order. It can be used by educational institutions to arrange a list of students in ascending order of their roll numbers

1. **Future Scope**

* The list of integers can be stored in a linked list , enabling more elasticity in manipulation of size of the list.

**Teacher’s Signature**

**ASSIGNMENT - 5**

**1.PROBLEM STATEMENT**

Write a program in C to implement selection sort.

**2. ALGORITHMS**

**Algorithm Selection\_Sort**

**Input:** A pointer to an integer array name arr[1…n] of size n.

**Output:** The same array arr in sorted order.

**Remarks:** Elements are sorted in ascending order and it is assumed that the array is not empty.

**Steps:**

1. **For** i=1 to (n-1) **do** // performing (n-1) iterations
2. j=**Findmin**(i,n) // finding the minimum from the given range
3. **If**(i ≠ j) **then** // if minimum element is not the current element
4. **Swap**(arr[i] ,arr[j]) // exchange the two elements
5. **EndIf**
6. **EndFor**
7. **Stop**

**Algorithm Findmin**

**Input:** A pointer to an array named arr[1…n] of size n with left and right being the given range.

**Output:** The index of the smallest integer in the list.

**Remarks:** If more than one minimum elements are present , the index of the first occurrence is returned.

**Steps:**

1. minele = arr[left] // taking left element as smallest
2. minloc = left // holding the index of left element
3. **For** i= left+1 to right **do** // search the array
4. **If**(arr[i]<minele) **then** // if an element is smaller than minele
5. minele=arr[i] // update minele
6. minloc=I // update minloc
7. **EndIf**
8. **EndFor**
9. **Return** minloc // return the location of the smallest element
10. **Stop**

**Algorithm Swap**

**Input:** The two variables named a and b whose data is to be swapped.

**Output:** The two variables a and b with interchanged data.

**Remarks:** The variables must be passed as pointers.

**Steps:**

1. a = a + b // a holds sum of a and b
2. b = a – b // b holds the previous value of a
3. a = a – b // a holds the previous value of b
4. **Stop**

**3.Source Code**

#include<stdio.h>

#include<stdlib.h>

// function to interchange two elements

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

{

\*a=\*a+\*b;

\*b=\*a-\*b;

\*a=\*a-\*b;

}

// function to display an array

void disparr(int\*arr,int num)

{

int i;

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

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

}

//function to take input in an array

void getarr(int \*arr,int num)

{

int i;

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

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

}

// function to find minimum element in an array

int findmin(int \*arr,int start,int end)

{

int i,minloc;

minloc=start-1; // element at starting location taken as smallest

for(i=start;i<end;i++){ // traversing

if(arr[i]<arr[minloc]) // if element is smaller than assumed one

minloc=i; // update minimum element’s location

}

**return** minloc;

}

void selectionsort(int \*arr,int num)

{

int i,j,minloc;

for(i=0;i<num-1;i++)

{

j=i+1; //setting j at the start of unsorted list

minloc=findmin(arr,j,num); // finding the minimum element

if(i!=minloc) // if minimum element is not the current element

swap(&arr[i],&arr[minloc]); // swap the elements

printf("\n\nPASS %d: ",i+1);

disparr(arr,num);

printf("\n");

}

}

int main(void)

{

int \*arr,num,i;

printf("Enter The Number Of Elements Needed: ");

scanf("%d",&num);

arr = (int\*)calloc(num,sizeof(int)) //creating array in heap

// checking if atleast two elements are present

if(num<2)

{

printf("Invalid Array Length\nPlease Retry");

return 0;

}

printf("\nEnter %d Elements Of The Array: ",num);

getarr(arr,num);

selectionsort(arr,num);

printf("\n\nSorted Array: ");

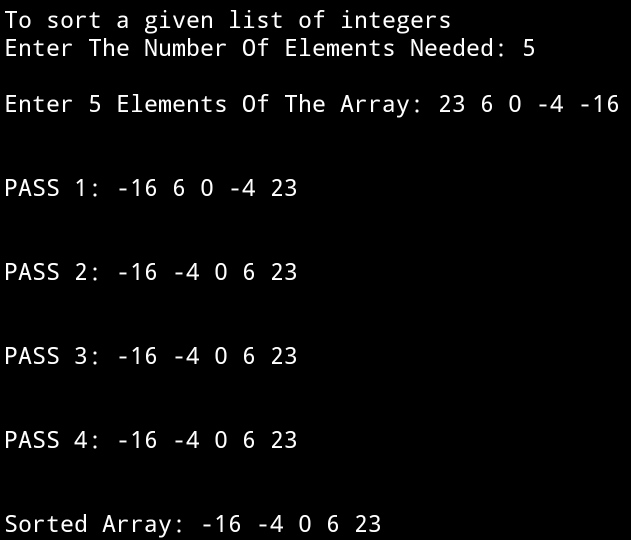
for(i=0;i<num;i++)

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

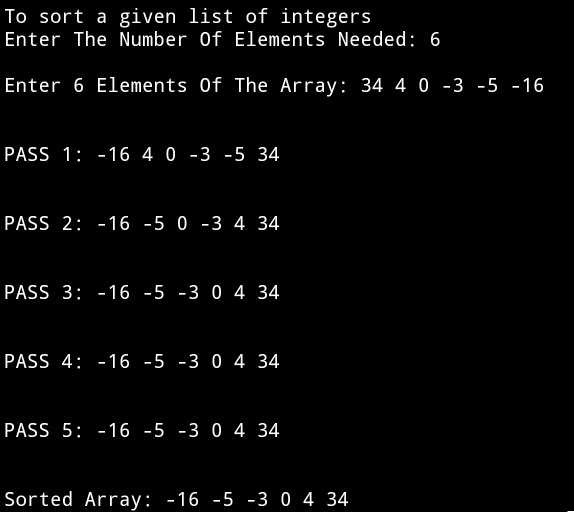
return 0;

}

**4.OUTPUT**

**SET 1: Odd Sized Input Array**

**SET 2: Even Sized Input Array**

****

**5.DISCUSSIONS**

1. **Variable Description**

**In main function**

* **\*arr:** pointer to the array.
* **num:** size of the array.
* **i:** loop counter.

**In selectionsort function**

* **i,j:** loop counter.
* **minloc:** holds the index of the minimum element in given range.

**In findmin function**

* **i:** loop counter.
* **minloc:** holds the index of the minimum element in given range.

**In getarr and disparr functions**

* **i:** loop counter.

1. **Limitations**

* The program uses an integer array to hold the list of integers entered by the user, since arrays are static data structures ,their size cannot be manipulated once it is allocated in the memory.

1. **Uses**

* The above program can be used to sort any list of integers in ascending order. It can be used to sort a list of employees working in an organization in ascending order of their wages.

1. **Future Scope**

* The list of integers can be stored in a linked list , enabling more elasticity in manipulation of size of the list.

**Teacher’s Signature**

**ASSIGNMENT - 6**

**PROBLEM STATEMENT**

Write a program in C to implement quick sort.

**ALGORITHMS**

Algorithm **Quick\_Sort**

**Input:** The pointer ‘arr’ to the array holding the list of integers, the left boundary index ‘left’ and and the right boundary index ‘right’.

**Output:** The array ‘arr’ with the integers sorted in ascending order.

**Remarks:** The algorithm works recursively.

**Steps:**

1. **If**(left<right) **then**
2. loc=**Partition**(arr,left,right) //get pivot element index
3. **Quick\_Sort**(arr,left,loc-1) //sort left subarray
4. **Quick\_Sort**(arr,loc+1,right) //sort right subarray
5. **EndIf**
6. **Stop**

Algorithm **Partition**

**Input:**The pointer ‘arr’ to the integer array, the left boundary index ‘left’ and the right boundary index ‘right’.

**Output:**The index of the pivot element placed in its sorted position.

**Steps:**

1. loc=left
2. **While**(left<right) **do**
3. **While**(arr[right]≥arr[loc] **AND** right>loc) **do**
4. right=right-1
5. **EndWhile**
6. **If**(arr[loc]>arr[right]) **then**
7. **Swap**(arr[loc],arr[right])
8. loc=right
9. left=left+1
10. **EndIf**
11. **While**(arr[left]≤arr[loc] **AND** left<loc) **do**
12. left=left+1
13. **EndWhile**
14. **If**(arr[loc]<arr[left]) **then**
15. **Swap**(arr[loc],arr[left])
16. loc=left
17. right=right-1
18. **EndIf**

**19. EndWhile**

**20.** **Return** loc

**21. Stop**

Algorithm **Swap**

**Input:** The two variables named ‘a’ and ‘b’ whose values are to be swapped.

**Output:** The values of variables ‘a’ and ‘b’ interchanged with one another.

**Remarks:** The variables must be passed as pointers.

**Steps:**

1. **a = a + b**
2. **b = a – b**
3. **a = a – b**
4. **Stop**

**3. SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to take input in an array

void getarr(int \*arr,int size)

{

int i;

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

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

}

//function to display an array

void disparr(int \*arr,int size)

{

int i;

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

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

}

//function to swap two variables

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

{

\*a=\*a+\*b;

\*b=\*a-\*b;

\*a=\*a-\*b;

}

//function to divide the array using pivot element

int partition(int \*arr,int left,int right)

{

int loc=left; //selecting left as pivot element

while(left<right) //traversing the whole array

{

//while right element is greater or equal to pivot element

while(arr[right]>=arr[loc] && loc<right)

right--;//decrement right

//if right element is less than pivot element

if(arr[right]<arr[loc])

{

swap(&arr[loc],&arr[right]);//swap right and pivot elements

loc=right;//set pivot location to right

left++;//left to right scan starts from left+1

}

//while left element is smaller/equal to pivot element

while(arr[left]<=arr[loc] && left<loc )

left++;//increment left

//if left element is greater than pivot element

if(arr[left]>arr[loc])

{

swap(&arr[left],&arr[loc]);//swap left and pivot element

loc=left;//set pivot location to left

right--;//right to left scan starts from right-1

}

}

return loc;

}

//recursive function for performing quicksort

void quicksort(int \*arr,int left,int right)

{

int loc;

if(left<right)//while there are more than one element

{

loc=partition(arr,left,right);//find pivot location

quicksort(arr,left,loc-1);//sort left subarray

quicksort(arr,loc+1,right);//sort right subarray

}

}

//function for input validation

void validate(int size)

{

if(size<2)

{

printf("The size must be atleast two");

exit(1);

}

}

int main(void)

{

int \*arr,size,left=0,right;

printf("To sort a list of integers using Quick Sort:\n");

printf("Enter the number of elements needed: ");

scanf("%d",&size);

validate(size);//validating input

right=size-1;

arr=(int\*)malloc(size\*sizeof(int));

printf("Enter %d elements of the array: ",size);

getarr(arr,size);

printf("Entered array: ");

disparr(arr,size);

quicksort(arr,left,right);

printf("\nSorted array: ");

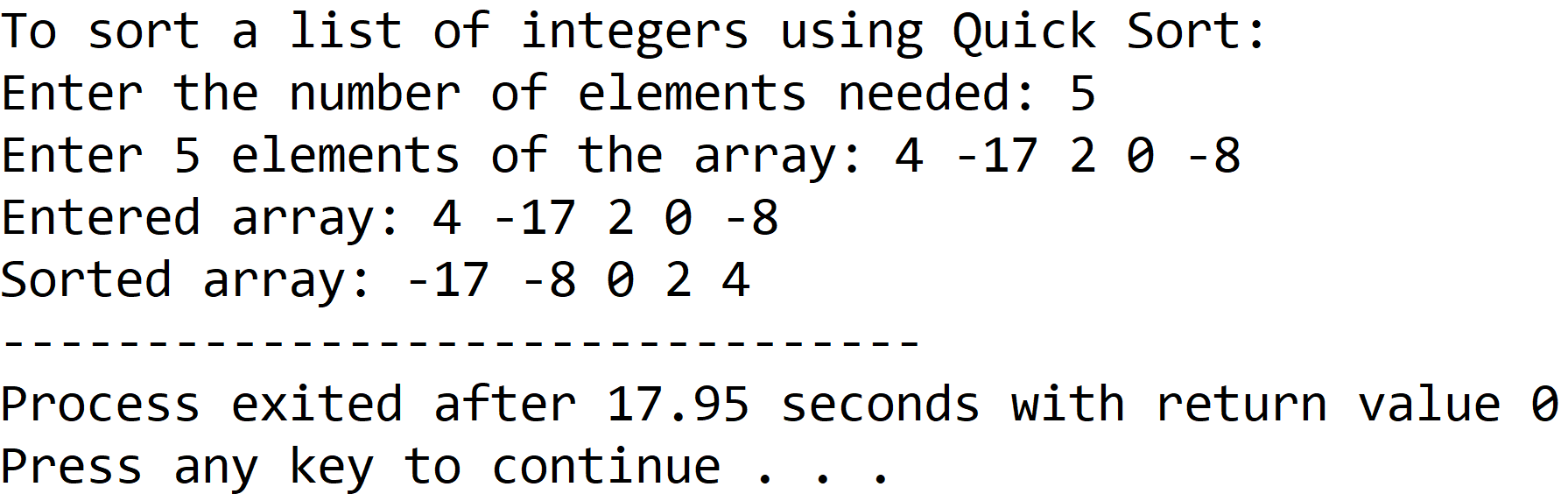
disparr(arr,size);

return 0;

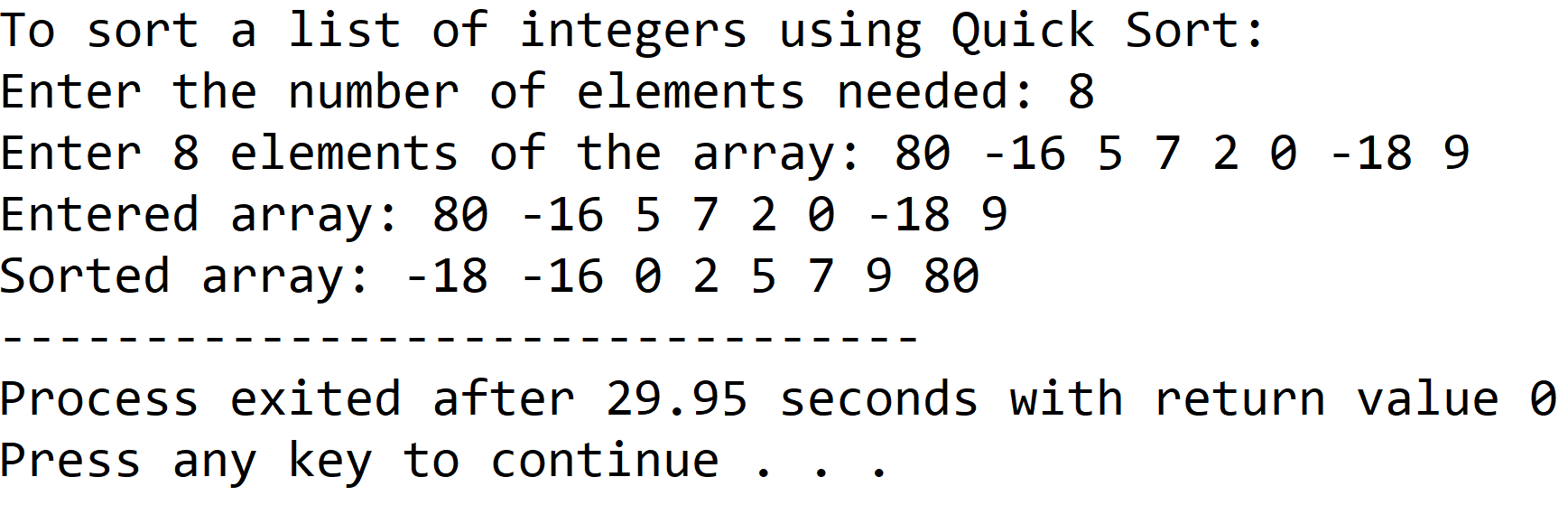
}

**4.OUTPUT**

**SET 1:** Odd length input array



**SET 2:** Even length input array



**5.DISCUSSIONS**

**Variable Description**

* **\*arr:** pointer to an integer array.
* **size:** length of the array input by the user.
* **left:** lowermost boundary index of concerned sublist.
* **right:** uppermost boundary index of concerned sublist.
* **loc:** index of the pivot element.
* **\*a,\*b:** pointers for swapping procedure.
* **i:** loop counter.

**Limitations**

* The program uses an integer array, which is a static data structure whose size cannot be manipulated once it is created in the computer’s memory, contiguous memory locations are needed to construct an array and if contiguous memory locations are not available, the array fails to be created.

**Uses**

* The program can be used to sort any list of integers in ascending order. For example, it can be used to sort files in a database based upon their order of creation.

**Future Scope**

* The array used in the program can be replaced with a linked list, making the program more memory efficient.

**Teacher’s Signature**

**ASSIGNMENT-7**

**1.PROBLEM STATEMENT**

Write a program in C to implement merge sort.

**2.ALGORITHMS**

Algorithm **Merge\_Sort**

**Input:**The pointer ‘arr’ to the array of integers, the left index ‘left’ and right index ‘right’ of the concerned sublist.

**Output:**The array ‘arr’ with the elements in sorted order.

**Remarks:**The function works recursively.

**Steps:**

1. **If**(left≥right) **then** // recursion termination condition
2. **Return**
3. **Else**
4. mid=(left+right)/2 //calculation of mid
5. **Merge\_Sort**(arr,left,mid) //Divide for the left subarray
6. **Merge\_Sort**(arr,mid+1,right) // Divide for the right subarray
7. **Merge**(arr,left,mid,right) // Merge the divided arrays in order

Algorithm **Merge**

**Input:** The pointer ‘arr’ to the array containing the sublists, the left index ‘left’ , the right index ‘right’ and the middle index ‘mid’.

**Output:** The elements of the two sublists merged back into the original array ‘arr’ in the right order.

**Remarks:**Merging happens using an auxillary array named ‘sorted’.

**Steps:**

1. i=left,j=mid+1,k=left //i and j hold the starting index of subarrays
2. **While**(i≤mid **AND** j≤right) **do** //while both arrays are not exhausted
3. **If**(arr[i]≤arr[j]) **then** //if left element ≤ right element
4. sorted[k]=arr[i] //insert left element in sorted array
5. i=i+1,k=k+1 //increment i and k by 1
6. **Else**
7. sorted[k]=arr[j] //insert right element in sorted array
8. j=j+1,k=k+1 //increment j and k by 1
9. **While**(i≤mid) **do** //if left sublist is remaining
10. sorted[k]=arr[i] //copy all elements to sorted array
11. k=k+1,i=i+1 //increment k and i
12. **While**(j≤right) do //if right sublist remains
13. sorted[k]=arr[j] //copy all elements to sorted array
14. k=k+1,j=j+1 //increment k and j
15. **For** i=left to right **do** //copying sorted array into input array
16. arr[i]=sorted[i]

**3.SOURCE CODE**

#include<stdlib.h>

#include<stdio.h>

void getarr(int \*arr,int size)

{

int i;

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

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

}

void disparr(int \*arr,int size)

{

int i;

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

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

}

//function to merge two subarrays of a single array

void merge(int \*arr,int left,int mid,int right)

{

int sorted[20],i,j,k=left;

i=left; //starting index of the left subarray

j=mid+1; //starting index of the right subarray

while(i<=mid && j<=right) //while both arrays are not exhausted

{

if(arr[i]<=arr[j]) //if left subarray element is greater

{

sorted[k]=arr[i]; //put the element in sorted array

k++;i++;

}

else

{

sorted[k]=arr[j]; //otherwise put the right element

k++;j++;

}

}

while(i<=mid) //if left subarray is remaining

{

sorted[k]=arr[i]; //copy all remaining elements

k++;i++;

}

while(j<=right) //if right subarray is remaining

{

sorted[k]=arr[j]; //copy all remaining elements

k++;j++;

}

for(i=left;i<=right;i++) //copy sorted array into original array

arr[i]=sorted[i];

}

//recursive function to divide the array into subarrays

void mergesort(int \*arr,int left,int right)

{

int mid;

if(right<=left) //recursion termination condition

return;

else

{

mid=(left+right)/2; //calculation of middle index

mergesort(arr,left,mid); // divide for the left subarray

mergesort(arr,mid+1,right); // divide for the right subarray

merge(arr,left,mid,right); // merge the divided arrays

}

}

int main(void)

{

int \*arr,size,left=0,right;

printf("To sort a list of integers using Merge Sort: \n");

printf("Enter the number of elements needed: ");

scanf("%d",&size);

if(size<1) //input validation

{

printf("Invalid array size, please try again");

exit(1);

}

right=(size-1);

arr=(int\*)malloc(size\*sizeof(int)); //creating array in heap

printf("Enter %d elements of the array: ",size);

getarr(arr,size);

printf("Entered array: ");

disparr(arr,size);

mergesort(arr,left,right);

printf("\nSorted Array: ");

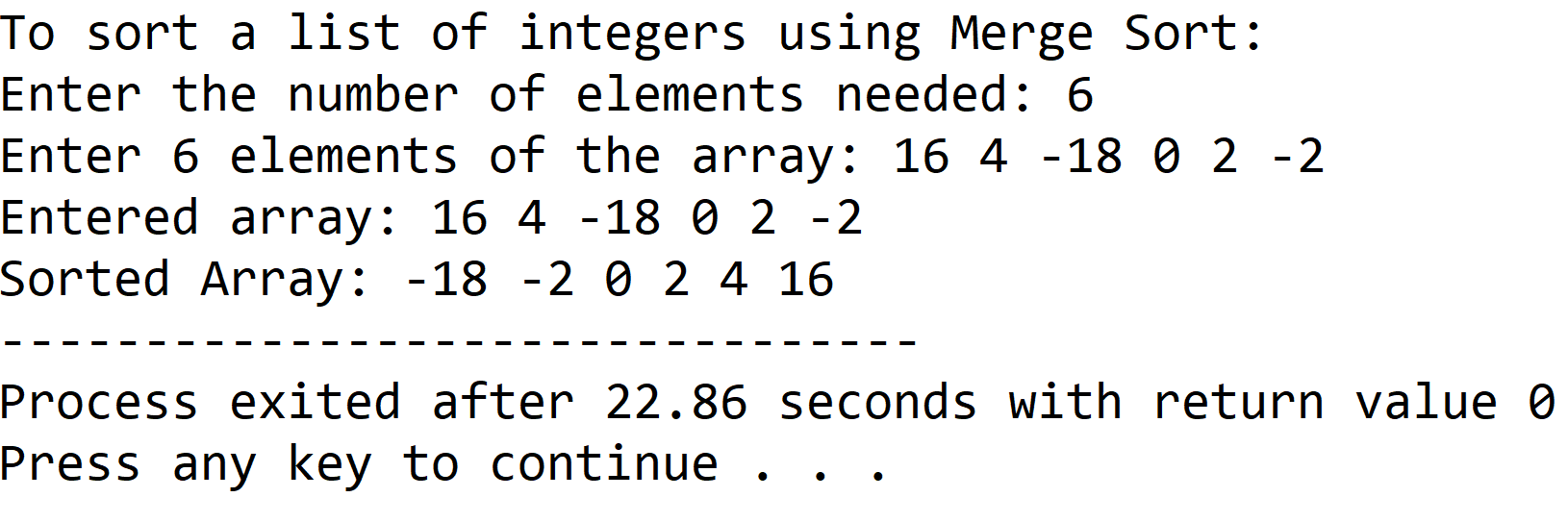
disparr(arr,size);

return 0;

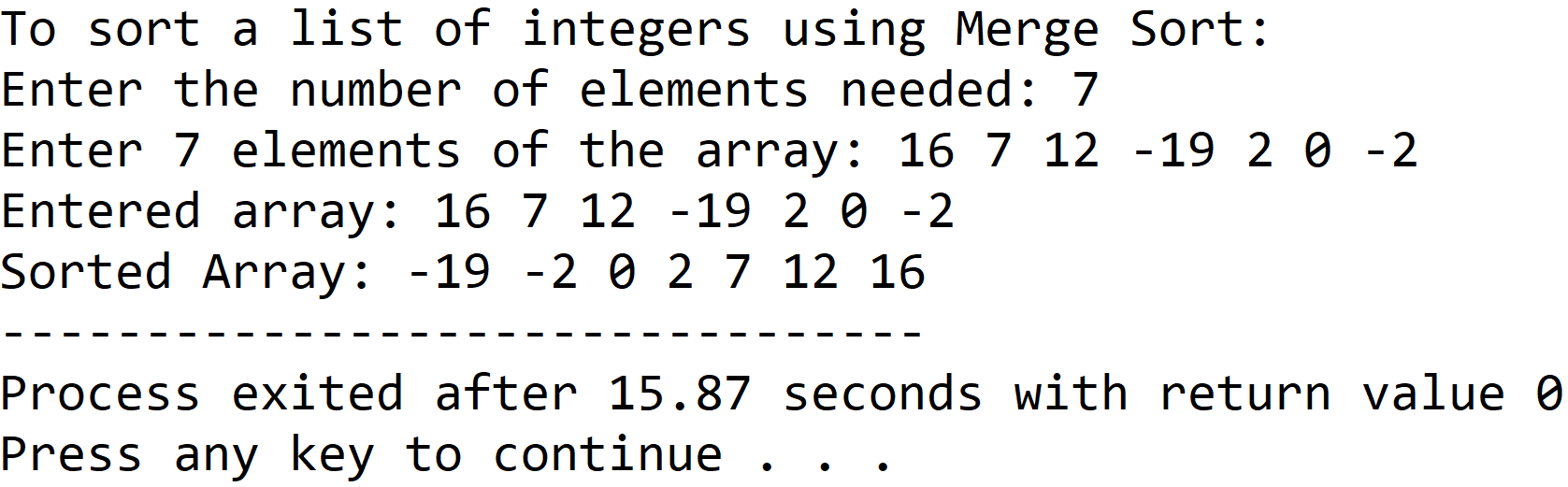
}

**4.OUTPUT**

**SET 1:** Even sized array



**SET 2:** Odd sized array



**5.DISCUSSIONS**

**Variable Description:**

* **\*arr:** pointer to an array
* **size:** the input length of the array
* **left:** lowermost index of the concerned array
* **right:** uppermost index of the concerned array
* **mid:** the index of the middle element of the concerned array
* **sorted**: auxillary array to perform merge procedure
* **i,j,k**: loop counters

**Limitations:**

* An integer array has been used to hold the list of integers, since arrays are static data structures, their size cannot be manipulated once they are constructed in the computer’s memory, also contiguous memory locations are necessary in order to create an array which is another drawback.

**Uses:**

* The program can be used to sort any list of integers in ascending order.For example, it can be used by educational institutions to arrange a random list of students on the basis of ascending order of their roll numbers.

**Future Scope:**

* The program can be made more memory efficient by replacing the array with a linked list.

**Teacher’s Signature**

**ASSIGNMENT – 8**

**1.PROBLEM STATEMENT**

Write a program in C to transpose a matrix.

**2.ALGORITHMS**

Algorithm **Transpose\_Matrix**

**Input:** The pointer ‘arr’ to the 2-D array input by the user, the dimensions of the 2-D array ‘row’ and ‘col’.

**Output:**The transpose of the 2-D array ‘arr’.

**Remarks:**The transpose operation is perfomed inplace.

**Steps:**

1. mid=1 //points to diagonal elements
2. **While**(mid<row) **do** //traversing the square part of the matrix
3. **For**(i=mid+1 to row) **do** //traversing non diagonal elements
4. **Swap**(arr[i][j],arr[j][i])//interchange the elements
5. **EndFor**
6. mid=mid+1//point mid to next diagonal element
7. **EndWhile**
8. **If**(row!=col) **then** //if input matrix is not a square matrix
9. j=row // points to non square part
10. **While**(j<col) **do** //traversing the non square part
11. **For**(i=1 to row) **do**
12. **Swap**(arr[i][j],arr[j][i])//interchange
13. **EndFor**
14. j=j+1//point j to next non square row
15. **EndWhile**
16. **EndIf**
17. **Stop**

Algorithm **Swap**

**Input:** The two variables ‘a’ and ‘b’ whose contents are to be swapped.

**Output:** The contents of ‘a’ and ‘b’ interchanged with one another.

**Remarks:** ‘a’ and ‘b’ must be passed as pointers.

**Steps:**

1. a = a+b //new value of a
2. b = a-b //b holds old value of a
3. a = a-b //a holds old value of b
4. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to take input in a matrix

void getmatrix(int (\*arr)[20],int row,int column)

{

int i,j;

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

{

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

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

}

}

//function to display a matrix

void dispmatrix(int (\*arr)[20],int row,int column)

{

int i,j;

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

{

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

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

printf("\n");

}

}

//function for interchanging variable values

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

{

\*a=\*a+\*b;

\*b=\*a-\*b;

\*a=\*a-\*b;

}

//function for input validation

void validate(int row,int col)

{

if(row<1 || col<1)

{

printf("Invalid Dimensions\nPlease Enter a Valid Dimension");

exit(1);

}

}

void trans(int(\*arr)[20],int row,int col)

{

int i,j,mid;

mid=0;

while(mid<row) //traversing the square part of the input matrix

{

for(i=mid+1;i<row;i++)

swap(&arr[mid][i],&arr[i][mid]);

mid++;

}

if(row!=col) //if input matrix is not a square matrix

{

j=row;

while(j<col)//traversing the non square part of the matrix

{

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

swap(&arr[i][j],&arr[j][i]);

j++;

}

}

}

int main(void)

{

int arr[20][20],row,col;

printf("Enter the number of rows needed: ");

scanf("%d",&row);

printf("Enter the number of columns needed: ");

scanf("%d",&col);

validate(row,col); //input validation

printf("Enter %d elements of the matrix: \n",row\*col);

getmatrix(arr,row,col);

printf("\nElements of the matrix: \n");

dispmatrix(arr,row,col); //displaying input matrix

trans(arr,row,col);

printf("\nTranspose: \n");

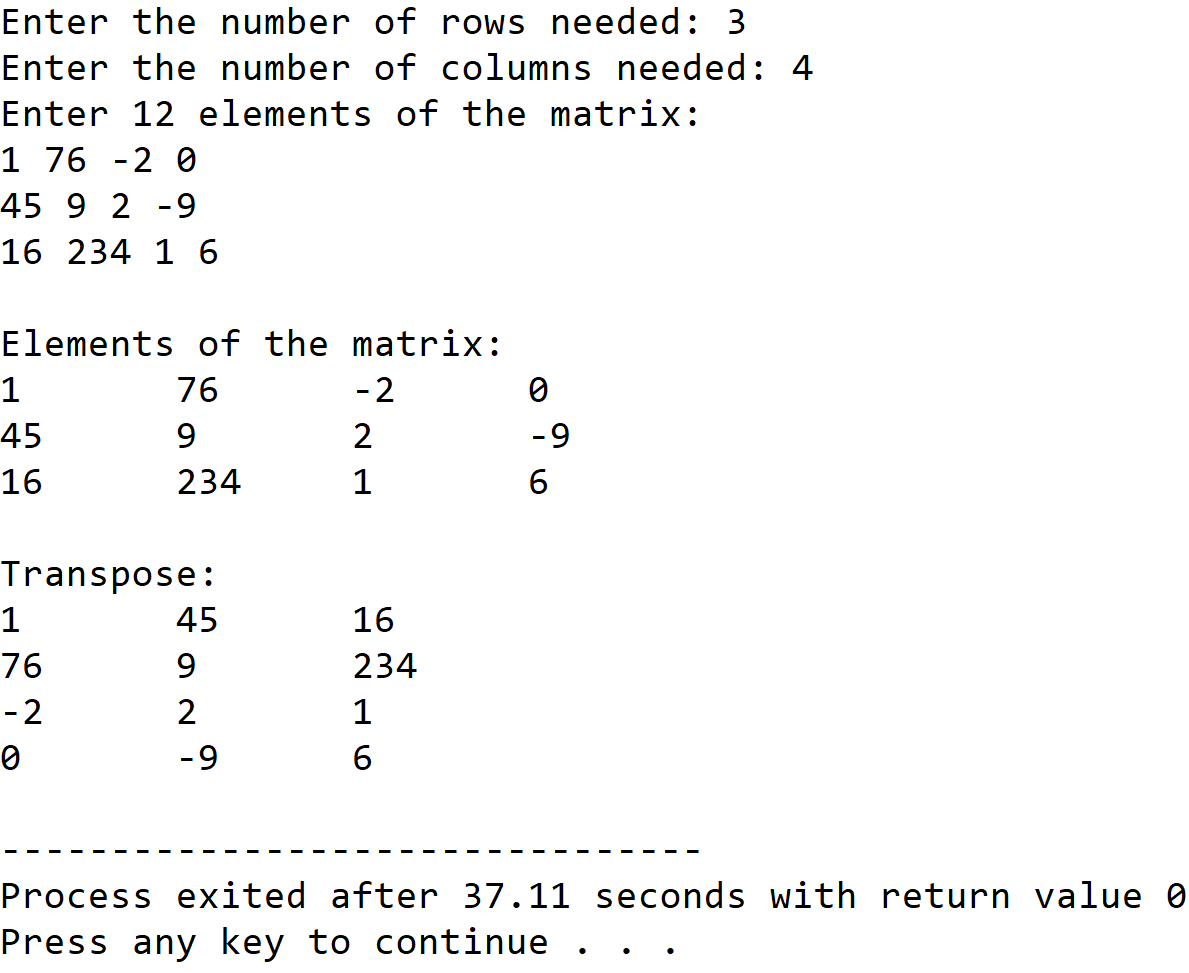
dispmatrix(arr,col,row);//displaying transposed input matrix

return 0;

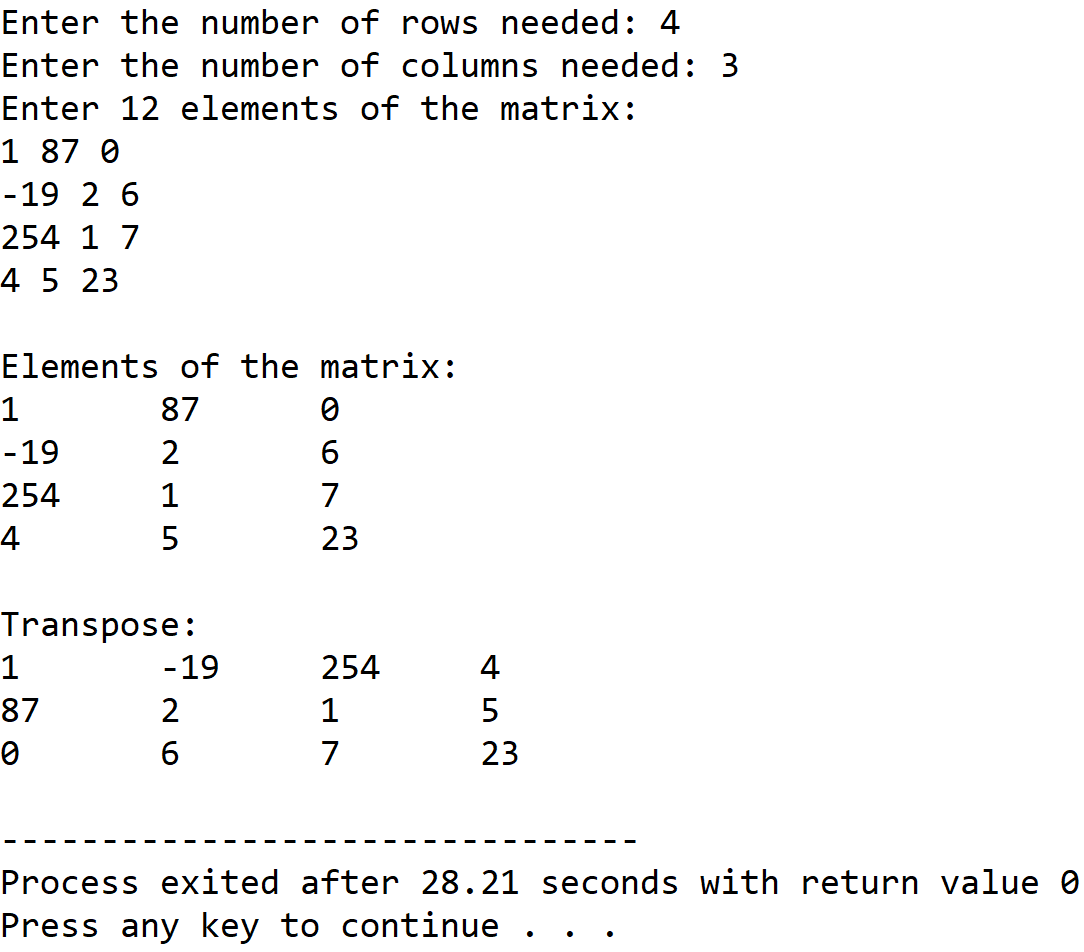
}

**4.OUTPUT**

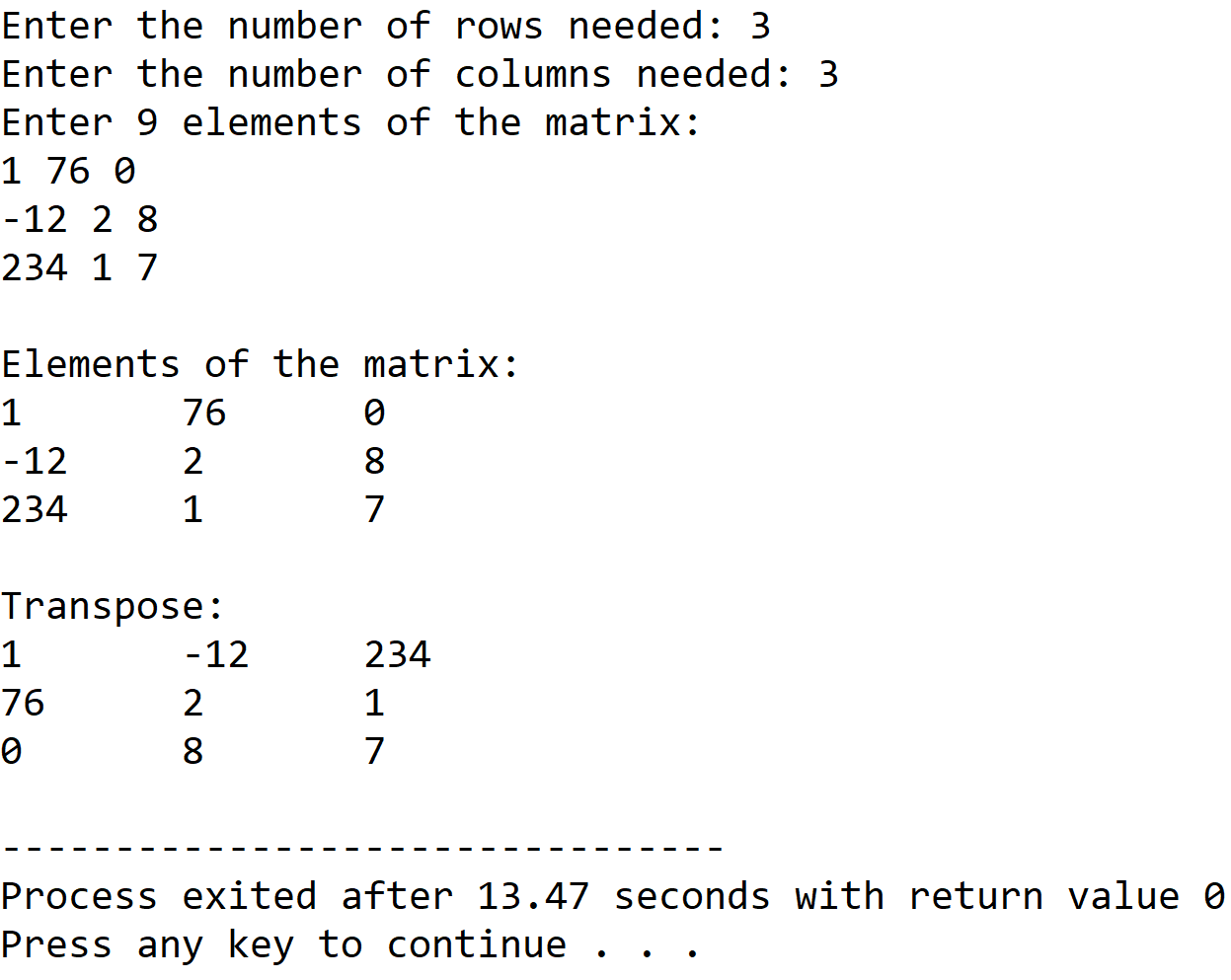
**SET 1:** Wide input matrix(row<col)



**SET 2:** Tall input matrix(row>col)



**SET 3:** Square matrix



**5.DISCUSSIONS**

**Variable Description:**

* **arr:** pointer to the 2-D array holding the matrix.
* **row,col**: dimensions of the matrix.
* **mid**:index of diagonal elements of square part of input matrix.
* **i,j**: loop counters

**Limitations:**

* The two dimensional array used in the program is statically allocated, which means that there could be wastage of memory or lack of required memory.

**Uses:**

* The program can be used to transpose any matrix containing integer values. The program can find in fields of data science in which transpose is a necessary operation for other operations like finding inverse of a matrix.

**Future Scope:**

* The two dimensional array used in the program can be allocated dyamically by the use of an array of pointers.

**Teacher’s Signature**

**ASSIGNMENT-9**

**1.AIM**

Write a program in C to implement diagonal matrix using one dimensional array

**2.ALGORITHM**

**Algorithm Is\_Diagonal\_Matrix**

**Input:** A pointer arr to the two dimesional matrix taken as input and the dimension ‘max’ .

**Output:** If the input matrix is not a diagonal matrix , an error message is displayed and the program terminates,else nothing is returned.

**Remarks:** Input matrix must be a square matrix.

**Steps:**

1. for(i=1 to max) //row traversal
2. for(j=1 to max) //column traversal
3. if(i≠j) //for non diagonal elements
4. if(arr[i][j]≠0)
5. print”Input Matrix Is Not A Diagonal Matrix”
6. Exit //terminate the program
7. **Return** //if above does not terminate,control is returned back
8. **Stop**

**3.SOURCE CODE**

**#include<stdio.h>**

**#include<stdlib.h>**

//function to take input in a matrix from the user

void getmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

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

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

}

//function to display a matrix

void dispmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

{

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

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

printf("\n");

}

}

//function to check if a matrix is a diagonal matrix

void isdiagmat(int(\*arr)[20],int row,int col)

{

int i,j;

for(i=0;i<row;i++) //row traversal

for(j=0;j<col;j++) //column traversal

{

if(i!=j)//for non diagonal elements

if(arr[i][j]!=0) // if not equal to 0

{

printf("\nEntered Matrix Is Not A Diagonal Matrix\n");

exit(1);

}

}

printf("\nEntered Matrix Is A Diagonal Matrix");

}

//function to store the diagonal elements in a 1-D array

void genlist(int(\*arr)[20],int \*list,int row,int col)

{

int i,j,k=0;

for(i=0;i<row;i++) // row traversal

for(j=0;j<col;j++) //column traversal

if(i==j) // for diagonal elements

{

list[k]=arr[i][j]; //assigning to 1-D array

k++;

}

}

//function to display an array

void displist(int\* list , int max)

{

int i;

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

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

}

//function to reconstruct matrix from 1-D array

void recons(int\*list,int max)

{

int i,j;

for(i=0;i<max;i++) //row traversal

{

for(j=0;j<max;j++)//column traversal

{

if(i==j)//for diagonal positions

printf("%d\t",list[i]);//print array

else

printf("0\t");

}

printf("\n");

}

}

int main(void)

{

int arr[20][20],\*list,row,col;

//input validation

while(1)

{

printf("Enter the number of rows needed: ");

scanf("%d",&row);

printf("Enter the number of columns needed: ");

scanf("%d",&col);

if(row==col)

break;

else

printf("Input Matrix Is Not A Square Matrix....Please Try Again");

}

printf("Enter %d elements of the matrix: \n",row\*col);

getmat(arr,row,col); //taking input in matrix

printf("\nEntered Matrix: \n");

dispmat(arr,row,col);//displaying entered matrix

isdiagmat(arr,row,col);//checkin for diagonal matrix

list=(int\*)calloc(row,sizeof(int));//creating array in heap

genlist(arr,list,row,col);//setting up array

printf("\nGenerated List: ");

displist(list,row);//displaying generated array

printf("\nReconstructed Matrix:\n");

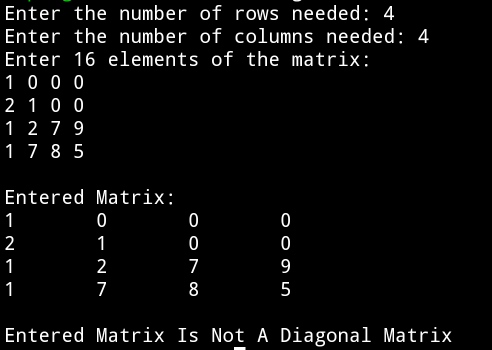
recons(list,row);//reconstructing matrix from array

return 0;

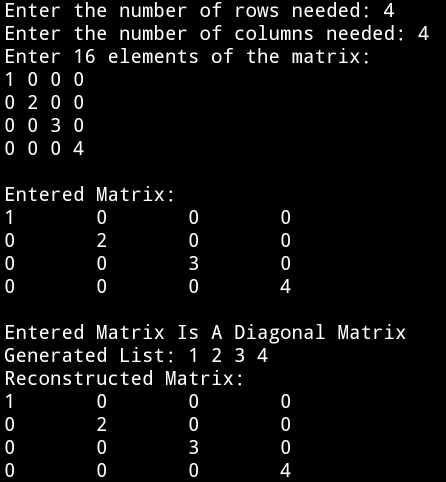
}

**4.OUTPUT**

**SET 1:** Input matrix is not a diagonal matrix



**SET 2:** Input matrix is a diagonal matrix



**5.DISCUSSIONS**

1. **Variable Description**

**In main function**

* **arr[20][20]:** 2-D array to hold a matrix
* **\*list:** Pointer to 1-D array
* **row:** number of rows in 2-D matrix
* **col:** number of columns in 2-D matrix

**loop counters used: i,j,k**

1. **Limitations**

* The program uses arrays to hold the input matrix and 1-D array, since arrays are a static data structure,their size cannot be changed once they are constructed in the memory.

1. **Uses**

* In a diagonal matrix, all the elements except the main diagonal elements are zero , if we store only the main diagonal elements in a one dimesional array and discard all the non diagonal values since they are zero, we can reconstruct the same matrix from a much smaller 1-D array.Thus this program shows memory optimization for storing data.

1. **Future Scope**

* Instead of using arrays, linked lists can be used to store the matrix and the list , allowing more flexibility in manipulation of the size.

**Teacher’s Signature**

**ASSIGNMENT-10**

**1.AIM**

Write a program in C to implement lower and upper triangular matrix using one dimensional array.

**2.ALGORITHM**

**Algorithm** Is\_Upper\_Triangular\_Matrix

**Input:** A pointer ‘arr’ to the matrix and the dimension ‘size’ of the matrix

**Output:** If the input matrix is a upper triangular matrix , nothing is returned else a error message is shown and the program terminates.

**Remarks:**It is assumed that the array is not empty.

**Steps:**

1. **For**(i=2 to size) //Traversing the lower triangular part
2. **For**(j=0 to i)
3. **If**(arr[i][j]≠0) **then** //non zero lower triangular element
4. print“Input matrix is not upper triangular matrix”
5. **Exit** // terminate the program
6. **Return** // If program does not exit, input matrix is upper triangular
7. **Stop**

**Algorithm** Is\_Lower\_Triangular\_Matrix

**Input:** A pointer ‘arr’ to the matrix and the dimension ‘size’ of the matrix

**Output:** If the input matrix is a lower triangular matrix , nothing is returned else a error message is shown and the program terminates.

**Remarks:**It is assumed that the array is not empty.

**Steps:**

1. **For**(i=2 to size) **do** //Traversing the upper triangular part
2. **For**(j=0 to i) **do**
3. **If**(arr[j][i]≠0) **then** //non zero upper triangular element
4. **print**“Input matrix is not lower triangular matrix”
5. **Exit** // terminate the program
6. **Return** // If program does not exit, input matrix is lower triangular
7. **Stop**

**Algorithm** Make\_Upper\_Triangular\_List

**Input:** A pointer ‘arr’ to the matrix , the dimesion ‘size’ and the pointer ‘list’ to the one dimesional array in which the upper triangular part is to be stored.

**Output:** The upper triangular part of ‘arr’ is fed into the one dimesional array ‘list’ by reference

**Remarks:** Size of the one dimesional array = (size(size+1))/2

**Steps:**

1. **For**(i=1 to size) **do** // Traversing the upper triangular part
2. **For**(j=i to size) **do**
3. list[k]=arr[i][j] // Feeding the one dimesional array
4. k = k+1
5. **Return**
6. **Stop**

**Algorithm** Make\_Lower\_Triangular\_List

**Input:** A pointer ‘arr’ to the matrix , the dimesion ‘size’ and the pointer ‘list’ to the one dimesional array in which the lower triangular part is to be stored.

**Output:** The lower triangular part of ‘arr’ is fed into the one dimesional array ‘list’ by reference

**Remarks:** Size of the one dimesional array = (size(size+1))/2

**Steps:**

1. **For**(i=1 to size) **do** // Traversing the lower triangular part
2. **For**(j=i to (i+1) **do**
3. list[k]=arr[i][j] // Feeding the one dimesional array
4. k=k+1
5. **Return**
6. **Stop**

**Algorithm** Reconstruct\_Upper\_Triangular\_Matrix

**Input:** A pointer ‘arr’ to the one dimesional array in which the upper triangular part is stored and the dimesion ‘size’ of the matrix to be reconstructed.

**Output:** The original input matrix is displayed using only the one dimesional array that holds the upper triangular part.

**Remarks:**Matrix is not recreated physically , it is only displayed.

**Steps:**

1. limit=size
2. **For**(i=1 to size) **do** // For non upper diagonal positions
3. **For**(j=limit to size)
4. **Print** “0” // display 0
5. **For**(k=1 to limit) // For upper diagonal positions
6. **Print** arr[l] // Display the contents of 1-D array
7. l=l+1
8. limit=limit-1
9. **Stop**

**Algorithm** Reconstruct\_Lower\_Triangular\_Matrix

**Input:** A pointer ‘arr’ to the one dimesional array in which the lower triangular part is stored and the dimesion ‘size’ of the matrix to be reconstructed.

**Output:** The original input matrix is displayed using only the one dimesional array that holds the upper triangular part.

**Remarks:**Matrix is not recreated physically , it is only displayed.

**Steps:**

1. limit=size
2. **For**(i=1 to size) **do** // For lower triangular positions
3. **For**(j=(limit-1) to size)
4. **Print** arr[l] // Display the contents of 1-D array
5. l=l+1
6. **For**(k=2 to limit) // For non lower triagular positions
7. **Print** “0” // Display 0
8. limit=limit-1
9. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

// function for input validation

void validate(int row,int col)

{

if(row!=col)

{

printf("Triangular Matrix Must Be A Square Matrix!\nPlease Try Again\n");

exit(1);

}

return;

}

// function to ask the user whether to continue with new input

void prompt(void)

{

int ch;

while(1)

{

printf("Press 1 to continue OR press 0 to exit: ");

scanf("%d",&ch);

if(ch==1)

return;

else if(ch==0)

exit(1);

printf("\nInvalid Input, Please Try Again\n");

}

}

// function to take input in a matrix

void getmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

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

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

}

//function to display a matrix

void dispmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

{

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

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

printf("\n");

}

}

//function to check whether input matrix is a upper triangular matrix

void isuppertri(int(\*arr)[20],int row,int col)

{

int i,j;

for(i=1;i<row;i++) //traversing the lower triangular part

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

if(arr[i][j]!=0) // if an element is not zero

{

printf("\nEntered Matrix Is Not A Upper Triangular Matrix\n");

exit(1); // terminate the program

}

printf("\nEntered Matrix Is A Upper Triangular Matrix");

}

//function to check whether input matrix is a lower triangular matrix

void islowertri(int(\*arr)[20],int row,int col)

{

int i,j;

for(i=1;i<row;i++) //traversing the upper triangualar part

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

if(arr[j][i]!=0) //if an element is not zero

{

printf("\nEntered Matrix Is Not A Lower Triangular Matrix\n");

exit(1); //terminate the program

}

printf("\nEntered Matrix Is A Lower Triangular Matrix");

}

//function to feed upper triangular matrix in 1-D array

void makeupperlist(int\*list,int(\*arr)[20],int row,int col)

{

int i,j,k=0;

for(i=0;i<row;i++) //traversing the upper triangular part

for(j=i;j<col;j++)

{

list[k]=arr[i][j]; //feeding the 1-D array

k++;

}

}

//function to feed lower triangular matrix in 1-D array

void makelowerlist(int\*list,int(\*arr)[20],int row,int col)

{

int i,j,k=0;

for(i=0;i<row;i++) //traversing the lower triangular part

for(j=0;j<(i+1);j++)

{

list[k]=arr[i][j]; //feeding the 1-D array

k++;

}

}

//function to display a 1-D array

void displist(int\*arr,int size)

{

int i;

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

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

}

//function to reconstruct matrix from 1-D array

void reconsuppmat(int\* arr,int row)

{

int i,j,k,l=0,limit;

limit=row;

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

{

for(j=limit;j<row;j++) //for non upper triangular positions

printf("0\t"); //display 0

for(k=0;k<limit;k++) // for upper triangular positions

{

printf("%d\t",arr[l]); //display contents of 1-D array

l++;

}

limit--;

printf("\n");

}

}

//function to reconstruct matrix from 1-D array

void reconslowmat(int\* arr,int row)

{

int i,j,k,l=0,lim=row;

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

{

for(j=(lim-1);j<row;j++) //for lower triangular positions

{

printf("%d\t",arr[l]); //display contents of 1-D array

l++;

}

for(k=1;k<lim;k++) //for non lower triangular positions

printf("0\t"); // display 0

lim--;

printf("\n");

}

}

int main(void)

{

int ch,row,col,arr[20][20],\*list,size;

while(1)

{

printf("Enter Number Of Rows: ");

scanf("%d",&row);

printf("Enter Number Of Columns: ");

scanf("%d",&col);

validate(row,col);

printf("\nEnter %d elements of the matrix: \n",row\*col);

getmat(arr,row,col);

printf("\nEntered Matrix: \n");

dispmat(arr,row,col);

printf("\nMenu: ");

printf("\n1.Upper Triangular Matrix\n2.Lower Triangular Matrix");

printf("\nEnter Your Choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

isuppertri(arr,row,col);

size=(row\*(row+1))/2; //length of 1-D array

list=(int\*)calloc(size,sizeof(int));

makeupperlist(list,arr,row,col);

printf("\nGenerated List: ");

displist(list,size);

printf("\nReconstructed Matrix:\n");

reconsuppmat(list,row);

prompt();

break;

case 2:

islowertri(arr,row,col);

size=(row\*(row+1))/2; //length of 1-D array

list=(int\*)calloc(size,sizeof(int));

makelowerlist(list,arr,row,col);

printf("\nGenerated List: ");

displist(list,size);

printf("\nReconstructed Matrix:\n");

reconslowmat(list,row);

prompt(); //ask the user whether to exit

return 0;

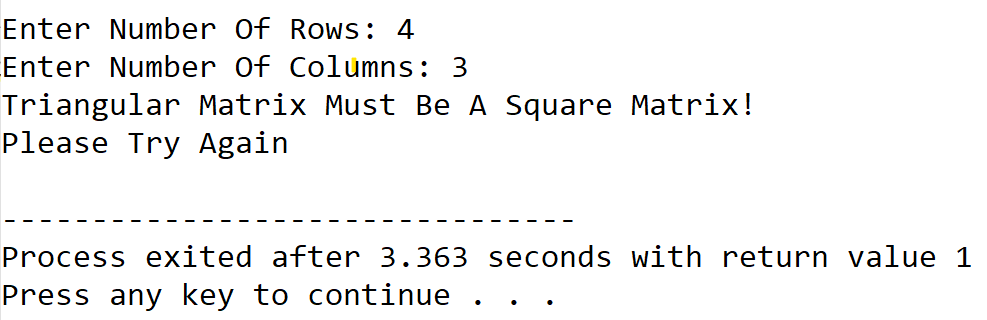
}

}

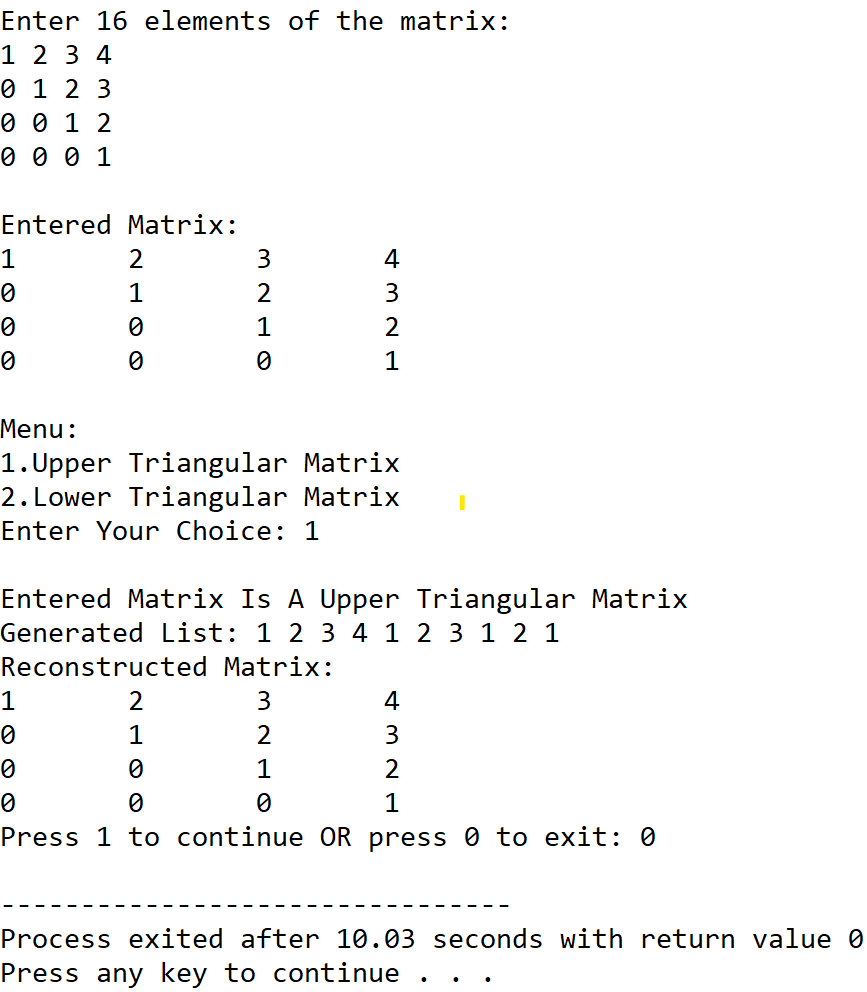
}

**4.OUTPUT**

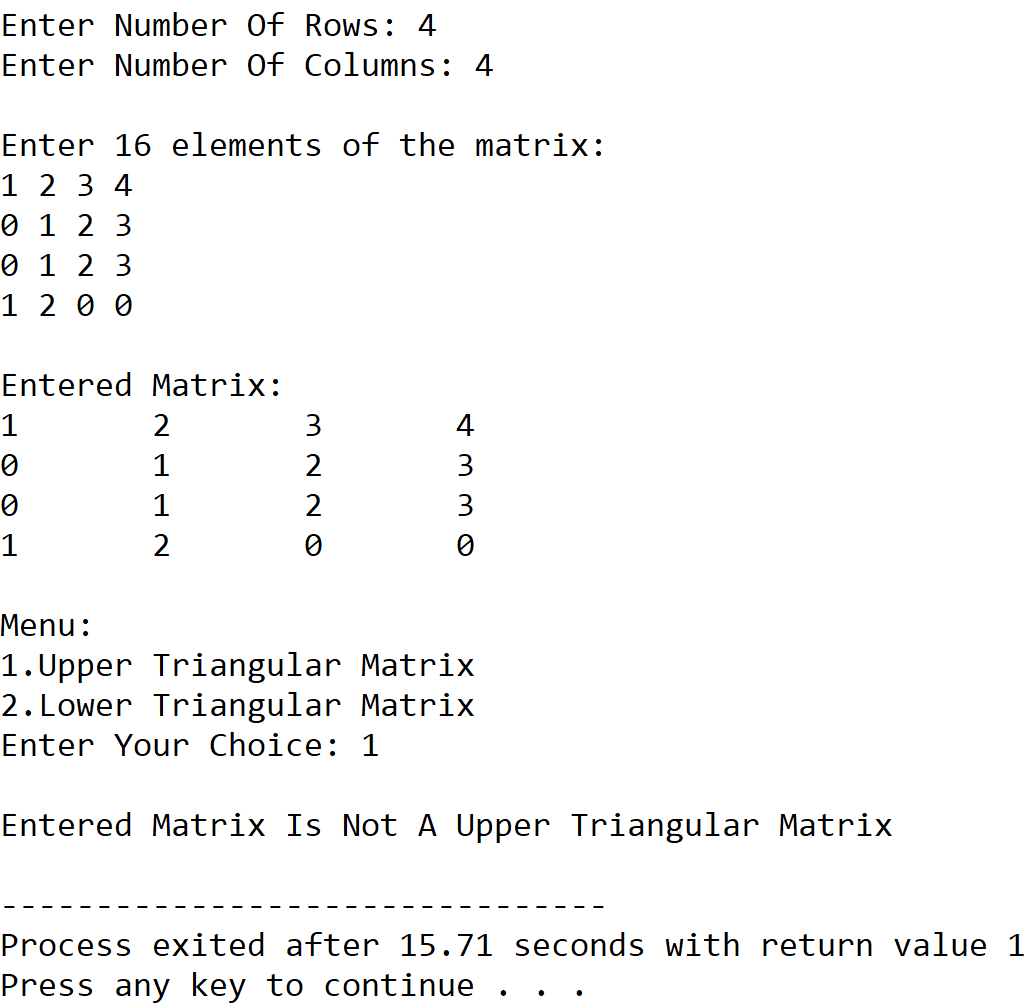
**SET 1:** Input matrix is not a square matrix



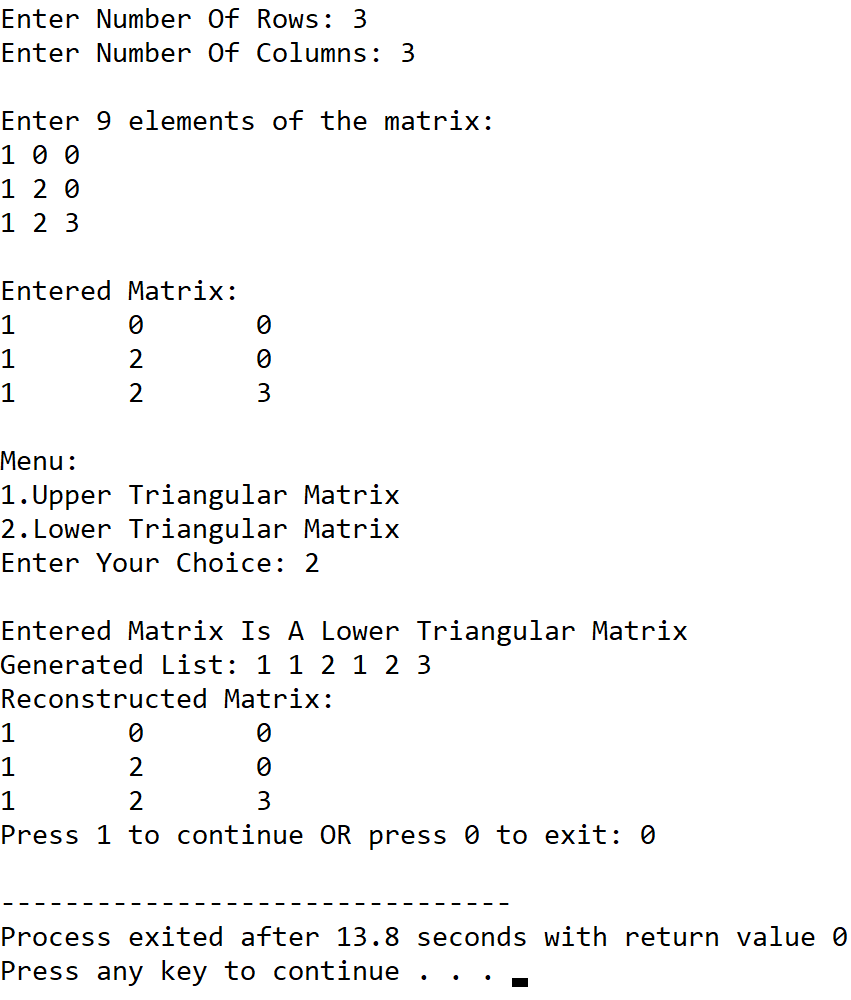
**SET 2:** Input matrix is an upper triangular matrix



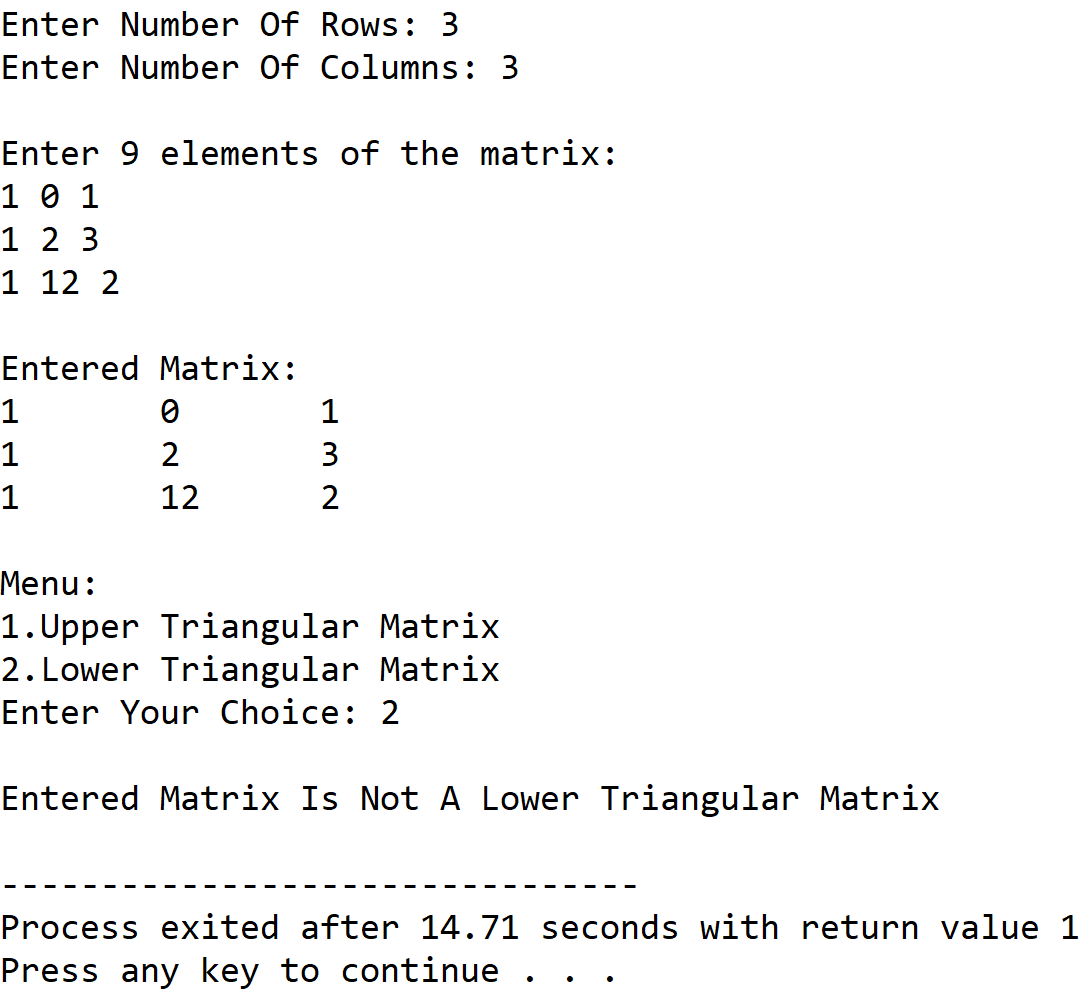
**SET 3:** Input matrix is not an upper triangular matrix



**SET 4:** Input matrix is a lower triangular matrix



**SET 5:** Input matrix is not a lower triangular matrix



**5.DISCUSSIONS**

**A. Variable Description**

* **row,col:** Dimesion of the matrix.
* **arr:** Two dimensional array to hold the matrix.
* **\*list:** pointer to a one dimesional array.
* **size:** length of the one dimesional array.
* **ch:** variable to receive user’s choice in switch-case-default.

**B. Limitations**

* The program uses a one dimensional array to hold the triangular sections of the matrix, as array is a static data structure ,it’s size cannot be altered once it its constructed in the memory.
* The two dimensional array is not dynamically allocated and thus leads to inefficient use of memory.

**C. Uses**

* The program showcases that triangular matrices can be represented by using less scomputer memory without any loss in information if only the upper triangular or lower triangular part is stored in a one dimesional array.This displays how a particular data structure can be represented in tne memory in a much more optimized way as compared to traditional methods.

**D. Future Scope**

* The two dimesional array can be dynamically allocated for more efficient memory use.
* The one dimensional array can be replaced with a linked list for more efficient memory use

**Teacher’s Signature**

**ASSIGNMENT – 11**

**1.PROBLEM STATEMENT**

Write a program in C to implement symmetric matrix using one dimensional array

**2.ALGORITHMS**

Algorithm **Is\_Symmetric\_Matrix**

**Input:** The matrix arr[row][col] with its dimensions ‘row’ and ‘col’.

**Output:** If the entered matrix arr[row][col] is not a symmetric matrix, an error message is shown and a flag variable indicates invalid input.

**Remarks:** It is assumed that the matrix is not empty.

**Steps:**

1. **For**(i=1 to row) **do** //traversing the matrix
2. **For**(j=1 to col) **do**
3. **If**(i≠j) **then** //for non diagonal elements
4. **If**(arr[i][j]≠arr[j][i]) **then** //unequal transpose values
5. **Print** “Input matrix is not symmetric
6. **Exit** //terminate the procedure
7. **EndIf**
8. **EndIf**
9. **EndFor**
10. **EndFor**

Algorithm **Symmetric\_Matrix\_To\_1D\_Array**

**Input:** The matrix arr[row][col] an the list in which the symmetric matrix is to be stored.

**Output:** The upper triangular matrix stored in the array representing the symmetric matrix.

**Remarks:** Memory needed in the 1-D array: row\*(row+1)/2

**Steps:**

1. k=0
2. **For**(i=0 to row) **do** //traversing the matrix
3. **For**(j=1 to col) **do**
4. list[k]=arr[i][j] /mapping matrix linearly to array
5. k=k+1
6. **EndFor**
7. **EndFor**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function for input validation

void validate(int row,int col)

{

    if(row!=col)

    {

        printf("Triangular Matrix Must Be A Square Matrix!\nPlease Try Again\n");

        exit(1);

    }

    return;

}

//function to take input in a matrix

void getmat(int(\*arr)[20],int row,int col)

{

    int i,j;

    for(i=0;i<row;i++)

        for(j=0;j<col;j++)

            scanf("%d",&arr[i][j]);

}

//function to display a matrix

void dispmat(int(\*arr)[20],int row,int col)

{

    int i,j;

    for(i=0;i<row;i++)

    {

        for(j=0;j<col;j++)

            printf("%d\t",arr[i][j]);

        printf("\n");

    }

}

//function to map matrix to linear array

void makelist(int\*list,int(\*arr)[20],int row,int col)

{

    int i,j,k=0;

    for(i=0;i<row;i++) //traversing the matrix

        for(j=i;j<col;j++)

        {

            list[k]=arr[i][j]; //mapping matrix to array

            k++;

        }

}

//function to display a 1-D array

void displist(int\*arr,int size)

{

    int i;

    for(i=0;i<size;i++)

        printf("%d ",arr[i]);

}

//function to check if a given matrix is symmetric or not

void issymatrix(int(\*arr)[20],int row,int col)

{

    int i,j;

    for(i=0;i<row;i++) //traversing

        for(j=0;j<col;j++)

            if(i!=j) //for non diagonal elements

            {

                if(arr[i][j]!=arr[j][i]) //for unequal transpose elements

                {

                    printf("Entered matrix is not symmetric");

                    exit(0);

                }

            }

}

int main(void)

{

    int ch,row,col,arr[20][20],\*list,size;

    printf("Enter Number Of Rows: ");

    scanf("%d",&row);

    printf("Enter Number Of Columns: ");

    scanf("%d",&col);

    validate(row,col); //input validation

    printf("\nEnter %d elements of the matrix: \n",row\*col);

    getmat(arr,row,col);

    printf("\nEntered Matrix: \n");

    dispmat(arr,row,col);

    issymatrix(arr,row,col); //checking entered matrix

    puts("\nEntered Matrix Is Symmetric");

    size=(row\*(row+1))/2; //number of elements of 1-D array

    list=(int\*)malloc(size\*sizeof(int));

    makelist(list,arr,row,col);

    printf("\nGenerated List: ");

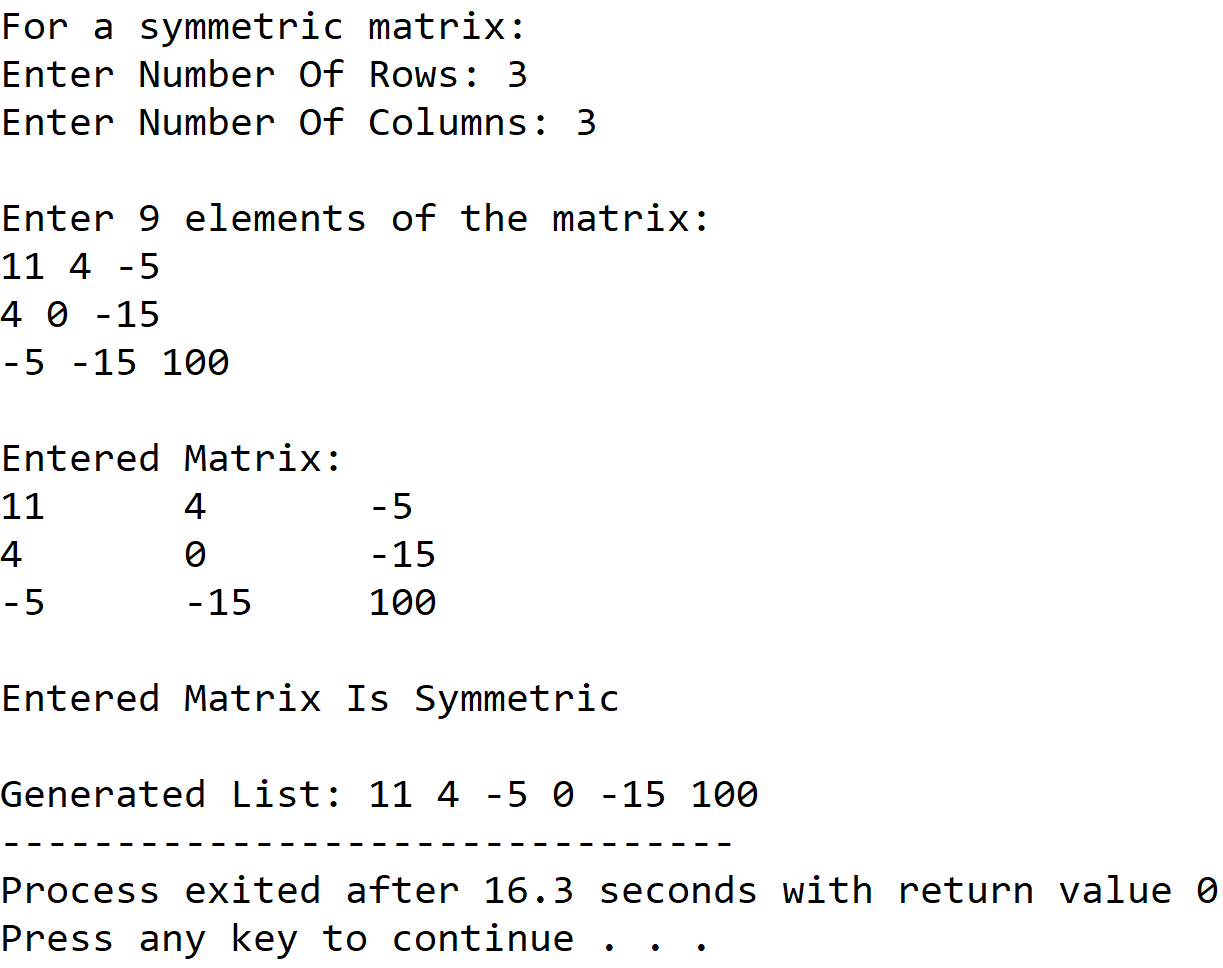
    displist(list,size);

    return 0;

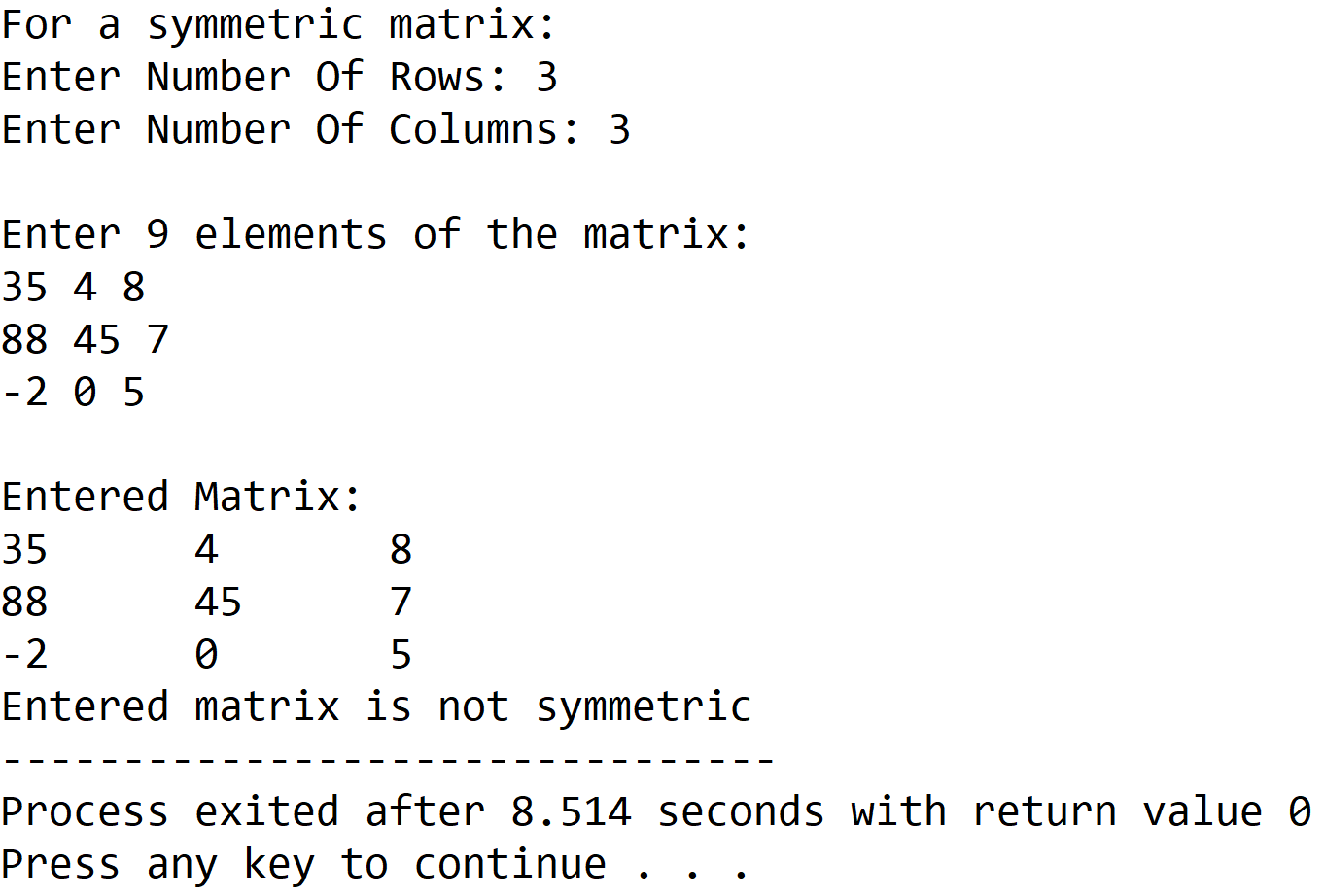
}

**4.OUTPUT**

**SET 1:** Input matrix is a symmetric matrix



**SET 2:** Input matrix is not a symmetric matrix



**5.DISCUSSIONS**

**Variable Description**

* **arr:** two dimensional array to store a matrix.
* **\*list:** pointer to an integer array to hold the symmetric matrix.
* **size:** number of elements required in list.
* **row,col:** dimensions of arr.
* **i,j,k:** loop counters.

**Limitations**

* The program uses a statically declared two dimensional matrix thus memory locations are either wasted or there is a shortage of memory.
* The symmetric matrix is stored in a 1-D array which is a static data structure which means that its size cannot be changed once it is constructed in memory.

**Uses**

* The program depicts how a particular data structure like a symmetric matrix can be stored in a much more efficient manner by holding only that data which is essential and not including repetitive, unrequired data.

**Future Scope**

* The 1-D array can be replaced with a linked list eliminating the need of contiguous memory locations and also making it easier to manipulate the size.
* The two dimensional array can be allocated dynamically using an array of pointers.

**Teacher’s Signature**

**ASSIGNMENT – 12**

**1.PROBLEM STATEMENT**

Write a program in C to implement sparse matrix using one dimensional array.

**2.ALGORITHMS**

Algorithm **Sparse\_Matrix\_to\_1D\_Array**

**Input:**The pointer ‘arr’ to the 2-D array holding the matrix, the pointer ‘list’ to the array in which sparse matrix is to be stored and the dimensions ‘row’ and ‘col’ of the matrix.

**Output:** The elements of the sparse matrix fed into the array ‘list’

**Remarks**:

* If total number of non-zero elements in the input matrix is less than one third of the total elements, then the matrix is taken to be a sparse matrix.
* The list that stores the sparse matrix has three sections within each variable, namely row,col,ele for storing row value,column value and element value respectively.

**Steps:**

1. k=0 // points at beginning of list
2. **For**(i=1 to row) **do** //traversing along rows
3. **For**(j=1 to col) **do** //traversing along columns
4. **If**(arr[i][j]≠0) **then //**if an element is not zero
5. row🡨list[k]🡨i //store i in row section of list
6. col🡨ist[k]🡨j //store j in col section of list
7. ele🡨list[k]🡨arr[i][j]//store value in ele section
8. k=k+1//point k to next variable in list
9. **EndIf**
10. **EndFor**

**11. EndFor**

**12.Stop**

Algorithm **Count\_Nonzero\_Elements**

**Input:**The pointer to the matrix ‘arr’, and the dimensions of the matrix ‘row’ and ‘col’.

**Output:** The count of the total number of non-zero elements.

**Remarks**: It is considered that the matrix is not empty.

**Steps:**

1. cnt=0 //set cnt to zero
2. **For**(i=1 to row) **do** //traversing along rows
3. **For**(j=1 to col) **do** //traversing along columns
4. **If**(arr[i][j]≠0 **then** //if array element is not zero
5. cnt=cnt+1 // increment cnt by 1
6. **EndIf**
7. **EndFor**
8. **EndFor**
9. **Return** cnt // return the value of cnt
10. **Stop**

Algorithm **Is\_Sparse\_Matrix**

**Input:** The pointer to the matrix ‘arr’, and the dimensions of the matrix ‘row’ and ‘col’.

**Output:** If the entered matrix is a sparse matrix, a successful message is shown and the procedure returns **True** otherwise, it returns **False** with an unsuccessful message.

**Remarks:** If total number of non-zero elements in the input matrix is less than one third of the total elements, then the matrix is taken to be a sparse matrix.

**Steps:**

1. count=**Count\_Nonzero\_Elements**(arr,row,col)
2. limit=(row\*col)/3 // maximum number of nonzero elements
3. **If**(count<limit) **then** //if number of nonzero elements is in range
4. **Print** “Input matrix is sparse”
5. **Return**
6. **Else**
7. **Print** “Input matrix is not sparse”
8. **Exit** //terminate the program
9. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//structure to hold information about sparse matrix elements

typedef struct spmat

{

int row;

int col;

int ele;

}spmat;

//function to take input in a matrix

void getmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

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

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

}

//function to display a matrix

void dispmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

{

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

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

printf("\n");

}

}

//function to count the number of nonzero elements in a matrix

int nonzero(int(\*arr)[20],int row,int col)

{

int i,j,cnt=0;

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

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

if(arr[i][j]!=0)

cnt++;

return cnt;

}

//function to tell if a input matrix is sparse or not

void issparse(int(\*arr)[20],int row,int col)

{

int cnt,limit,total=row\*col;

cnt=nonzero(arr,row,col);

limit=total/3;

if(cnt<limit) //if number of nonzero elements is in allowed range

{

printf("Input matrix is a sparse matrix\n");

return;

}

else

{

printf("Input matrix is not a sparse matrix\n");

exit(0);

}

}

//function to store sparse matrix elements in 1-D array

void makesparsearr(int(\*arr)[20],spmat \*list,int row,int col)

{

int i,j,k=0;

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

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

if(arr[i][j]!=0)

{

list[k].row=i;

list[k].col=j;

list[k].ele=arr[i][j];

k++;

}

}

//function to recreate sparse matrix from the auxillary 1-D array

void dispsparse(spmat \*list,int cnt)

{

int i=0;

printf("ROW\tCOLUMN\tELEMENT\n");

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

{

printf("%d\t%d\t%d\n",list[i].row,list[i].col,list[i].ele);

}

}

//function to display an array

void disparr(spmat \*arr,int size)

{

int i;

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

{

printf("%d %d %d ",arr[i].row,arr[i].col,arr[i].ele);

}

}

int main(void)

{

int arr[20][20],row,col,cnt;

spmat \*list; //pointer to a array of structures

printf("Enter the number of rows: ");

scanf("%d",&row);

printf("Enter the number of columns: ");

scanf("%d",&col);

printf("Enter %d elements of the matrix: \n",row\*col);

getmat(arr,row,col);

printf("Entered matrix: \n");

dispmat(arr,row,col);

issparse(arr,row,col);

cnt=nonzero(arr,row,col);

list=(spmat\*)malloc(cnt\*sizeof(spmat));

makesparsearr(arr,list,row,col);

printf("\nElements of the 1-D array: ");

disparr(list,cnt);

printf("\nSparse matrix constructed from 1-D array: \n");

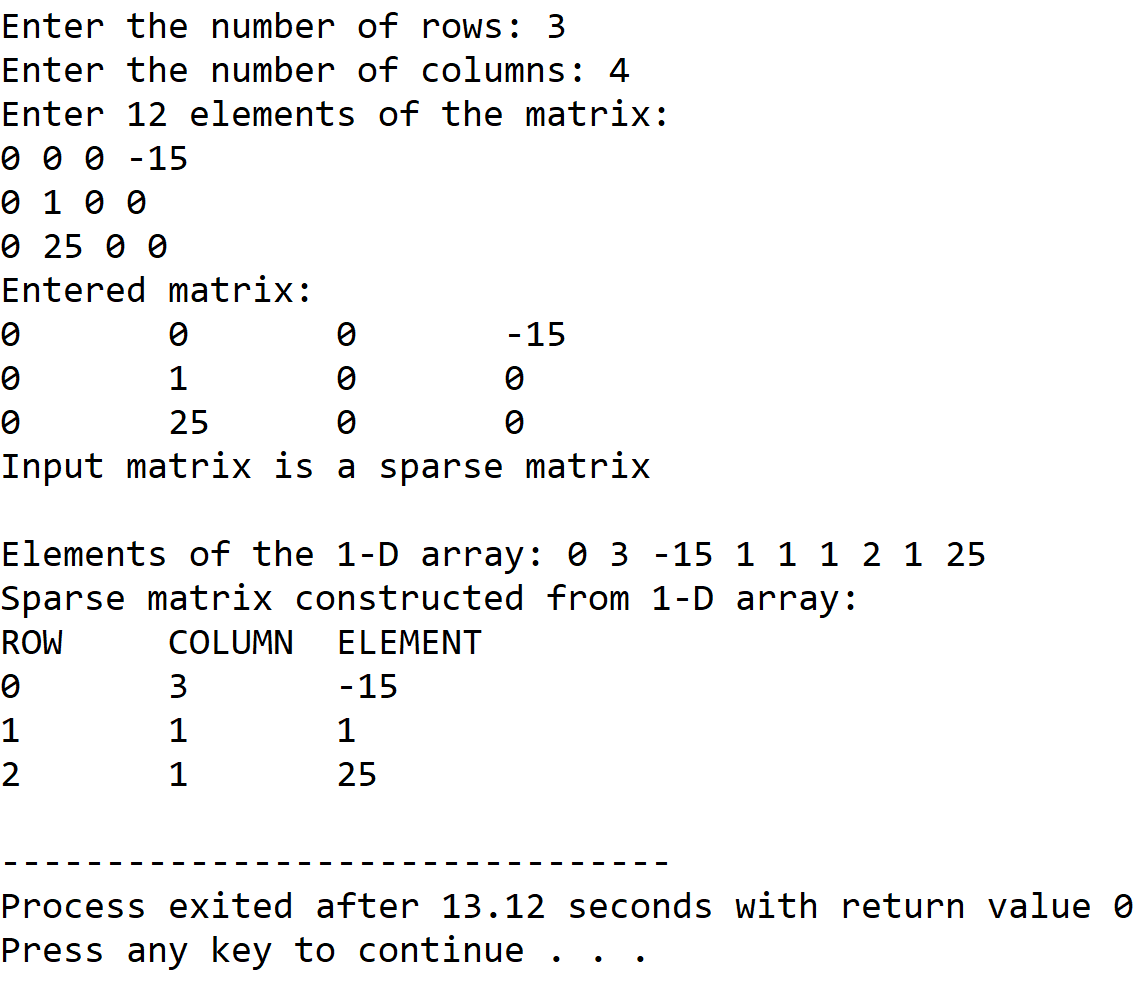
dispsparse(list,cnt);

return 0;

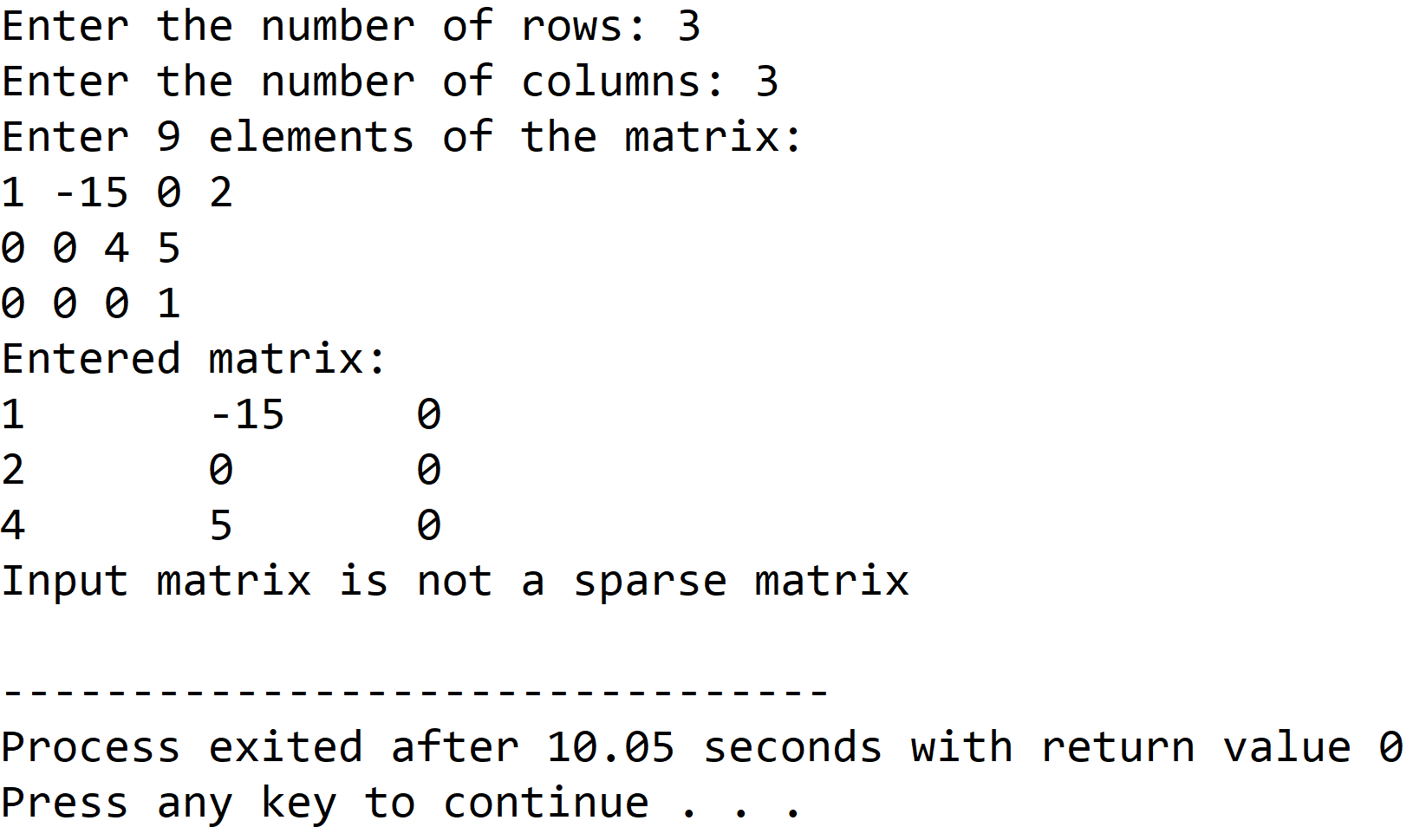
}

**4.OUTPUT**

**SET 1:**Input matrix is a sparse matrix.



**SET 1:**Input matrix is not a sparse matrix.



**5.DISCUSSIONS**

**Variable Description:**

* **arr:** Pointer to the 2-D arraty holding the matrix.
* **\*list:**Pointer to the array of structures to hold the sparse matrix information.
* **row,col:** Dimensions of the 2-D array.
* **cnt:** Count of the number of non-zero elements in matrix.
* **limit:**Maximum non-zero element count for a sparse matrix.
* **total:** Total number of elements in a matrix.
* **I,j,k:** loop counters.

**Limitations:**

* The two dimensional array used in the program is statically allocated, which means that there could be wastage of memory or lack of required memory.

**Uses:**

* The program shows how a particular data structure like a sparse matrix can be stored in the computer’s memory in a much more optimized manner if we store only that information which is unique and that cannot be reconstructed.

**Future Scope:**

* The two dimensional array used in the program can be dynamically allocated, leading to more effiecient use of memory.

**Teacher’s Signature**

**ASSIGNMENT – 13**

**1.PROBLEM STATEMENT**

Write a program in C to implement matrix multiplication

**2.ALGORITHMS**

Algorithm **Matrix\_Multiplication**

**Input:** The pointer to the matrices ‘a1’,’a2’ and the result matrix ‘res’,the dimensions of the first matrix ‘row1’,’col1’ and then dimensions of the second matrix ‘row2’,’col2’.

**Output:**The product of the matrices ‘a1’ and ‘a2’ stored in ‘a3’.

**Remarks:** Matrix multiplication can be performed only if the number of columns of the first matrix is equal to the number of rows of the second matrix. The resultant matrix has number of rows equal to that of the first matrix and number of columns to that of second matrix.

**Steps:**

1. sum=0 //setting sum variable to zero
2. **For**(i=1 to row1) **do** //traversing along rows of first matrix
3. **For**(count=0 to col2) **do** //multiplying one row with all columns
4. sum=0
5. **For**(j=1 to row2) **do** //traversing rows of second matrix
6. sum=sum+(a1[i][j]\*a2[j][cnt]) //calculating sum
7. res[i][cnt]=sum // storing sum in result matrix
8. **Return**
9. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to multiply two input matrices

void multiply(int(\*arrone)[20],int(\*arrtwo)[20],int(\*res)[20],int rowone,int colone,int rowtwo,int coltwo)

{

int i,j,cnt,sum=0;

for(i=0;i<rowone;i++) //traversing along the rows of first matrix

{

for(cnt=0;cnt<coltwo;cnt++)//multiplying row with all columns

{

sum=0;

for(j=0;j<rowtwo;j++) //traversing columns of second matrix

{

sum=sum+(arrone[i][j]\*arrtwo[j][cnt]);//finding sum

}

res[i][cnt]=sum;//storing result for one pass in res matrix

}

}

}

//function to display the resultant matrix

void dispres(int(\*res)[20],int rowone,int coltwo)

{

int i,j;

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

{

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

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

printf("\n");

}

}

//function to take input in a matrix

void getmat(int(\*arr)[20],int row,int col)

{

int i,j;

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

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

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

}

//function to check if given matrices can be multiplied or not

void validate(int colone,int rowtwo)

{

if(colone!=rowtwo)

{

printf("Given matrices cannot be multiplied....Please try again");

exit(0);

}

}

int main(void)

{

int arrone[20][20],arrtwo[20][20],res[20][20],rowone,colone,rowtwo,coltwo;

printf("Enter the number of rows of the first matrix: ");

scanf("%d",&rowone);

printf("Enter the number of columns of the first matrix: ");

scanf("%d",&colone);

printf("Enter the number of rows of the second matrix: ");

scanf("%d",&rowtwo);

printf("Enter the number of columns of the second matrix: ");

scanf("%d",&coltwo);

validate(colone,rowtwo); //input validation

printf("Enter the elements of the first matrix: \n");

getmat(arrone,rowone,colone);

printf("Enter the elements of the second matrix: \n");

getmat(arrtwo,rowtwo,coltwo);

multiply(arrone,arrtwo,res,rowone,colone,rowtwo,coltwo);

printf("\nRESULT: \n");

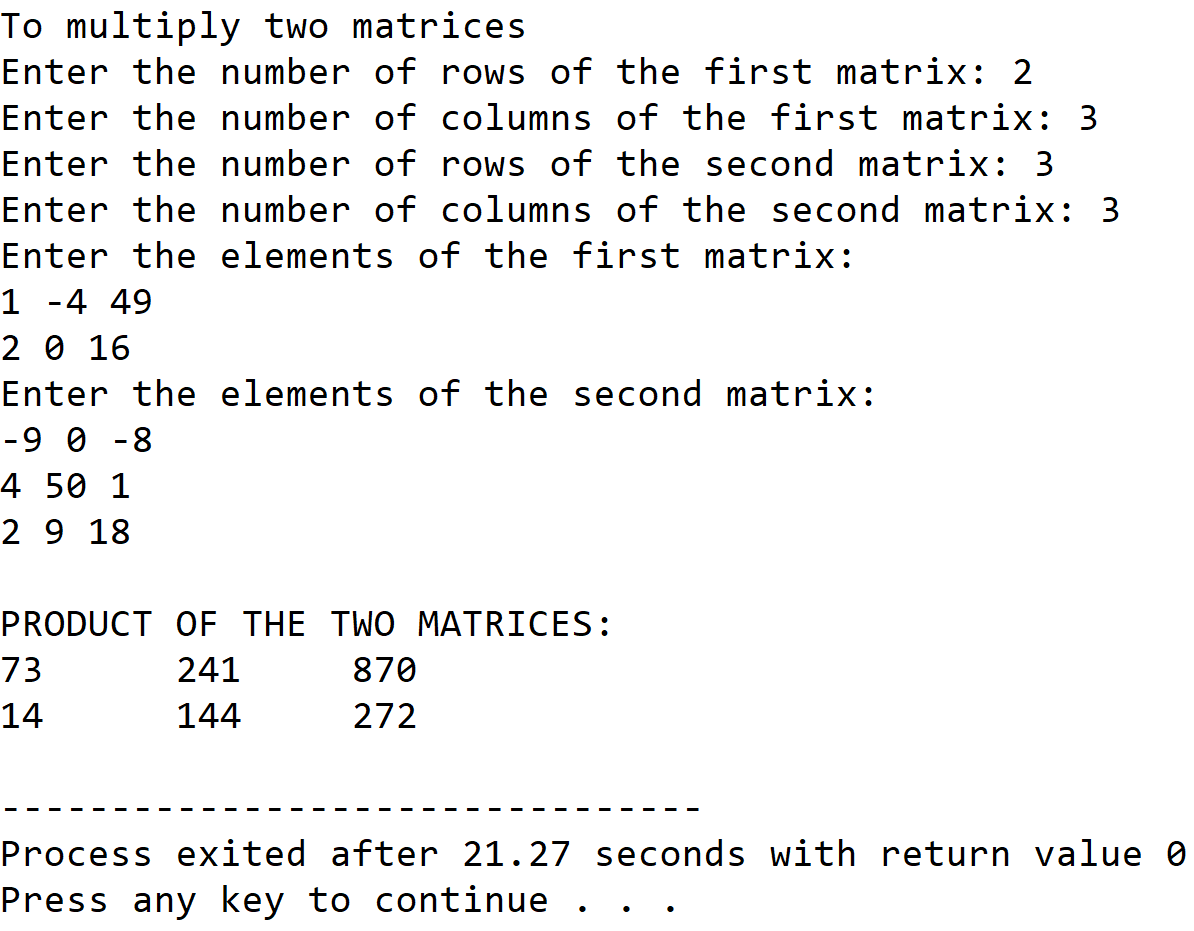
dispres(res,rowone,coltwo);

return 0;

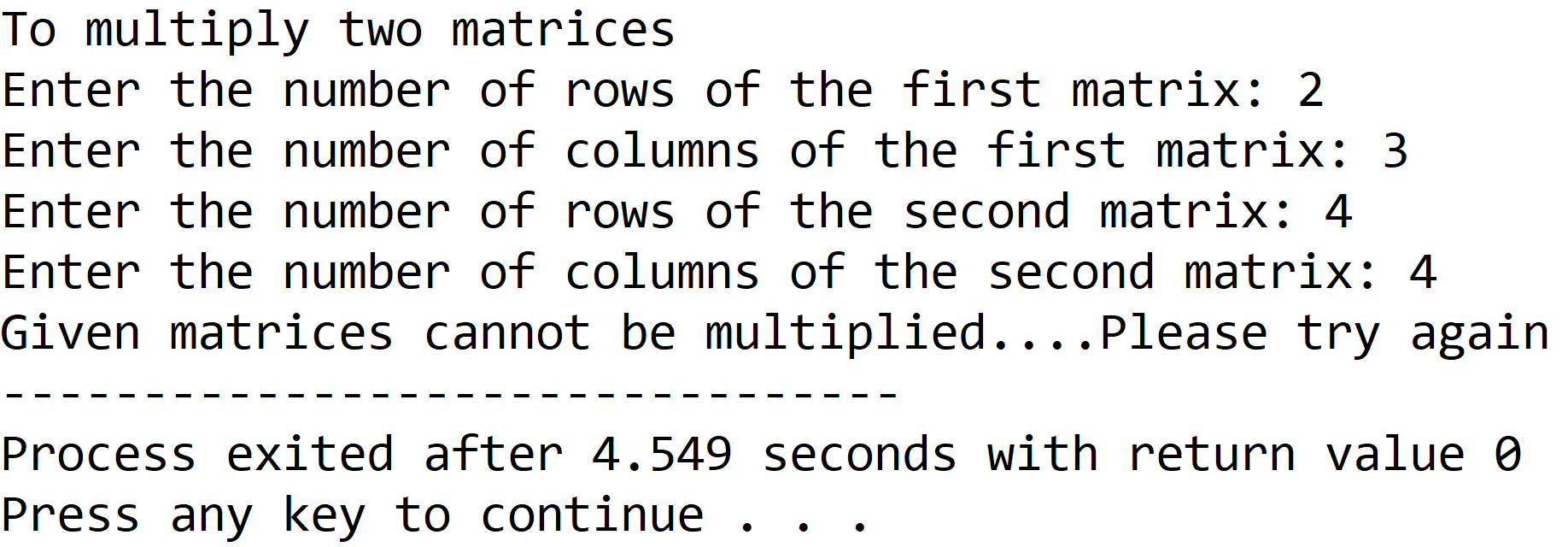
}

**4.OUTPUT**

**SET 1:** Valid Input



**SET 2:** Invalid Input



**5.DISCUSSIONS**

**Variable Description**

* **arrone:** 2-D array to hold first input matrix.
* **arrtwo:** 2-D array to hold second input matrix.
* **res:** 2-D array to hold the result of product of the input matrices.
* **rowone:** number of rows of the first matrix.
* **colone:** number of columns of the first matrix.
* **rowtwo:** number of rows of the second matrix.
* **coltwo:** number of columns of the second matrix.
* **sum:** variable for calculation of product.
* **i,j,cnt:** loop counters.

**Uses**

* The program can be used to calculate the product of any two matrices holding integer values. The program can find application in fields of data science wherever matrix multiplication is needed.

**Limitations**

* The two dimensional arrays used in this program are not dynamically allocated and can lead to either wastage, or lack of required memory locations.
* The product of the two matrices can also be stored in a one dimensional array that requires much less memory than a two dimensional array.

**Future Scope**

* Dynamic memory allocation can be used to construct the arrays.
* The data structure for storing the product can be changed to one dimensional arrays or even linked lists.

**Teacher’s Signature**

**ASSIGNMENT – 14**

**1.PROBLEM STATEMENT**

Write a program in C to dynamically allocate a two dimensional array

**2.ALGORITHMS**

Algorithm **Create\_2D\_Array**

**Input:** The required dimensions ‘row’ and ‘col’ of the two dimensional array.

**Output:** The pointer to the two dimensional array allocate dynamically.

**Remarks:** Dynamic allocation takes place using an array of pointers.

**Steps:**

1. An array of pointers named ‘arr’ with size equal to row, is dyamically allocated in heap
2. **For**(i=1 to row) **do**
3. The pointer arr[i] is pointed to a integer array of size col dynamically allocated in heap
4. The pointer ‘arr’ is returned.
5. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to dynamically allocate a 2-D array using array of pointers

int\*\* creatematrix(int row,int col)

{

int i;

int\*\* arr;

arr=(int\*\*)malloc(row\*sizeof(int\*)); //creating array of pointers of size row

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

arr[i]=(int\*)malloc(col\*sizeof(int));//pointing arr[i] to integer arrays

return arr;

}

//function to take input in a matrix

void getmatrix(int \*\*arr,int row,int col)

{

int i,j;

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

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

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

}

//function to display a matrix

void dispmatrix(int \*\*arr,int row,int col)

{

int i,j;

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

{

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

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

printf("\n");

}

}

int main(void)

{

int\*\* arr;

int row,col;

printf("Enter the number of rows needed: ");

scanf("%d",&row);

printf("Enter the number of columns needed: ");

scanf("%d",&col);

arr=creatematrix(row,col); //dynamic allocation of 2-D array

printf("Enter %d elements: \n",row\*col);

getmatrix(arr,row,col);

printf("Entered elements: \n");

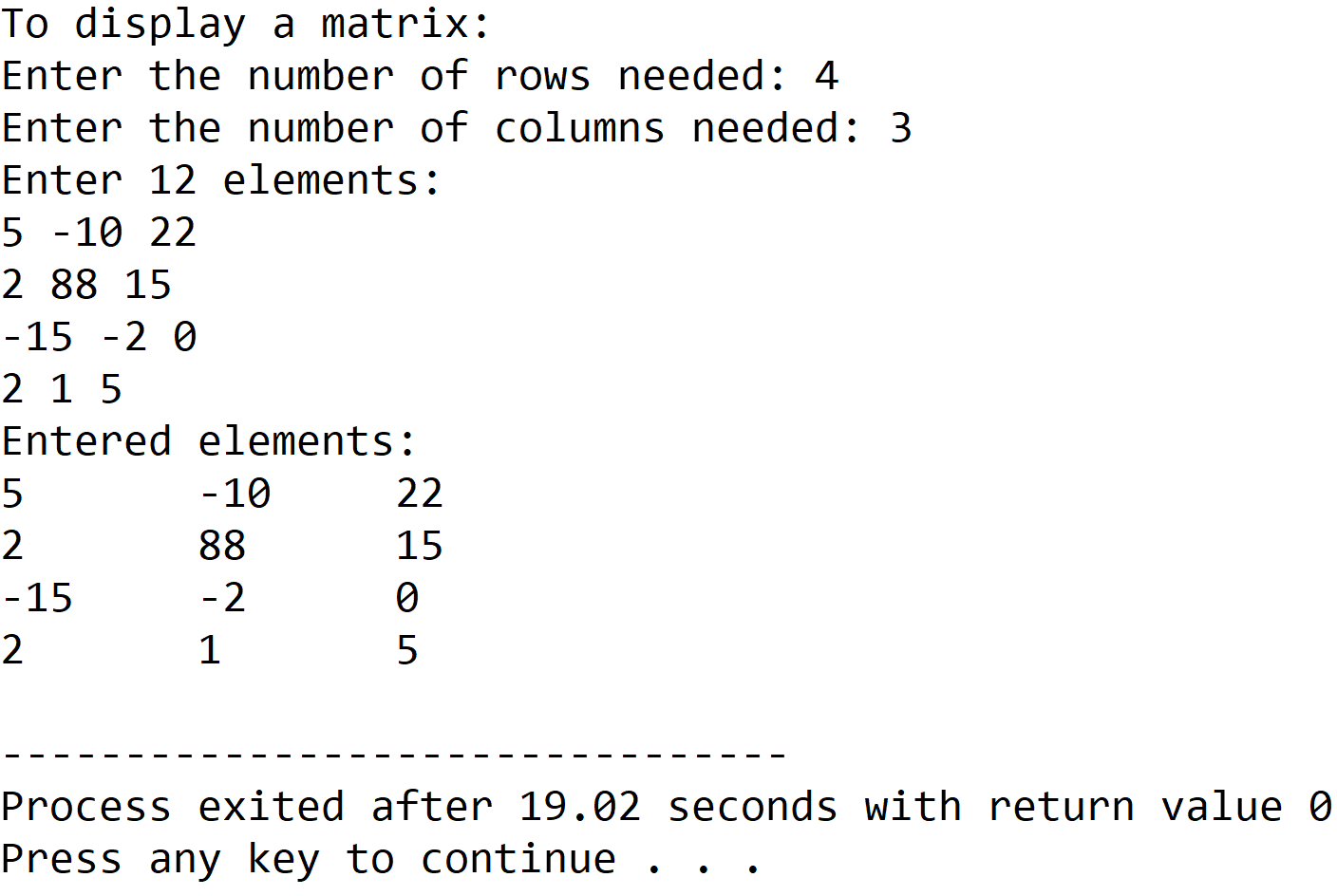
dispmatrix(arr,row,col);

return 0;

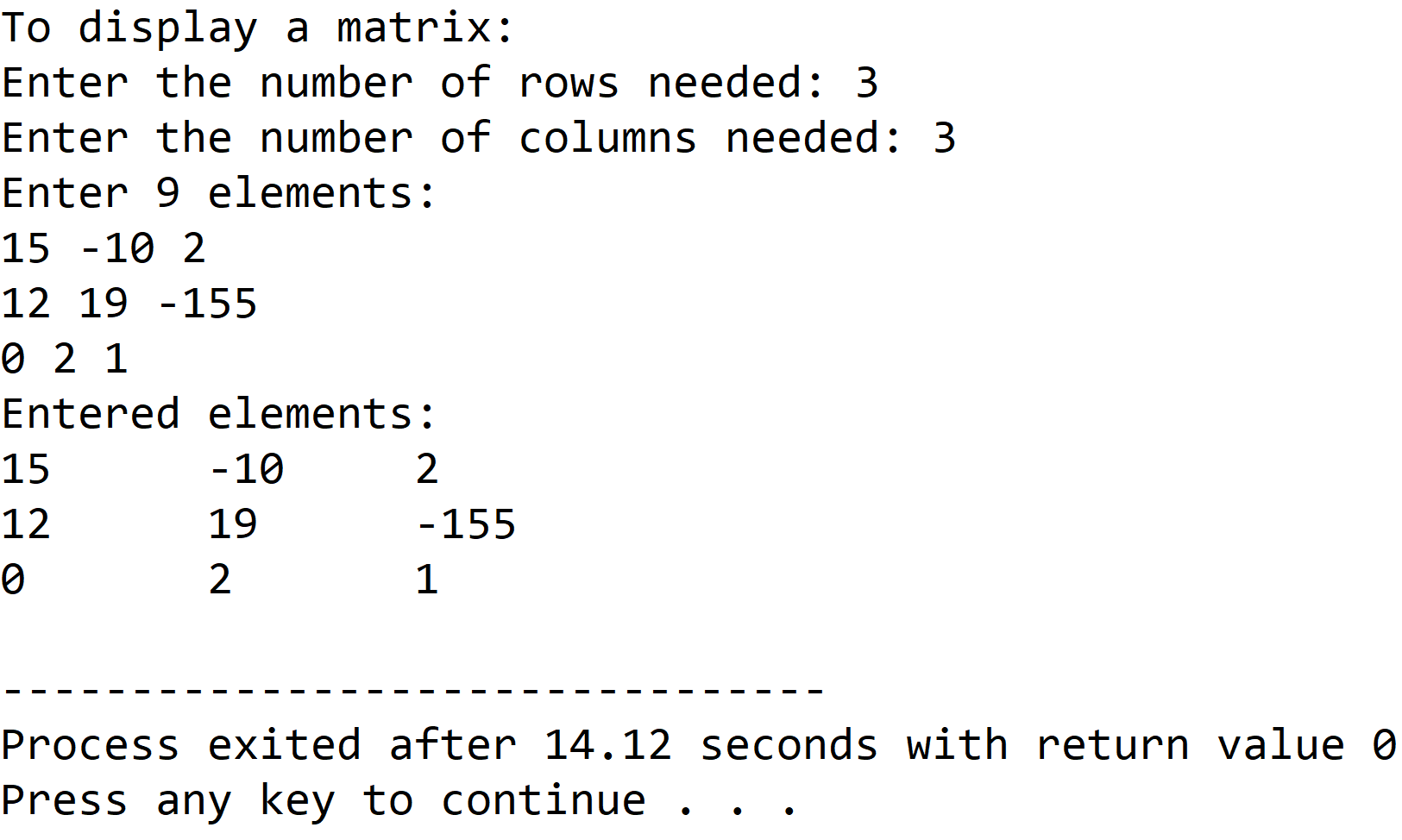
}

**4.OUTPUT**

**SET 1:** Input/Output of a 4x3 matrix



**SET 2 :** Input/Output of a 3x3 matrix



**5.DISCUSSIONS**

**Variable Description:**

* **\*\*arr:** pointer to an integer pointer to hold array of integer pointers.
* **row,col:** dimensions of the 2-D array.
* **i,j:** loop counters.

**Limitations:**

* Even though a dynamic memory allocation procedure has been used in the program, the size of the 2-D array cannot be changed once it is constructed in the memory, thus the array still presents itself as a static data structure.

**Uses:**

* Traditional syntax for creating a 2-D array involves a single 1-D array storing all the elements using a specific indexing scheme. This program shows how a 2-D array can be allocated dynamically with the size being determined at runtime.

**Future Scope:**

* The integer arrays used in the program can be replaced with a linkedlist to eliminate the dependency of the program on contiguous blocks of memory.

**Teacher’s Signature**

**ASSIGNMENT – 15**

**1.PROBLEM STATEMENT**

Write a program in C to implement stack using array.

**2.ALGORITHMS**

Algorithm **Push\_Stack**

**Input:** The pointer to the stack ‘arr’, ‘top’ pointer,size of the stack ‘max’ and the value ‘item’ to be pushed in the stack.

**Output:** The element ‘item’ pushed into the stack at appropriate location.

**Remarks:** ‘top’ indicates the position of the topmost element in the stack and must be passed as a pointer.

**Steps:**

1. **If**(top==max) **then** // if the stack is full
2. **Print** “Stack Overflow”
3. **Exit** //terminate the program
4. **Else**
5. top=top+1 //increment the value of top by one
6. arr[top]=item //insert item at the position of top in stack
7. **EndIf**
8. **Return**
9. **Stop**

Algorithm **Pop\_Stack**

**Input:** The pointer to the stack ‘arr’, ‘top’ pointer.

**Output:** The item at ‘top’ removed from the scope of the stack.

**Remarks:** In array representation of stack, popped elements are not physically erased from the memory.

**Steps:**

1. **If**(top==0) then //if stack is empty
2. **Print** “Stack Underflow
3. **Exit** //terminate the program
4. **Else**
5. top=top-1 //decrement top by one
6. **EndIf**
7. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to display stack

void display(int \*arr, int top)

{

int i;

if(top==-1)

{

printf("Stack Is Empty\n");

return;

}

printf("STACK: ");

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

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

}

//function to push an item into the stack

void push(int \*arr,int\* top,int ele)

{

(\*top)++; // increment top by one

arr[\*top]=ele; // insert the item at the position of top in the stack

printf("PUSHED:%d\n",ele);

display(arr,\*top); //display stack

}

//function to pop an item from the stack

void pop(int\* arr,int \*top)

{

printf("POPPED: %d\n",arr[\*top]);

(\*top)--; //decrement top by one

display(arr,\*top); //display stack

}

int main(void)

{

int max,\*arr,top,ch,ele,i;

printf("Enter the size of the stack needed: ");

scanf("%d",&max);

arr=(int\*)calloc(max,sizeof(int)); //allocating stack in heap

top=-1;

while(1)

{

printf("\n1.PUSH\n2.POP\n3.DISPLAY\n4.Exit");

printf("\nEnter your choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

if(top==max-1) //if stack is full

{

printf("Stack Overflow!");

return 0;

}

else

{

printf("Enter The Element To Be Pushed: ");

scanf("%d",&ele);

push(arr,&top,ele);

}

break;

case 2:

if(top==-1) //if stack is empty

{

printf("Stack Underflow");

return 0;

}

else

pop(arr,&top);

break;

case 3:

display(arr,top);

break;

case 4:

exit(0);

default:

printf("INVALID CHOICE\nPlease Try Again\n");

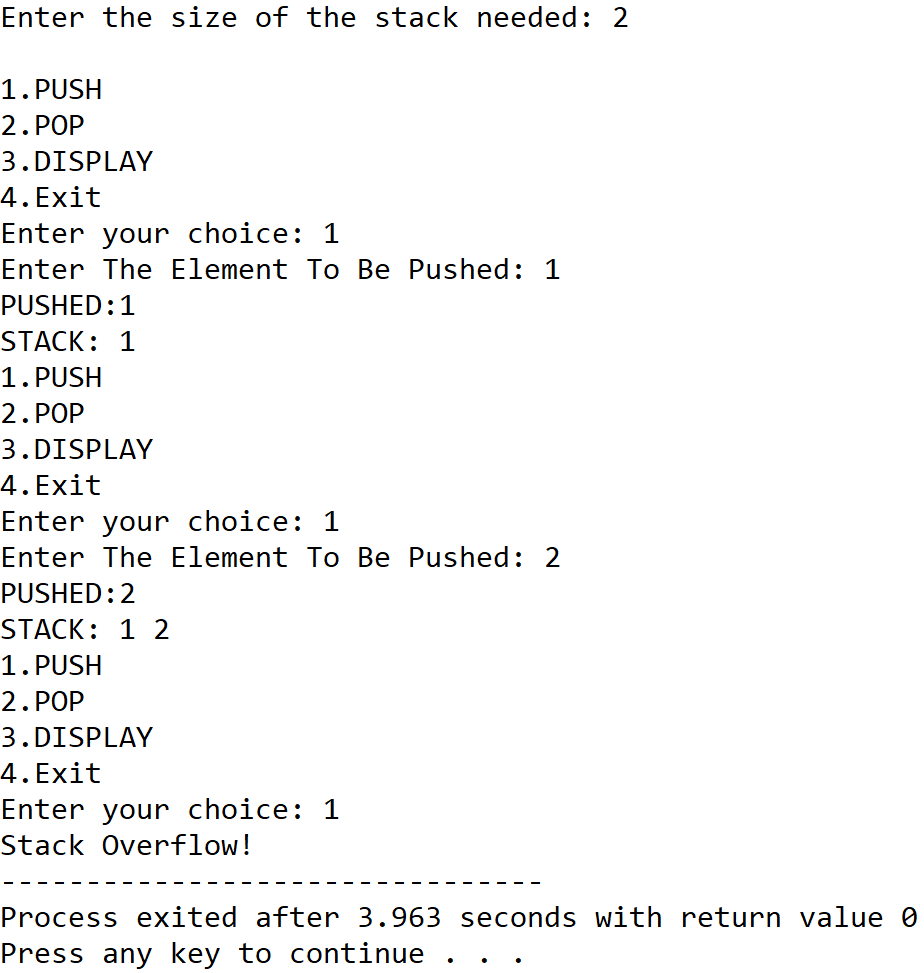
}

}

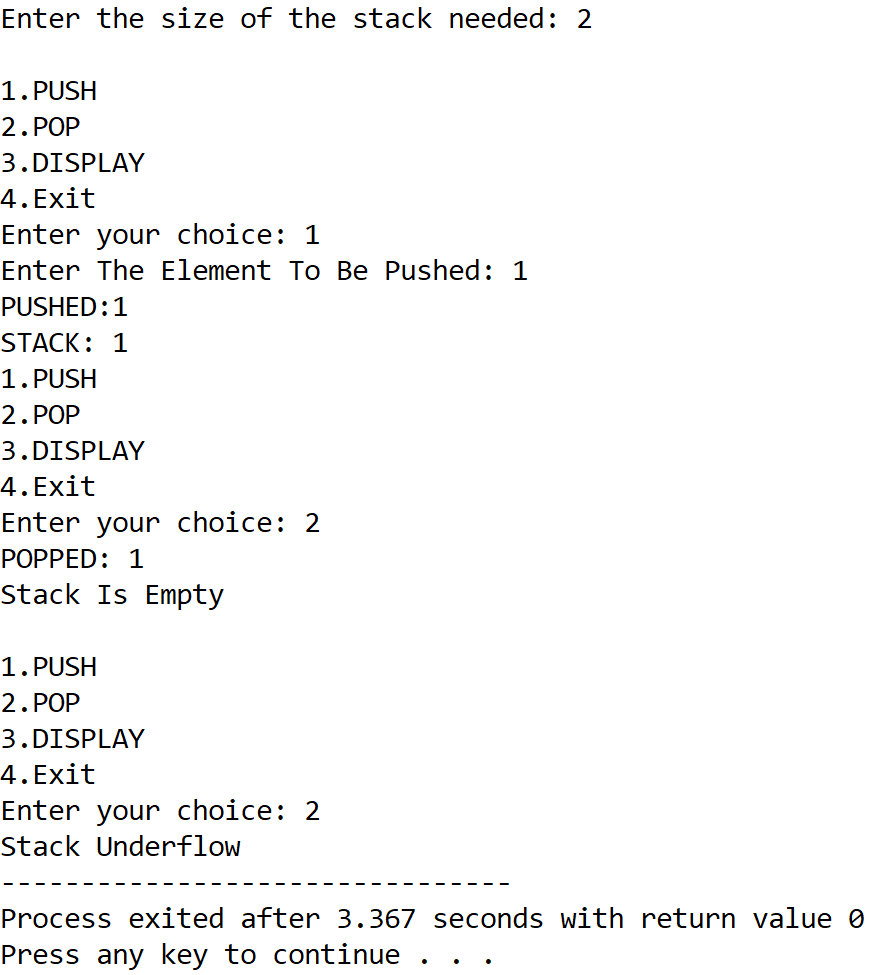
}

**4.OUTPUT**

**SET 1:** Push till Stack Overflow



**SET 2:** Push and Pop till Stack Underflow



**5.DISCUSSIONS**

**Variable Description**

* **\*arr:**pointer to an integer array to represent the stack.
* **max:**Number of elements of the stack.
* **top:** Indicates topmost element of stack.
* **ch:** variable to take user’s choice in switch-case-default.
* **ele:** variable to store variables to push into the stack.
* **i:** loop counter.

**Limitations**

* The program uses array representation of stack, in which stack overflow is very prominent.
* The popped elements are not physically removed from the memory and are just taken out of the scope of the stack, leading to wastage of memory locations.

**Uses**

* Stacks can be used to compute arithmetic expressions.
* Stacks are used by various programming languages to track function calls and store their stackframes.

**Future Scope:**

* The program can be made more memory efficient and the overflow state can be minimized by using a linked list to represent the stack.

**Teacher’s Signature**

**ASSIGNMENT – 16**

**1.PROBLEM STATEMENT**

Write a program in C to implement linear queue using array.

**2.ALGORITHMS**

Algorithm **Insert\_In\_Queue**

**Input:** The pointer to the array ‘arr’ representing the queue, the ‘front’ and ‘rear’ pointers pointing to the first and last element of the queue respectively, and the element ‘item’ to be inserted.

**Output:** The element ‘item’ inserted into the queue at ‘rear’.

**Remarks:** The ‘front’ and ‘rear’ variables must be passed as pointers. When queue is empty, ‘front’ and ‘rear’ are both equal to 0.

**Steps:**

1. rear=rear+1 //increment rear by 1
2. arr[rear]=item //insert item at the position of rear in queue
3. **If**(front=0) **then** //if front is 0,that is during first insertion
4. front=front+1 //increment front by 1
5. **EndIf**
6. **Stop**

Algorithm **Remove\_From\_Queue**

**Input:** The pointer to the array ‘arr’ representing the queue, the ‘front’ and ‘rear’ pointers pointing to the first and last element of the queue respectively.

**Output:** The element at ‘front’ is removed from the queue and is returned.

**Remarks:** The ‘front’ and ‘rear’ variables must be passed as pointers. When queue is empty, ‘front’ and ‘rear’ are both equal to 0.

**Steps:**

1. value=arr[front] //store the element at front in value
2. front=front+1 //increment front by 1
3. **If**(front>rear) **then** //if front exceeds rear,that is queue is empty
4. front=rear=0 //set front and rear to empty state
5. **EndIf**
6. **Return** value
7. **Stop**

**3.SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

//function to display queue

void display(int \*arr, int \*front, int \*rear)

{

int i;

if (\*front > \*rear) //if front exceeds rear

{

printf("Queue is empty");

return;

}

printf("\nQueue Elements: ");

for (i = \*front; i <= \*rear; i++)//traverse from front to rear

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

}

//function to insert an element in queue

void qinsert(int \*arr, int \*front, int \*rear, int item)

{

(\*rear)++; //increment rear by 1

arr[\*rear] = item; //insert item at rear

printf("INSERTED: %d", item);

if (\*front == -1) //for first insertion

(\*front)++; //increment front by 1

display(arr, front, rear);

}

//function to remove an element from queue

int qremove(int \*arr, int \*front, int \*rear)

{

int val;

val = arr[\*front]; //store element at front in value

(\*front)++; //increment front by 1

display(arr, front, rear);

if (\*front > \*rear) //if front exceeds rear,queue becomes empty

\*front = \*rear = -1; //set front,rear to empty state

return val;

}

int main(void)

{

int \*arr, max, ch, front, rear, item;

printf("Enter the size of the queue: ");

scanf("%d", &max);

arr = (int \*)calloc(max, sizeof(int));

front = rear = -1;

while (1)

{

printf("\nMENU:");

printf("\n1.INSERT\n2.REMOVE\n3.DISPLAY\n4.EXIT");

printf("\nEnter Your Choice: ");

scanf("%d", &ch);

switch (ch)

{

case 1:

if (rear == max - 1) //if rear is at the end of the queue

{

printf("Queue Is Full");

break;

}

else

{

printf("Enter the element to be inserted: ");

scanf("%d", &item);

qinsert(arr, &front, &rear, item);

}

break;

case 2:

if (rear == -1) //in empty state, front=rear=-1

{

printf("Queue is already empty...cannot remove");

break;

}

else

{

item = qremove(arr, &front, &rear);

printf("\nREMOVED: %d", item);

}

break;

case 3:

display(arr, &front, &rear);

break;

case 4:

return 0;

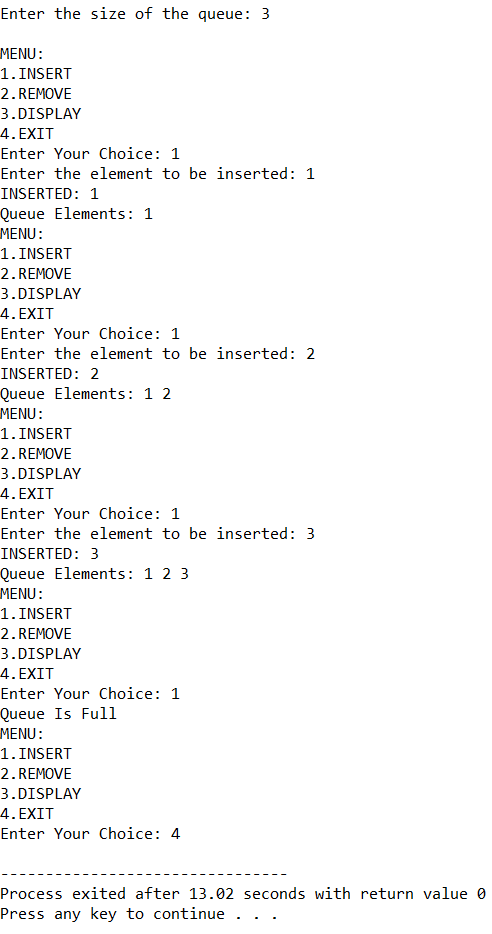
}

}

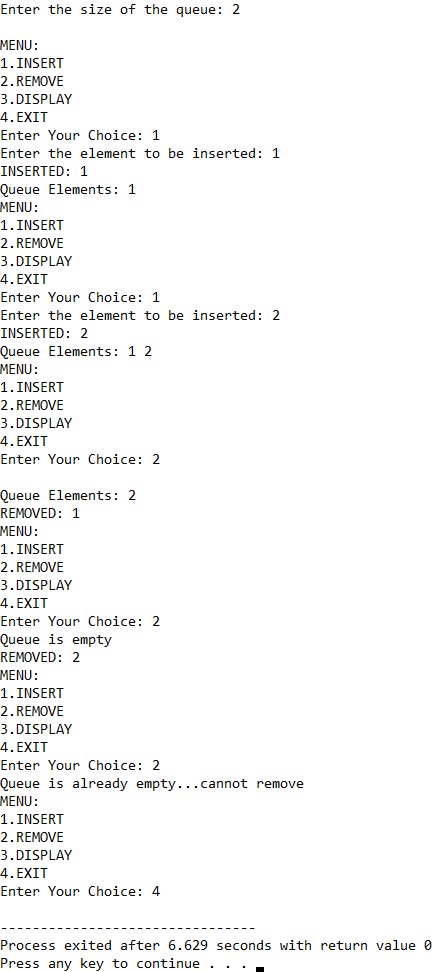
}

**4.OUTPUT**

**SET 1:** Insertion tillqueue full



**SET 2:** Insertion and deletion till queue empty



**5.DISCUSSIONS**

**Variable Description**

* **\*arr:** pointer to the array that represents a queue.
* **Max:** size of the queue.
* **Ch:** variable to take user input in switch-case-default.
* **Front:** pointer to the first element in queue.
* **Rear:** pointer to the las element in queue.
* **Item:** Element to be inserted using Insert\_In\_Queue method.
* **i:** loop counter.

**Uses:**

* In a multiprogramming environment, a CPU has to serve multiple programs, operating systems maintain a ‘priority queue’ in which tasks are pushed with certain priorities, helping the CPU to perform the most important tasks first.
* They can be used to create simulation models of real-life scenarios that involve queueing of data.

**Limitations:**

* The program displayed above generates a linear queue, thus if at one point of time, the front pointer has traversed some distance in the array and there are free memory locations behind it still the queue full state will be called if the rear pointer reaches the end of the queue, thus linear queue implementation is inefficient in terms of memory efficiency.

**Future Scope:**

* The linear queue can be modified into a much more memory effiecient circular queue.
* The array representation can be replace with a linkedlist representation for better memory efficiency.

**Teacher’s Signature**

**ASSIGNMENT – 17**

**1.PROBLEM STATEMENT**

Write a program in C to implement circular queue using array

**2.ALGORITHMS**

Algorithm **Insert\_In\_CircularQueue**

**Input:** The pointer to the array ‘arr’ representing the circular queue, the ‘front’ and ‘rear’ pointers pointing to the element at the start and element at the end of the queue respectively, the element ‘item’ to be inserted into the queue and the size ‘max’ of the queue.

**Output:** The element ‘item’ inserted into the queue at ‘rear’.

**Remarks:** ‘front’ and ‘rear’ must be passed as pointers. An indexing formula x=(x+1) **MOD** max is used to traverse the queue circularly.

**Steps:**

1. **If**(front=rear=0) **then** //when inserting the very first element
2. front=rear=1 //set front and rear to the start of the list
3. arr[rear]=item store item at ‘rear’ position in queue
4. **Return**
5. **Else**
6. rear=(rear+1) **MOD** max //traverse rear using this indexing formula
7. arr[rear]=item //store item at ‘rear’ position in queue
8. **EndIf**
9. **Stop**

Algorithm **Remove\_From\_CircularQueue**

**Input:** The pointer to the array ‘arr’ representing the circular queue, the ‘front’ and ‘rear’ pointers pointing to the element at the start and element at the end of the queue respectively and the size ‘max’ of the queue.

**Output:** The element at front removed from the queue and returned

**Remarks:** ‘front’ and ‘rear’ must be passed as pointers. An indexing formula x=(x+1) **MOD** max is used to traverse the queue circularly.

**Steps:**

1. item=arr[front] //store the item at ‘front’ in item
2. **If**(front=rear) **then** //if front and item are pointing to same location
3. front=rear=0 //set queue to empty state
4. **Return** item
5. **Else**
6. front=(front+1) **MOD** max//traverse front with this indexing formula
7. **EndIf**
8. **Return** item
9. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to insert an element in the queue

void enqueue(int\*arr,int \*front,int\*rear,int item,int max)

{

if(\*front==-1&&\*rear==-1) //for inserting first element

{

\*front=\*rear=0; //set front and rear to the start of queue

arr[\*rear]=item; //store item at position of rear in queue

return;

}

else

{

\*rear=(\*rear+1)%max;//traverse rear with this indexing formula

arr[\*rear]=item; //store item at rear in queue

}

}

//function to remove an element from the queue

int dequeue(int\*arr,int\*front,int\*rear,int max)

{

int item;

item=arr[\*front]; //store the element at front in item

if(\*front==\*rear) //if front and rear point to same location

{

\*front=\*rear=-1; //set queue to empty state

return item;

}

else

\*front=(\*front+1)%max;//traverse front with indexing formula

return item;

}

//function to display a queue

void dispq(int\*arr,int\*front,int\*rear,int max)

{

int i;

i=\*front;

if(\*front==-1&&\*rear==-1) // if queue is in empty state

{

printf("Queue is empty");

return;

}

else

{

printf("QUEUE:");

while(i!=\*rear) //traverse i to rear

{

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

i=(i+1)%max; //increment I using indexing formula

}

printf("%d ",arr[\*rear]);

}

}

int main(void)

{

int\*arr,max,front,rear,ch,item;

printf("Enter the size of the queue :");

scanf("%d",&max);

arr=(int\*)calloc(max,sizeof(int));

front=rear=-1;

while(1)

{

printf("\nMENU\n");

printf("1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.EXIT\n");

printf("Enter your choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

if((rear+1)%max==front) //if rear precedes front

{

printf("QUEUE IS FULL");

break;

}

else

{

printf("Enter the element to be inserted: ");

scanf("%d",&item);

enqueue(arr,&front,&rear,item,max);

printf("INSERTED:%d\n",item);

}

dispq(arr,&front,&rear,max);

break;

case 2:

if(front==-1&&rear==-1) //if queue is empty

{

printf("Queue is already empty");

break;

}

else{

item=dequeue(arr,&front,&rear,max);

printf("DELETED:%d\n",item);

}

dispq(arr,&front,&rear,max);

break;

case 3:

dispq(arr,&front,&rear,max);

break;

case 4:

exit(0);

default:

printf("Wrong Choice....Please Try Again");

}

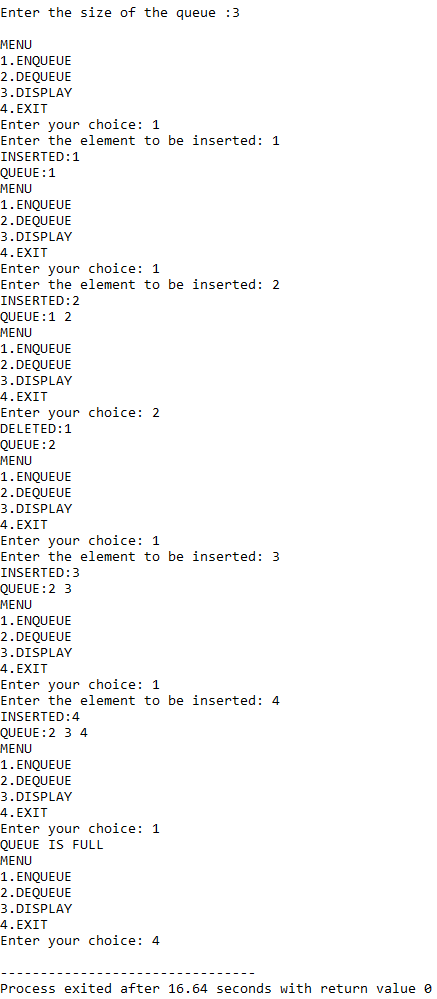
}

return 0;

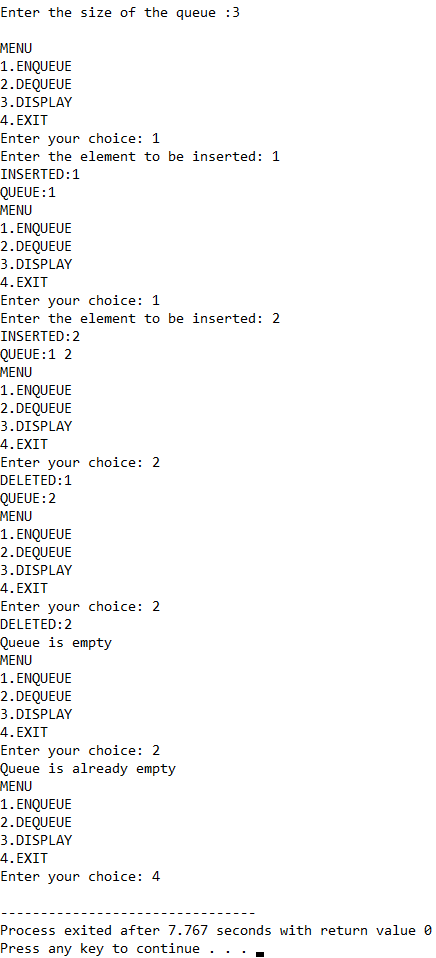
}

**4.OUTPUT**

**SET 1:** Circular insertion in circular queue till queue full

****

**SET 2:** Deletion till queue empty



**5.DISCUSSIONS**

**Variable Description**

* **\*arr:** pointer to hold an array that represents the circular queue.
* **max:** size of the queue.
* **front:** pointer to the element at start in a queue.
* **rear:** pointer to the element at end in a queue.
* **ch:** variable to take user’s input in switch-case-default.
* **item:** variable to store element to be inserted in queue.
* **i:** loop counter.

**Limitations**

* An array has been used to represent the circular queue, since arrays are static data structures, their size cannot be manipulated once they are constructed in the memory, thus they offer less flexibility in size.
* When an element is removed from the queue, it does not get physically erased from the memory rather, it is only removed from the scope of the queue, thus this also leads to inefficient use of memory.

**Uses**

* Circular queues are more memory efficient as compared to linear queues, since in circular queues, if the ‘rear’ pointer reaches the end of the queue, it will check if any space is available at the beginning of the list, and it there is , then the ‘rear’ pointer jumps to the beginning of the queue to utilize the available memory and does not throw a “queue full” message. This is possible due to the mapping formula rear=(rear+1) **MOD** max.

**Future Scope**

* The array representation can be replaced with a linkedlist representation, ending the dependency of the program on contiguous memory locations. It will also allow more flexibility in manipulation of the size.

**Teacher’s Signature**

**ASSIGNMENT – 18**

**1.PROBLEM STATEMENT**

Write a program in C to scan polynomial using array. Implement addition, subtraction, multiplication of two polynomials.

**2.ALGORITHMS**

Algorithm **Add\_Polynomial**

**Input:** Pointers ‘poly1’ and ‘poly2’ that point to the array holding the first and second polynomial respectively with their respective lengths ‘size1’ and ‘size2’.

**Output:** The two polynomials added together and stored in an array named ‘res’, whose pointer is returned.

**Remarks:** Each element of the polynomial array has two members namely ‘coeff’ and ‘pow’ representing the coeffiecent and exponent of each term respectively.

**Steps:**

1. A polynomial array named ‘res’ is allocated with size equal to the sum of size1 and size2.
2. i=0,j=0,k=0 //initialising loop counters
3. **While**(i<size1 **AND** j<size2) **do** //Traversing both lists until one ends
4. **If**(pow🡨poly1[i]==pow🡨poly2[j]) **then**//if pow of poly1 item is equal
5. //assign sum of coefficients of poly1 and poly2 to res
6. coeff🡨res[k] = (coeff🡨poly1[i]) + (coeff🡨poly2[j])
7. pow🡨res[k] = pow🡨poly1[i] //assign pow of poly1 term to res term
8. i=i+1,j=j+1,k=k+1 //increment i,j,k by 1
9. **Else**
10. **If**(pow🡨poly1[i]>pow🡨poly2[j]) **then**//pow of poly2 term is greater
11. coeff🡨res[k] = coeff🡨poly1[i]//coeff of poly1 = coeff of res
12. pow🡨res[k] = pow🡨poly1[i]//pow of poly1 = pow of res
13. i=i+1,k=k+1 //increment i,k by 1
14. **Else**
15. coeff🡨res[k] = coeff🡨poly2[j]//coeff of poly2 = coeff of res
16. pow🡨res[k] = pow🡨poly2[j]//pow of poly2 = pow of res
17. j++;k++;//increment j,k by 1
18. **EndIf**
19. **EndIf**
20. **While**(i<size) **do** //while poly1 does not end
21. coeff🡨res[k]= coeff🡨poly1[i]//coeff of poly1 = coeff of res
22. pow🡨res[k]= pow🡨poly1[i] //pow of poly1 = pow of res
23. i=1+1,k=k+1 //increment i,k by 1
24. **EndWhile**
25. **While**(j<size2) **do** //while poly2 does not end
26. coeff🡨res[k]= coeff🡨poly2[j] //coeff of poly2 = coeff of res
27. pow🡨res[k]= pow🡨poly2[j] //pow of poly2 = pow of res
28. j=j+1;k=k+1; //increment j,k by 1
29. **EndWhile**
30. **Return** res

Algorithm **Subtract\_Polynomial**

**Input:** Pointers ‘poly1’ and ‘poly2’ that point to the array holding the first and second polynomial respectively with their respective lengths ‘size1’ and ‘size2’.

**Output:** The difference of the two polynomials stored in an array named ‘res’, whose pointer is returned.

**Remarks:** Each element of the polynomial array has two members namely ‘coeff’ and ‘pow’ representing the coeffiecent and exponent of each term respectively.

**Steps:**

1. A polynomial array named ‘res’ is allocated with size equal to the sum of size1 and size2.
2. i=0,j=0,k=0 //initialising loop counters
3. **While**(i<size1 **AND** j<size2) **do** //Traversing both lists until one ends
4. **If**(pow🡨poly1[i]==pow🡨poly2[j]) **then**//if pow of poly1 item is equal
5. //assign sum of coefficients of poly1 and poly2 to res
6. coeff🡨res[k] = (coeff🡨poly1[i]) - (coeff🡨poly2[j])
7. pow🡨res[k] = pow🡨poly1[i] //assign pow of poly1 term to res term
8. i=i+1,j=j+1,k=k+1 //increment i,j,k by 1
9. **Else**
10. **If**(pow🡨poly1[i]>pow🡨poly2[j]) **then**//pow of poly2 term is greater
11. coeff🡨res[k] = coeff🡨poly1[i]//coeff of poly1 = coeff of res
12. pow🡨res[k] = pow🡨poly1[i]//pow of poly1 = pow of res
13. i=i+1,k=k+1 //increment i,k by 1
14. **Else**
15. coeff🡨res[k] = coeff🡨poly2[j]//coeff of poly2 = coeff of res
16. pow🡨res[k] = pow🡨poly2[j]//pow of poly2 = pow of res
17. j++;k++;//increment j,k by 1
18. **EndIf**
19. **EndIf**
20. **While**(i<size) **do** //while poly1 does not end
21. coeff🡨res[k]= coeff🡨poly1[i]//coeff of poly1 = coeff of res
22. pow🡨res[k]= pow🡨poly1[i] //pow of poly1 = pow of res
23. i=1+1,k=k+1 //increment i,k by 1
24. **EndWhile**
25. **While**(j<size2) **do** //while poly2 does not end
26. coeff🡨res[k]= coeff🡨poly2[j] //coeff of poly2 = coeff of res
27. pow🡨res[k]= pow🡨poly2[j] //pow of poly2 = pow of res
28. j=j+1;k=k+1; //increment j,k by 1
29. **EndWhile**
30. **Return** res

Algorithm **Multiply\_Polynomial**

**Input:** Pointers ‘poly1’ and ‘poly2’ that point to the array holding the first and second polynomial respectively with their respective lengths ‘size1’ and ‘size2’.

**Output:** The product of the two polynomials stored in a polynomial array named ‘res’ whose pointer is returned.

**Remarks:** The product polynomial is stored without sorting it in decending order of the exponent of the terms.

**Steps:**

1. A polynomial array named ‘res’ is allocated with size equal to the product of size1 and size2.
2. i=j=k=0 // initialising loop counters
3. **For**(i=0 to size1) **do** //traversing poly1 till end
4. **For**(j=0 to size2) **do** //traversing poly2 till end
5. coeff🡨res[k]=(coeff🡨poly1[i])\*(coeff🡨poly2[j]//storing product
6. pow🡨res[k]= (pow🡨poly1[i])+(pow-<poly2[j])//storing exponent
7. k++ //increment k by 1
8. **EndFor**
9. **EndFor**
10. **Return** res

**3. SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//structure to create an array for holding polynomials

typedef struct term

{

int coeff;

int pow;

}term;

int count=0;

//function to take input in a polynomial array

void getpoly(term\* poly,int len)

{

int i;

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

{

printf("Enter Coefficient: ");

scanf("%d",&poly[i].coeff);

printf("Enter Exponent: ");

scanf("%d",&poly[i].pow);

}

}

//function to display a polynomial array

void dispoly(term \*poly,int len)

{

int i;

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

if(poly[i].coeff!=0)

printf("%+dx^%d ",poly[i].coeff,poly[i].pow);

}

//function to add two polynomials together

term\* poly\_add(term \*poly1,int size1,term \*poly2,int size2)

{

term \*res=(term\*)malloc((size1+size2)\*sizeof(term));//allocate res array

int i=0,j=0,k=0; //set loop counters

while(i<size1 && j<size2) //while both arrays are not exhausted

{

if(poly1[i].pow==poly2[j].pow) //if powers of both terms are equal

{

res[k].coeff = (poly1[i].coeff) + (poly2[j].coeff);//add terms

res[k].pow = poly1[i].pow;//set power of res term to poly1 term

i++;j++;k++; //increment i,j,k by one

}

else

{

if(poly1[i].pow>poly2[j].pow) //if power of poly1 term is greater

{

res[k].coeff = poly1[i].coeff; //set res coeff to poly1 coeff

res[k].pow = poly1[i].pow;//set res pow to poly1 pow

i++;k++;//increment i,j by one

}

else

{

res[k].coeff = poly2[j].coeff; //set res coeff to poly2 coeff

res[k].pow = poly2[j].pow;//set res pow to poly1 pow

j++;k++; //increment j,k by one

}

}

}

while(i<size1) //if poly1 remains

{

res[k].coeff=poly1[i].coeff; //set res coeff to poly1 coeff

res[k].pow=poly1[i].pow; //set res pow to poly1 pow

i++;k++; //increment i,k by one

}

while(j<size2) //if poly2 remains

{

res[k].coeff=poly2[j].coeff; //set res coeff to poly1 coeff

res[k].pow=poly2[j].pow; //set res pow to poly1 pow

j++;k++;//increment j,k by one

}

count=k;

return res;

}

term\* poly\_sub(term \*poly1,int size1,term \*poly2,int size2)

{

term \*res=(term\*)malloc((size1+size2)\*sizeof(term));//allocate res array

int i=0,j=0,k=0; //set loop counters

while(i<size1 && j<size2) //while both arrays are not exhausted

{

if(poly1[i].pow==poly2[j].pow) //if powers of both terms are equal

{

res[k].coeff = (poly1[i].coeff) - (poly2[j].coeff);//add terms

res[k].pow = poly1[i].pow;//set power of res term to poly1 term

i++;j++;k++; //increment i,j,k by one

}

else

{

if(poly1[i].pow>poly2[j].pow) //if power of poly1 term is greater

{

res[k].coeff = poly1[i].coeff; //set res coeff to poly1 coeff

res[k].pow = poly1[i].pow;//set res pow to poly1 pow

i++;k++;//increment i,j by one

}

else

{

res[k].coeff = poly2[j].coeff; //set res coeff to poly2 coeff

res[k].pow = poly2[j].pow;//set res pow to poly1 pow

j++;k++; //increment j,k by one

}

}

}

while(i<size1) //if poly1 remains

{

res[k].coeff=poly1[i].coeff; //set res coeff to poly1 coeff

res[k].pow=poly1[i].pow; //set res pow to poly1 pow

i++;k++; //increment i,k by one

}

while(j<size2) //if poly2 remains

{

res[k].coeff=poly2[j].coeff; //set res coeff to poly1 coeff

res[k].pow=poly2[j].pow; //set res pow to poly1 pow

j++;k++;//increment j,k by one

}

count=k;

return res;

}

term\* poly\_mult(term \*poly1,int size1,term \*poly2,int size2)

{

term \*res=(term\*)malloc((size1\*size2)\*sizeof(term));//allocate res array

int i,j,k=0;//set loop counters

for(i=0;i<size1;i++) //traversing poly1

for(j=0;j<size2;j++)//traversing poly2

{

res[k].coeff=(poly1[i].coeff)\*(poly2[j].coeff);

res[k].pow=(poly1[i].pow)+(poly2[j].pow);

k++;

}

count=k;

return res;

}

//function for input validation

void validate(term \*poly,int size)

{

int i;

for(i=0;i<size-1;i++)

{

if(poly[i].pow<poly[i+1].pow)

{

printf("The terms must be in decending order of their exponents\nPlease Retry");

exit(1);

}

}

}

int main(void)

{

int size1,size2,ch;

term \*poly1,\*poly2,\*res;

printf("To perfrom arithmetic operations on two polynomials \n");

printf("Enter the number of terms in first polynomial: ");

scanf("%d",&size1);

printf("Enter the number of terms in second polynomial: ");

scanf("%d",&size2);

poly1=(term\*)malloc(size1\*sizeof(term));

poly2=(term\*)malloc(size2\*sizeof(term));

printf("\nPlease enter terms in decending order of their exponents: \n");

printf("For the first polynomial: \n");

getpoly(poly1,size1);

validate(poly1,size1);

printf("For the second polynomial: \n");

getpoly(poly2,size2);

validate(poly2,size2);

printf("First polynomial: ");

dispoly(poly1,size1);

printf("\nSecond polynomial: ");

dispoly(poly2,size2);

printf("\nMENU:\n");

printf("1.Addition\n2.Subtraction\n3.Multiplication");

printf("\nEnter your choice:");

scanf("%d",&ch);

switch(ch)

{

case 1:

res=poly\_add(poly1,size1,poly2,size2);

printf("SUM: ");

dispoly(res,count);

break;

case 2:

res=poly\_sub(poly1,size1,poly2,size2);

printf("DIFFERENCE: ");

dispoly(res,count);

break;

case 3:

res=poly\_mult(poly1,size1,poly2,size2);

printf("PRODUCT: ");

dispoly(res,count);

break;

free(poly1);free(poly2);free(res);

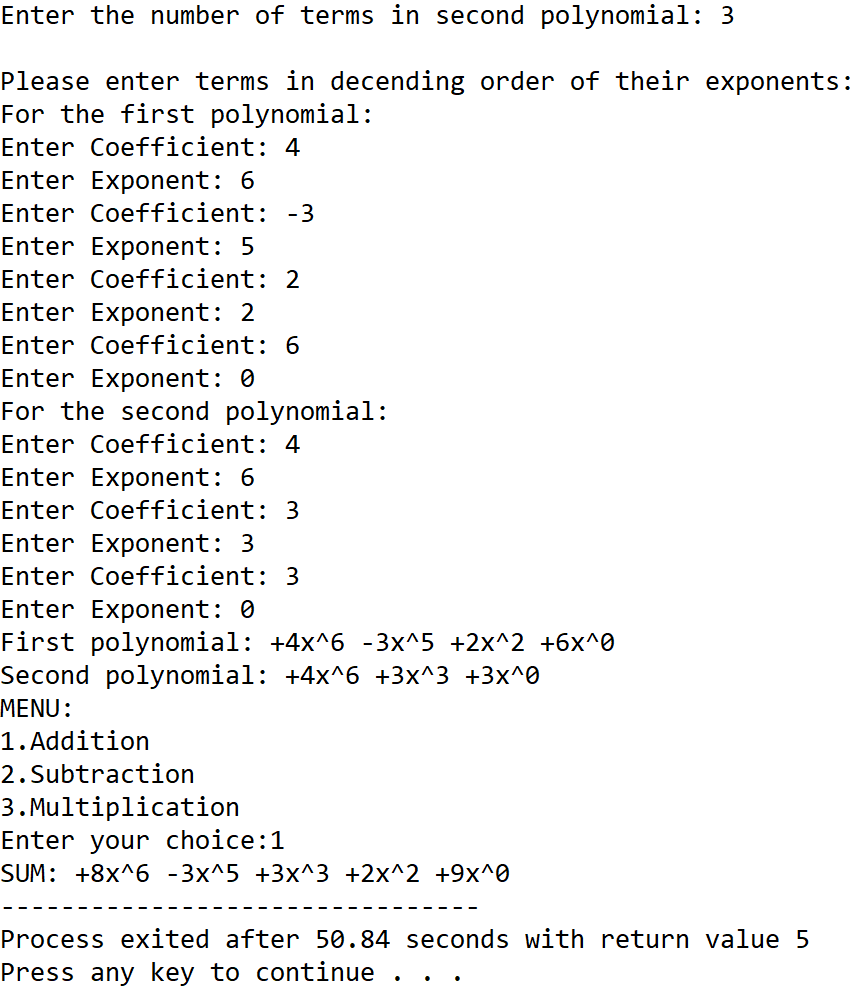
return 0;

}

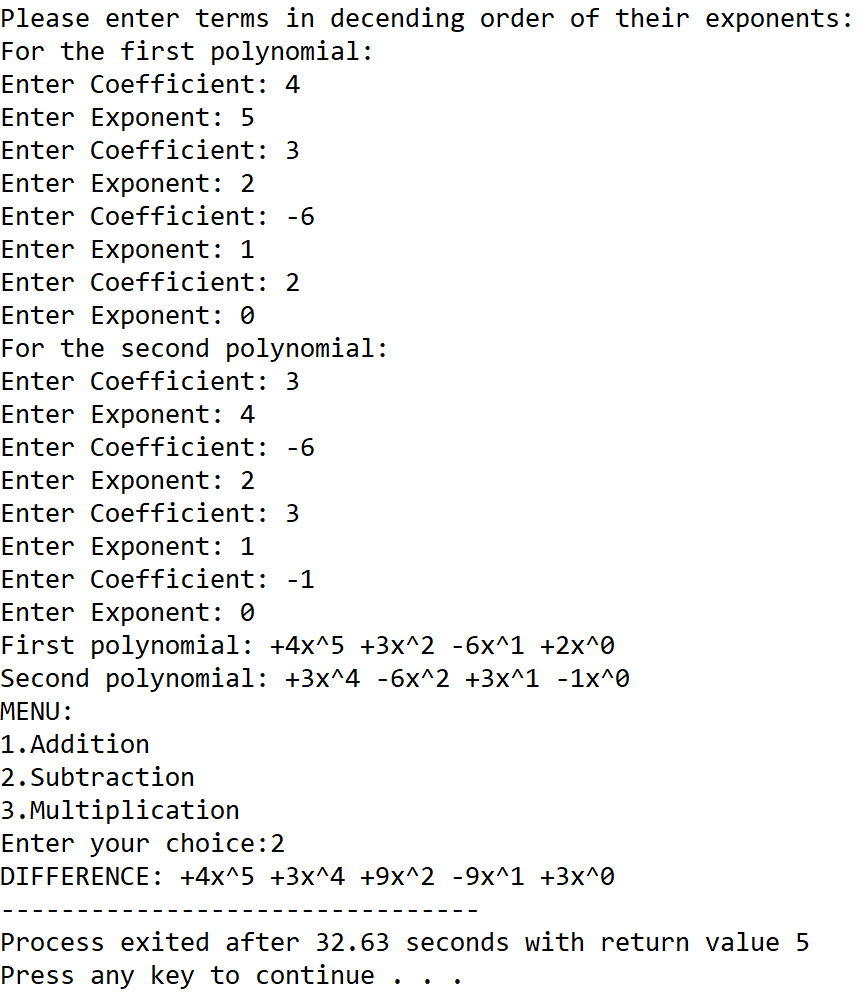
}

**4.OUTPUT**

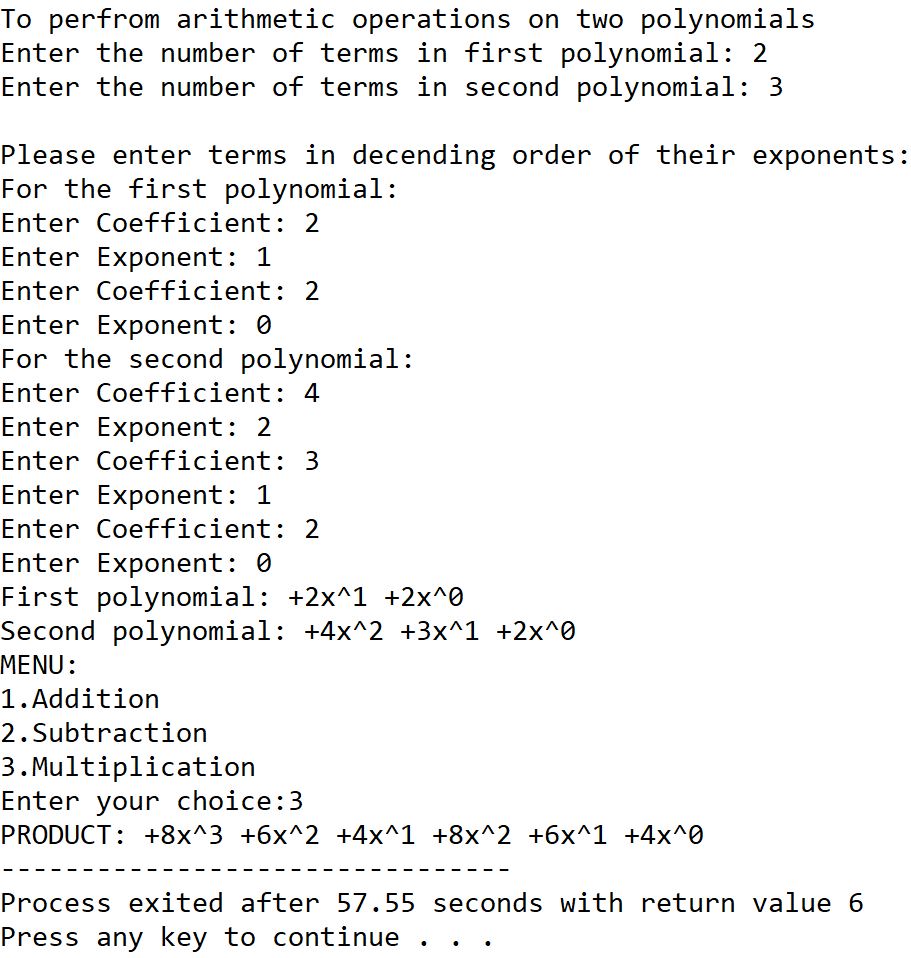
**SET 1:** Addition of two polynomials

****

**SET 2:** Subtraction of two polynomials

****

**SET 3:** Multiplication of two polynomials



**5.DISCUSSIONS**

**Variable Description**

* **size1:** Number of terms in first polynomial.
* **size2:** Number of terms in second polynomial.
* **ch:** variable to take user input in switch-case-default.
* **\*poly1:** pointer to the array of structures holding the first polynomial.
* **\*poly2:** pointer to the array of structures holding the second polynomial.
* **\*res:** pointer to the array of structures holding the result polynomial after an operation is performed.
* **count:** global variable to track the size of res array
* **i,j,k:** loop counters

**Limitations**

* The program needs the user to enter the polynomials with the exponents sorted in decending order since it applies the concept of merging of two sorted lists into a single list.
* The program can handle polynomials with one variable only.
* The program uses an array of structures which is a static data structure and whose size cannot be manipulated once it is constructed in the memory.

**Uses**

* The program can be used to perfrom addition,subtraction and multiplication of two polynomials having one variable.The program can find implementation in a calculator application.

**Future Scope:**

* The logic of the program can be changed to ask the uses the number of variables present in the polynomial and the operations are done accordingly.
* The data stucture used in the program can be changed to linkedlists making it more memory efficient and removing dependency on contiguous memory locations.

**Teacher’s Signature**

**ASSIGNMENT – 19**

**PROBLEM STATEMENT**

Write a program in C to implement evaluation of postfix expressions.

**ALGORITHMS**

Algorithm **Evaluate**

**Input:** The input array holding the postfix expression to be evaluated

**Output:** Result of the input postfix expression.

**Remarks:** It is assumed that the input expression is valid

**Steps;**

1. i=0,top=0
2. **While** end of input expression is not reached, **repeat steps 3 to 13**
3. **If** the element at i is an operand **then**
4. **While** a blank space is not encountered **repeat steps 5 to 6**
5. **Push**(stack,operand,top)
6. i=i+1
7. **Else If** the element at i is an operator **then**
8. **While** a blank space is not encountered **repeat steps 9 to 13**
9. y=**Pop**(stack,top) //pop second operand
10. x=**Pop**(stack,top) //pop first operand
11. res=**solve**(x,operator,y)//solve present operator and operands
12. **Push**(stack,top,res) //push result back into the stack
13. i=i+1 //increment i by one
14. res=**Pop**(stack,top) //pop final result from stack
15. **Return** res
16. **Stop**

Algorithm **Push**

**Input:** The stack in which elements are to be pushed, the value of top indicator and the element ‘item’ to be inserted

**Output:** The element ‘item’ inserted into the stack at appropriate position.

**Remarks:** The value of top must be passed as pointer

**Steps:**

1. top=top+1 //increment value of top by one
2. stack[top]=item //insert item at top of stack
3. **Return**
4. **Stop**

Algorithm **Pop**

**Input:** The stack from which elements are to be popped and it’s top indicator.

**Output:** The element at top removed from the stack and returned.

**Remarks:** top must be passed in as a pointer

**Steps:**

1. ele=stack[top] //stop element at top in ele
2. top=top-1 //decrement top by one
3. **Return** ele
4. **Stop**

Algorithm **Solve**

**Input:** Two operands x and y along with the operator opr

**Output:** The operands evaluated as per (x “operator” y)

**Remarks:** The supported operations are addition,subtraction and multiplication

**Steps:**

1. **If**(operator= \* ) **then** //for multiplication
2. res=x\*y
3. **Else If**(operator= / ) **then** //for division
4. **If**(y=0) **then**
5. Print “Divide by zero error”
6. **Exit**
7. **EndIf**
8. **Else If**(operator= + ) **then** //for addition
9. res=x+y
10. **Else** //for subtraction
11. res=x-y
12. **EndIf**
13. **Return** res
14. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

//function to solve for present operator and operands

int solve(int x,char opr,int y)

{

    int res;

    if(opr=='\*')

        res = x\*y;

    else if(opr=='/')

    {

        if(y==0)

        {

            printf("Divide by zero error...please retry");

            exit(1);

        }

        res = x/y;

    }

    else if(opr=='+')

        res = x+y;

    else

        res = x-y;

    return res;

}

//function to push elements into stack

void push(int \*stack,int \*top,int opr)

{

    (\*top)++;

    stack[\*top]=opr;

}

//function to pop elements from stack

int pop(int \*stack,int \*top)

{

    int ele;

    if(\*top==-1)

        printf("Empty Stack");

    else

    {

        ele=stack[\*top];

        (\*top)--;

    }

    return ele;

}

//function to evaluate a given postfix expression

int evaluate(char \*expr)

{

    int i=0,count=0,opr=0,x,y,res,stack[100],top=-1;

    while(expr[i]!='\0') //while the end of string is not reached

    {

        if(expr[i]>47 && expr[i]<58) //if element at i is an operand

        {

            while(expr[i]!=' ') //while a space is not encountered

            {

                opr=(opr\*pow(10,count))+(expr[i]-48); //calculate operand

                count++;i++;

            }

            push(stack,&top,opr); //push operand into stack

            opr=0;count=0;//reset opr and count to zero for next iteration

        }

        else //an operator is encountered

        {

            while(expr[i]!=' ' && expr[i]!='\0')

            {

                if(expr[i]>41 && expr[i]<48) //if element at i is operator

                {

                    y=pop(stack,&top); //pop second operand

                    x=pop(stack,&top); //pop first operand

                    res=solve(x,expr[i],y); //solve

                    push(stack,&top,res); //push result into stack

                    i++;

                }

            }

        }

        if(expr[i]=='\0') //if end of the string is reached

            break;

        else

            i++;

    }

    res=pop(stack,&top); //pop final result from the stack

    return res;

}

int main(void)

{

    char expr[100];

    int res;

    printf("Enter the postfix expression\nPlease separate adjacent numbers and operators using spaces:\n");

    gets(expr);

    printf("Entered expression: ");

    puts(expr);

    res=evaluate(expr);

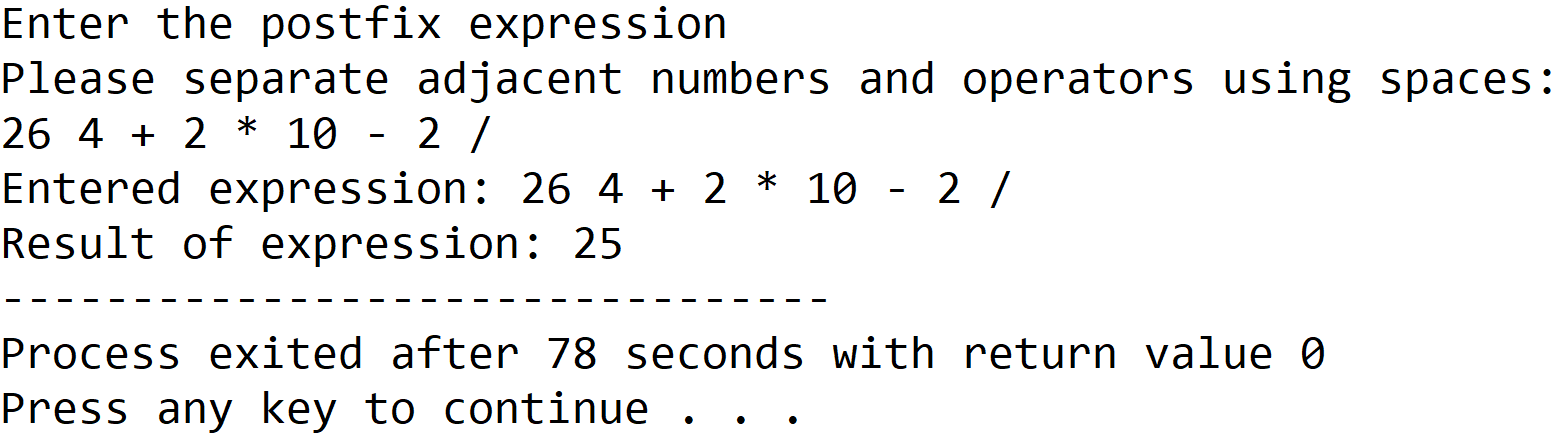
    printf("Result of expression: %d",res);

    return 0;

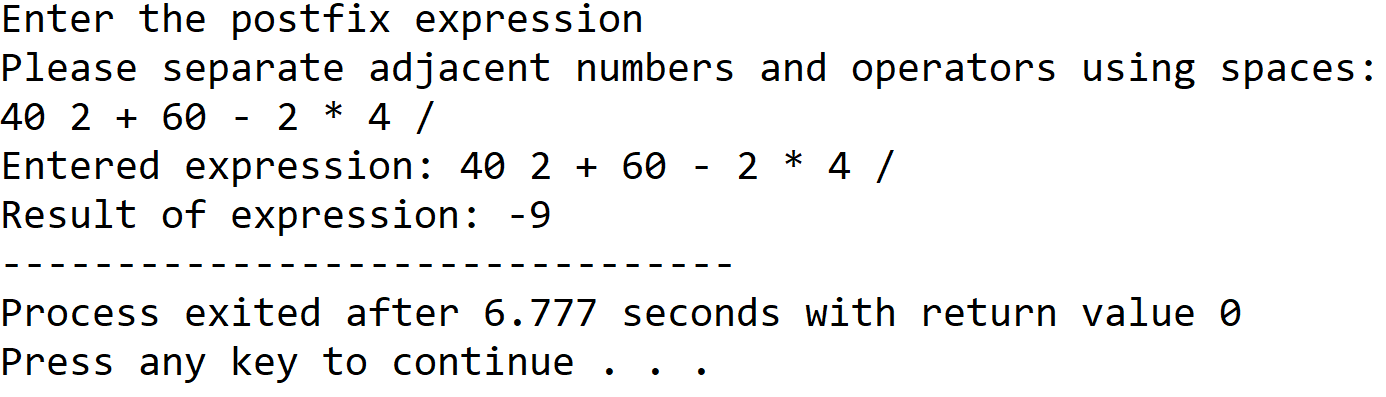
}

**4.OUTPUT**

**SET 1:** Positive integer as result

****

**SET 2:** Negative integer as result



**5.DISCUSSIONS**

**Variable Description**

* **expr:** string to hold the user input postfix expression.
* **res:** To store result of a calculation.
* **Opr:** To calculate value of operand.
* **x,y:** To hold values of operands.
* **Stack:** An integer array representing a stack.
* **Top:** pointer to the topmost element in the stack.
* **i,count:** loop counter.

**Limitations**

* It is assumed that the user enters a valid postfix expression and the if the input postfix expression is invalid, undesired output will be received.
* Arrays have been used to hold the postfix expression and also to represent the stack, since they are static data structures, their size cannot be manipulated once they are constructed in the memory.

**Uses**

* The program can be used to evaluate a postfix expression with integer operands and the above mentioned operations, the program can find implementation in a calculator application.

**Future Scope**

* The data structures can be replaced from arrays to linked lists making the program more memory effiecient and terminating the dependency on contiguous memory locations.

**Teacher’s Signature**

**ASSIGNMENT – 20**

**1.PROBLEM STATEMENT**

Write a program in C to implement singly linked list.

**2.ALGORITHMS**

Algorithm **Create\_Node**

**Input:** The item to be stored in a node

**Output:** The address to a node constructed in memory with item stored in its data part

**Remarks:** The node allocated has two sections: a data section to store integer data and a link section to reference another node in memory.

**Steps:**

1. A node is allocated in memory and it’s address is stored in a pointer ptr
2. Set the link part to a null value
3. Set the data part to the input item
4. **Return** ptr
5. **Stop**

Algorithm **Insert\_At\_End**

**Input:** The address of the first node ‘start’ of the list and the item to be inserted.

**Output:** The item stored in a node and added to the linked list at end.

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. ptr=**Create\_Node**(item)
2. **If**(ptr=null) **then** //failed allocation of ptr
3. **Print** “Insufficient memory”
4. **Exit**
5. **EndIf**
6. **If**(start=null)
7. start=ptr //update the address in start
8. end=ptr //set end to ptr
9. **Else**
10. end.LINK=ptr//point end link to ptr
11. end=ptr //set end to ptr
12. **EndIf**
13. **Return** start
14. **Stop**

Algorithm **Insert\_At\_Front**

**Input:** The address of the first node ‘start’ of the list and the item to be inserted.

**Output:** The item stored in a node and added to the linked list at front.

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. ptr=**Create\_Node**(item)
2. **If**(ptr=null) **then** //failed allocation of ptr
3. **Print** “Insufficient memory”
4. **Exit**
5. **EndIf**
6. **If**(start=null)
7. start=ptr //update start
8. end=ptr //update end
9. **Else**
10. ptr.LINK=start //point ptr link to start
11. start=ptr //reset start to ptr
12. **EndIf**
13. **Return** start
14. **Stop**

Algorithm **Insert\_At\_Any\_Position**

**Input:** The address of the first node ‘start’ of the list, the item to be inserted and the position ‘pos’ at which it is to be inserted in the list.

**Output:** The item stored in a node and added to the linked list at the position ‘pos’

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. **count=1**
2. **If**(pos=1) **then** //if insertion is to be done at front
3. start=**Insert\_At\_Front**(start,item)
4. **Else If**(pos=max) **then** //if insertion needs to be done at end
5. start=**Insert\_At\_End**(start,item)
6. **Else**
7. ptr=**Create\_Node**(item)
8. **If**(ptr=null) **then** //failed allocation of ptr
9. **Print** “insufficient memory”
10. **Exit**
11. **EndIf**
12. temp=start //set a temp pointer to start
13. **While**(count≠pos) **do** //count till pos is reached
14. count=count+1
15. temp=temp.LINK //point temp to temp link
16. **EndWhile**
17. ptr.LINK=temp.LINK //point ptr link to temp link
18. temp.LINK=ptr //point temp link to ptr
19. **EndIf**
20. **Return** start
21. **Stop**

Algorithm **Delete\_From\_Front**

**Input:** The address of the first node ‘start’ of the linked list

**Output:** The first node of the list deleted and freed from memory.

**Remarks:** The list should not be empty

**Steps:**

1. ptr=start //set ptr to beginning of the list
2. start=start.LINK //reset start to it’s link address
3. free the allocated pointer ptr from memory
4. **Return** start
5. **Stop**

Algorithm **Delete\_From\_End**

**Input:** The address of the first node ‘start’ of the linked list

**Output:** The last node of the list deleted and freed from memory.

**Remarks:** The list should not be empty

**Steps:**

1. ptr=start
2. **While**(ptr.LINK≠end) **do**
3. ptr=ptr.LINK
4. **EndWhile**
5. temp=end
6. end=ptr
7. end.LINK=null //set end pointer to null
8. The allocated memory at address temp is freed
9. **Stop**

Algorithm **Delete\_From\_Any\_Position**

**Input:** The address of the first node ‘start’ of the list and the position ‘pos’ of the node which is to be deleted.

**Output:** The node at position ‘pos’ deleted from the list and freed from memory.

**Remarks:** The list should not be empty.

**Steps:**

1. max = **Get\_Length**(start) //get number of nodes in list
2. count=1 //loop counter
3. **If**(pos=max) **then** //if end element is to be deleted
4. **Delete\_From\_End**(start)
5. **Else If**(pos=1) **then //**if front element is to be deleted
6. start=**Delete\_From\_Front**(start)
7. **Else**
8. temp=start
9. **While**(count≠pos) **do** //while pos is not reached
10. count=count+1
11. temp=temp.LINK //traverse the list
12. **EndWhile**
13. temp2=temp.LINK //point temp2 to temp link
14. temp.LINK=temp2.LINK //point temp link to temp2 link
15. The allocated memory at address temp2 is freed
16. **EndIf**
17. **Return** start
18. **Stop**

Algorithm **Get\_Length**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The total number of nodes in the list.

**Remarks:** The list should not be empty.

**Steps:**

1. temp=start
2. **While**(temp≠null) **do** //traverse till the end of the list
3. count=count+1
4. temp=temp.LINK
5. **Return** start
6. **Stop**

Algorithm **Display\_List**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The data of the list displayed to the user

**Remarks:** The list should not be empty.

**Steps:**

1. temp=start
2. **If**(start=null)
3. **Print** “the list is empty”
4. **Return**
5. **EndIf**
6. **While**(temp≠null) **do** //traverse till the end of the list
7. Print the data section of each node
8. temp=temp.LINK
9. **Return**
10. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//structure of nodes of the linked list

typedef struct node

{

    int data;

    struct node\* link;

}node;

//function to create a new node in heap

node\* createnode(int item)

{

    node\* ptr;

    ptr=(node\*)malloc(sizeof(node));

    ptr->link=NULL;

    ptr->data=item;

    return ptr;

}

node\* end; //global pointer variable to indicate end of the list

//function to display the data of list chronologically

void displist(node \*start)

{

    node \*temp;

    temp=start;

    if(start==NULL) //when list is empty

    {

        printf("LIST EMPTY");

        return;

    }

    printf("LIST: ");

    while(temp!=NULL) //traverse list till end

    {

        printf("%d ",temp->data);

        temp=temp->link;

    }

}

//function to insert a node at the end of the list

node\* endinsert(node \*start,int item)

{

    node \*ptr,\*temp;

    ptr=createnode(item);

    if(ptr=NULL) //failed allocation of ptr

    {

        printf("Insufficient memory");

        exit(1);

    }

    if(start==NULL) //when list is empty

    {

        start=ptr;

        end=ptr; //update end pointer

        ptr->link=NULL;

        return start;

    }

    else

    {

        end->link=ptr; //add new node after node pointed by end

        ptr->link=NULL;

        end=ptr; //update end

        return start;

    }

}

//function to insert a node at the front of the list

node\* frontinsert(node \*start,int item)

{

    node \*ptr;

    ptr=createnode(item);

    if(ptr=NULL) //failed allocation of ptr

    {

        printf("Insufficient memory");

        exit(1);

    }

    If(start==NULL) //when list is empty

    {

        start=ptr; //set ptr as start

        end=ptr; //set ptr as end since a single element is present

        ptr->link=NULL;

        return start;

    }

    Else

    {

        ptr->link=start; //point ptr link to start

        start=ptr; //update start

        return start;

    }

}

//function to calculate the number of elements in the list

int getmax(node\* start)

{

    int count=1; //loop counter

    node\* temp=start;

    while(temp!=NULL) //traverse till the end of the list

    {

        count++;

        temp=temp->link;

    }

    return count;

}

//function to insert a node at a user chosen position in the list

node\* insertatany(node\* start,int item,int pos)

{

    node\* ptr,\*temp;

    int count=1,max=getmax(start);

    if(pos==1) //if node is to be inserted at first position

        start=frontinsert(start,item);

    else if(pos==max) //if node is to be inserted at last position

        start=endinsert(start,item);

    else

    {

        ptr = createnode(item);

        if(ptr=NULL) //failed allocation of ptr

        {

            printf("Insufficient memory");

            exit(1);

        }

        temp = start;

        while(count!=pos-1) //traversing till pos-1 is reached

        {

            count++;

            temp=temp->link;

        }

        ptr->link=temp->link; //point ptr link to temp link

        temp->link=ptr; //point temp link to ptr

    }

        return start;

}

//function to delete the front node of the list

node\* frontdelete(node \*start)

{

    node \*ptr;

    ptr=start;

    start=start->link; //move start pointer by one node

    free(ptr); //delete the front node

    return start;

}

//function to delete the last node of the list

void enddelete(node \*start)

{

    node \*temp,\*temp2;

    temp=start;

    while((temp->link!=end)) //while end is not reached

        temp=temp->link; //traverse the list

    temp2=end;

    end=temp; //reset end to temp

    end->link=NULL;

    free(temp2);

}

//function to delete a user chosen node from the list

node\* deletefromany(node\* start,int pos)

{

    node\* temp,\*temp2;

    int max=getmax(start);

    int count=1;

    if(pos==max) //if last node needs to be deleted

        enddelete(start);

    else if(pos==1) //if first node is to be deleted

    {

        start=frontdelete(start);

        return start;

    }

    else

    {

        temp=start;

        while(count!=pos-1) //while pos-1 is not reached

        {   count++;

            temp=temp->link; //traverse the list

        }

        temp2=temp->link;

        temp->link=temp2->link; //point temp link to temp2 link

        free(temp2);

    }

    return start;

}

int main(void)

{

    node \*start=NULL;

    int ch,item,pos,max;

    while(1)

    {

        printf("\nMENU:");

        printf("\n1.INSERT AT END\n2.INSERT AT FRONT\n3.INSERT AT ANY POSITION\n4.DELETE FROM FRONT\n5.DELETE FROM END\n6.DELETE FROM ANY POSITION\n7.DISPLAY LIST\n8.EXIT");

        printf("\nEnter Your Choice:");

        scanf("%d",&ch);

        switch(ch)

        {

            case 1:

            printf("Enter the element to be inserted: ");

            scanf("%d",&item);

            start=endinsert(start,item);

            displist(start);

            break;

            case 2:

            printf("Enter the element to be inserted:");

            scanf("%d",&item);

            start=frontinsert(start,item);

            displist(start);

            break;

            case 3:

            if(start==NULL)

                start=endinsert(start,item);

            else

            {

                printf("Enter the insertion position: ");

                scanf("%d",&pos);

                max=getmax(start);

                //Checking validity of given position

                if(pos<1||pos>max+1)

                    printf("\nInvalid Position\n");

                else

                {

                    printf("Enter the element to be inserted:");

                    scanf("%d",&item);

                    start=insertatany(start,item,pos);

                }

            }

            displist(start);

            break;

            case 4:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                start=frontdelete(start);

                displist(start);

            }

            break;

            case 5:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                if(start->link==NULL)

                    start=frontdelete(start);

                else

                    enddelete(start);

                displist(start);

            }

            break;

            case 6:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                printf("Enter the deletion position: ");

                scanf("%d",&pos);

                max=getmax(start);

                if(pos<1 || pos>max)

                    printf("Invalid position");

                else

                {

                    start=deletefromany(start,pos);

                    displist(start);

                }

            }

            break;

            case 7:

                displist(start);

            break;

            case 8:

                exit(0);

            break;

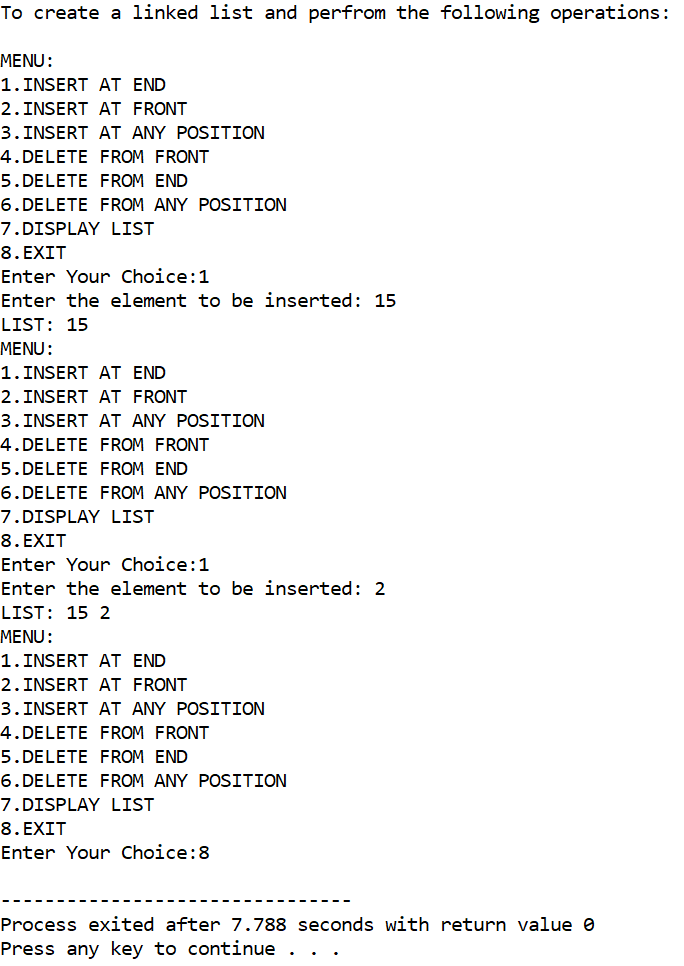
        }

    }

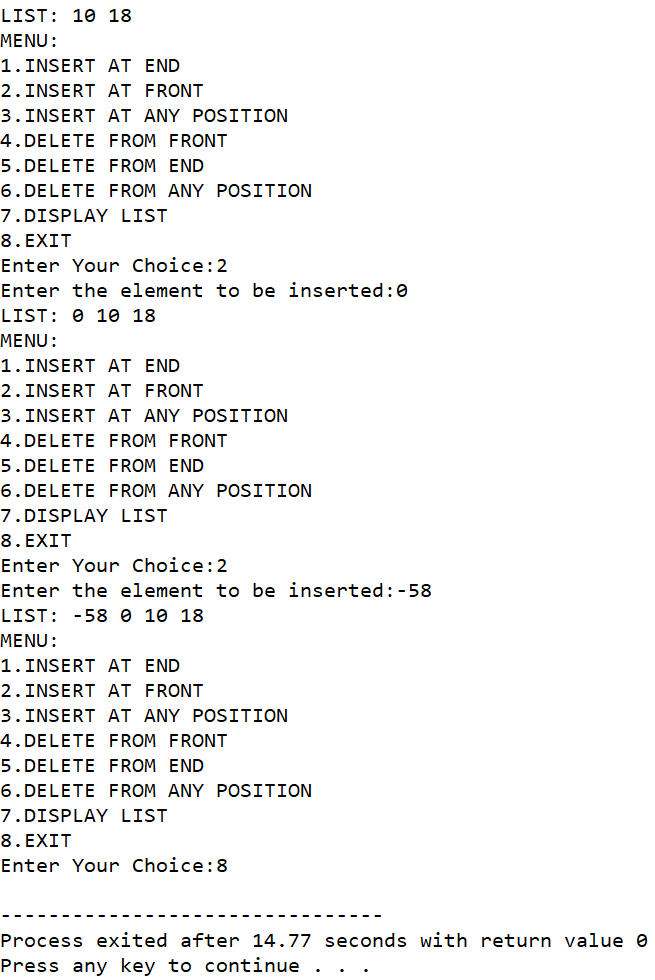
}

**4.OUTPUT**

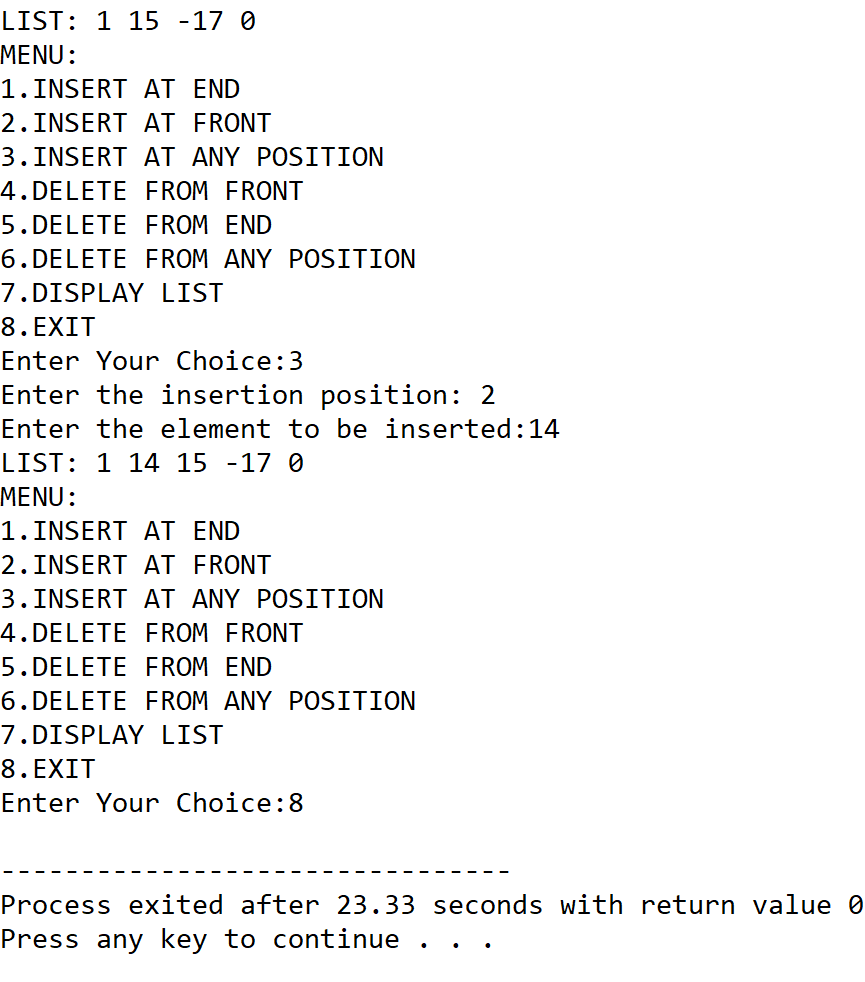
**SET 1:** Insert at end



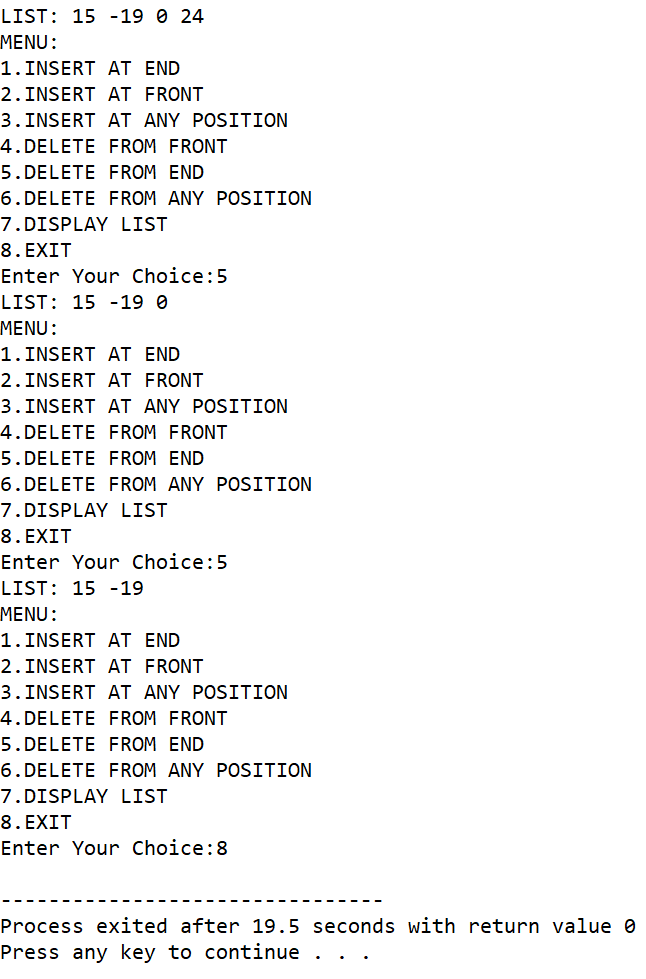
**SET 2:** Insert at front



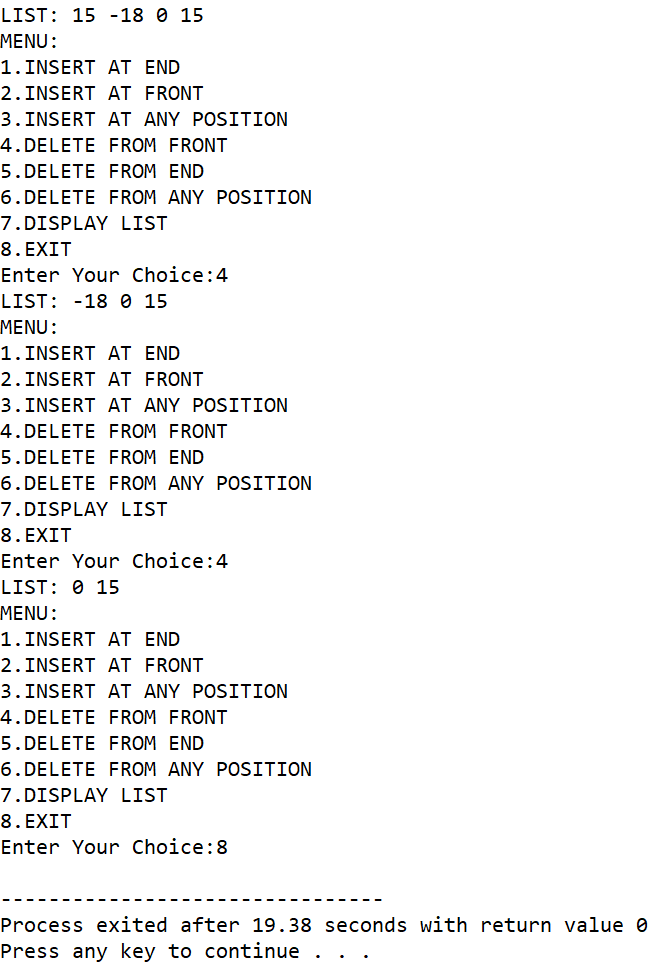
**SET 3:** Insert at any position



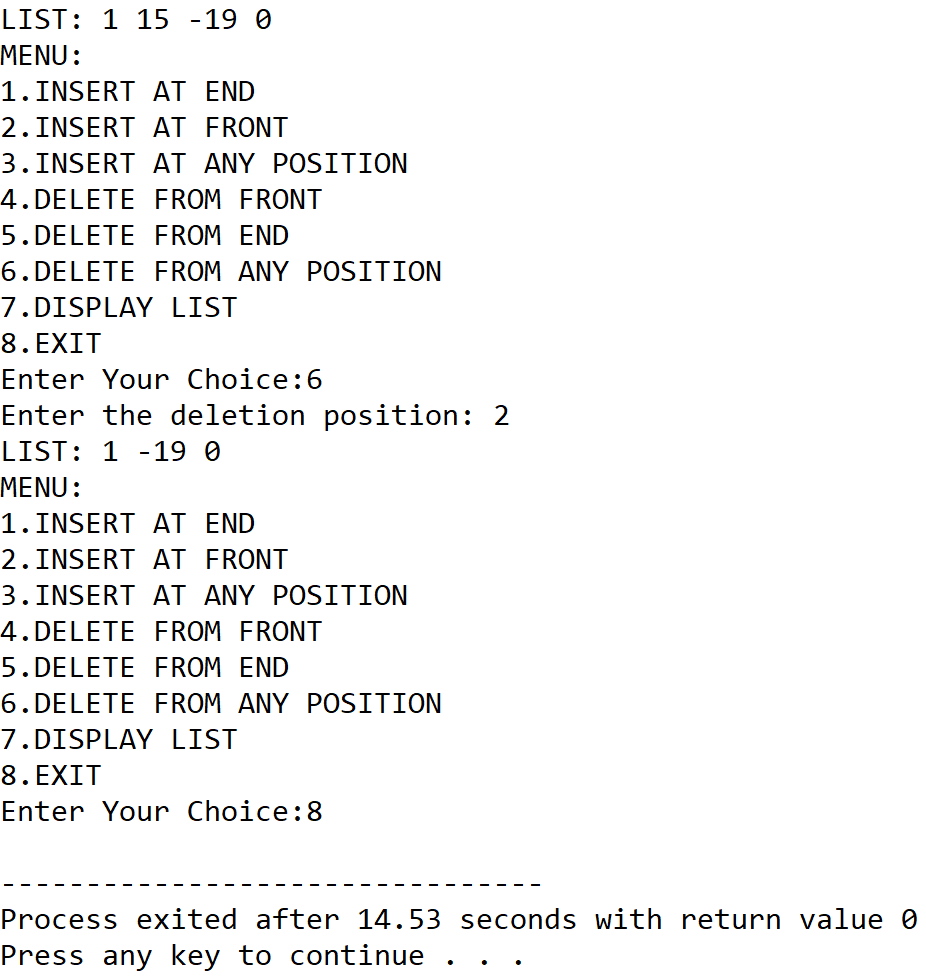
**SET 4:** Delete from end



**SET 5:** Delete from front



**SET 6:** Delete from any position



**5.DISCUSSIONS**

**Variable Description**

* **start:** pointer to the first node of the linked list.
* **item:** The item to be inserted into a node.
* **pos:** to hold user input position of a node.
* **max:** The total number of nodes in the list.
* **end:** pointer to the last element of the list.
* **ptr,temp,temp2**: pointers to the nodes of the list
* **count:** loop counter

**Limitations**

* Unlike arrays, extra variables need to be stored as pointers to refer to other nodes in the list, thus more memory is required to create the data structure.
* The nodes are not indexed intrinsically and thus, loops need to be used to find the location of a node.
* There is no provision to traverse the previous node from a particular node.

**Uses**

* The program can be used to perfrom various operations including insertion and deletion of nodes. For example, it can find implementation in a employee database where a list of employee data needs to be stored and the list needs to be updated with time by adding and deleting nodes at various positions throughout the list.

**Future Scope:**

* A separate field can be added to instrinsically store the index of a node for easier maintainance of very large lists.

**Teacher’s Signature**

**ASSIGNMENT – 21**

**1.PROBLEM STATEMENT**

Write a program in C to implement doubly linked list.

**2.ALGORITHMS**

Algorithm **Create\_Node**

**Input:** The item to be stored in a node

**Output:** The address to a node constructed in memory with item stored in its data part

**Remarks:** The node allocated has three sections: a data section to store integer data, a prev section to reference the previous node and a next section to reference the next node in a linked list.

**Steps:**

1. A node is allocated in memory and it’s address is stored in a pointer ptr
2. Set the next part to a null value
3. Set the prev part to a null value
4. Set the data part to the input item
5. **Return** ptr
6. **Stop**

Algorithm **Insert\_At\_End**

**Input:** The address of the first node ‘start’ of the list and the item to be inserted.

**Output:** The item stored in a node and added to the linked list at end.

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. ptr=**Create\_Node**(item)
2. **If**(ptr=null) **then** //failed allocation of ptr
3. **Print** “Insufficient memory”
4. **Exit**
5. **EndIf**
6. **If**(start=null)
7. start=ptr //update the address in start
8. end=ptr //set end to ptr
9. **Else**
10. end.NEXT=ptr//point end link to ptr
11. ptr.PREV=end
12. end=ptr //set end to ptr
13. **EndIf**
14. **Return** start
15. **Stop**

Algorithm **Insert\_At\_Front**

**Input:** The address of the first node ‘start’ of the list and the item to be inserted.

**Output:** The item stored in a node and added to the linked list at front.

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. ptr=**Create\_Node**(item)
2. **If**(ptr=null) **then** //failed allocation of ptr
3. **Print** “Insufficient memory”
4. **Exit**
5. **EndIf**
6. **If**(start=null)
7. start=ptr //update start
8. end=ptr //update end
9. **Else**
10. ptr.NEXT=start //point ptr link to start
11. start=ptr //reset start to ptr
12. **EndIf**
13. **Return** start
14. **Stop**

Algorithm **Insert\_At\_Any\_Position**

**Input:** The address of the first node ‘start’ of the list, the item to be inserted and the position ‘pos’ at which it is to be inserted in the list.

**Output:** The item stored in a node and added to the linked list at the position ‘pos’

**Remarks:** Node is created only when sufficient memory is available.

**Steps:**

1. **count=1**
2. **If**(pos=1) **then** //if insertion is to be done at front
3. start=**Insert\_At\_Front**(start,item)
4. **Else If**(pos=max) **then** //if insertion needs to be done at end
5. start=**Insert\_At\_End**(start,item)
6. **Else**
7. ptr=**Create\_Node**(item)
8. **If**(ptr=null) **then** //failed allocation of ptr
9. **Print** “insufficient memory”
10. **Exit**
11. **EndIf**
12. temp=start //set a temp pointer to start
13. **While**(count≠pos) **do** //count till pos is reached
14. count=count+1
15. temp=temp.NEXT //point temp to temp link
16. **EndWhile**
17. temp2=temp.NEXT
18. ptr.NEXT=temp2 //point ptr link to temp link
19. ptr.PREV=temp
20. temp.NEXT=ptr
21. temp2.PREV=ptr
22. **EndIf**
23. **Return** start
24. **Stop**

Algorithm **Delete\_From\_Front**

**Input:** The address of the first node ‘start’ of the linked list

**Output:** The first node of the list deleted and freed from memory.

**Remarks:** The list should not be empty

**Steps:**

1. ptr=start //set ptr to beginning of the list
2. **If**(start.NEXT=null) **then**
3. start=null
4. **Else**
5. start=start.NEXT
6. Start->PREV=null
7. **EndIf**
8. free the allocated pointer ptr from memory
9. **Return** start
10. **Stop**

Algorithm **Delete\_From\_End**

**Input:** The address of the first node ‘start’ of the linked list

**Output:** The last node of the list deleted and freed from memory.

**Remarks:** The list should not be empty

**Steps:**

1. ptr=start
2. **While**(ptr.NEXT≠end) **do**
3. ptr=ptr.NEXT
4. **EndWhile**
5. temp=end
6. end=ptr
7. end.NEXT=null //set end pointer to null
8. The allocated memory at address temp is freed
9. **Stop**

Algorithm **Delete\_From\_Any\_Position**

**Input:** The address of the first node ‘start’ of the list and the position ‘pos’ of the node which is to be deleted.

**Output:** The node at position ‘pos’ deleted from the list and freed from memory.

**Remarks:** The list should not be empty.

**Steps:**

1. max = **Get\_Length**(start) //get number of nodes in list
2. count=1 //loop counter
3. **If**(pos=max) **then** //if end element is to be deleted
4. **Delete\_From\_End**(start)
5. **Else If**(pos=1) **then //**if front element is to be deleted
6. start=**Delete\_From\_Front**(start)
7. **Else**
8. temp=start
9. **While**(count≠pos) **do** //while pos is not reached
10. count=count+1
11. temp=temp.NEXT //traverse the list
12. **EndWhile**
13. temp2=temp.NEXT //point temp2 to temp link
14. temp3=temp2.NEXT
15. temp.NEXT=temp3 //point temp link to temp2 link
16. temp3.PREV=temp
17. The allocated memory at address temp2 is freed
18. **EndIf**
19. **Return** start
20. **Stop**

Algorithm **Get\_Length**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The total number of nodes in the list.

**Remarks:** The list should not be empty.

**Steps:**

1. temp=start
2. **While**(temp≠null) **do** //traverse till the end of the list
3. count=count+1
4. temp=temp.LINK
5. **Return** start
6. **Stop**

Algorithm **Display\_List**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The data of the list displayed to the user

**Remarks:** The list should not be empty.

**Steps:**

1. temp=start
2. **If**(start=null)
3. **Print** “the list is empty”
4. **Return**
5. **EndIf**
6. **While**(temp≠null) **do** //traverse till the end of the list
7. Print the data section of each node
8. temp=temp.LINK
9. **Return**
10. **Stop**

Algorithm **Display\_List\_In\_Reverse**

**Input:** The address of the first node ‘start’ of the list.

**Output:** The data of the list displayed in reverse order

**Remarks:** The list should not be empty.

**Steps:**

1. ptr=end
2. **If**(start=null) **then**
3. **Print** “list is empty”
4. **Exit**
5. **EndIf**
6. **While**(ptr≠null) **do**
7. display the data part of each node
8. ptr=ptr.PREV
9. **EndWhile**
10. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

typedef struct node

{

    struct node\* prev;

    int data;

    struct node\* next;

}node;

node\* createnode(int item)

{

    node\* ptr;

    ptr=(node\*)malloc(sizeof(node));

    ptr->data=item;

    ptr->next=NULL;

    ptr->prev=NULL;

    return ptr;

}

node\* end;

void displist(node \*start)

{

    node \*temp;

    temp=start;

    if(start==NULL)

    {

        printf("LIST EMPTY");

        return;

    }

    printf("LIST: ");

    while(temp!=NULL)

    {

        printf("%d ",temp->data);

        temp=temp->next;

    }

}

node\* endinsert(node \*start,int item)

{

    node \*ptr,\*temp;

    ptr=createnode(item);

    if(start==NULL)

    {

        start=ptr;

        end=ptr;

        return start;

    }

    else

    {

        end->next=ptr;

        ptr->prev=end;

        end=ptr;

        return start;

    }

}

node\* frontinsert(node \*start,int item)

{

    node \*ptr;

    ptr=createnode(item);

    if(start==NULL)

    {

        start=ptr;

        end=ptr;

        return start;

    }

    else

    {

        ptr->next=start;

        start=ptr;//caution

        return start;

    }

}

int getmax(node\* start)

{

    int count=0;

    node\* temp=start;

    while(temp!=NULL)

    {

        temp=temp->next;

        count++;

    }

    return count;

}

node\* insertatany(node\* start,int item,int pos)

{

    node\* ptr,\*temp,\*temp2;

    int count=1,max=getmax(start);

    if(pos==1)

        start=frontinsert(start,item);

    else if(pos==max)

        start=endinsert(start,item);

    else

    {

        ptr = createnode(item);

        temp = start;

        while(count!=pos-1)

        {

            count++;

            temp=temp->next;

        }

        temp2=temp->next;

        ptr->next=temp2;

        ptr->prev=temp;

        temp->next=ptr;

        temp2->prev=ptr;

    }

        return start;

}

node\* frontdelete(node \*start)

{

    node \*ptr;

    ptr=start;

    if(start->next==NULL)

        start=NULL;

    else

    {

        start=start->next;

        start->prev=NULL;

    }

    free(ptr);

    return start;

}

void enddelete(node \*start)

{

    node \*temp,\*temp2;

    temp=start;

    while((temp->next!=end))

        temp=temp->next;

    temp2=end;

    end=temp;

    end->next=NULL;

    free(temp2);

}

node\* deletefromany(node\* start,int pos)

{

    node\* temp,\*temp2,\*temp3;

    int max=getmax(start);

    int count=0;

    if(pos==max)

        enddelete(start);

    else if(pos==1)

    {

        start=frontdelete(start);

        return start;

    }

    else

    {

        temp=start;

        while(count!=pos-2)

        {   count++;

            temp=temp->next;

        }

        temp2=temp->next;

        temp3=temp2->next;

        temp->next=temp3;

        temp3->prev=temp;

        free(temp2);

    }

    return start;

}

void disprev(node\* start)

{

    node\* ptr;

    ptr=end;

    if(start==NULL)

    {

        printf("LIST EMPTY");

        return;

    }

    printf("REVERSE LIST: ");

    while(ptr!=NULL)

    {

        printf("%d ",ptr->data);

        ptr=ptr->prev;

    }

}

int main(void)

{

    node \*start=NULL;

    int ch,item,pos,max;

    while(1)

    {

        printf("\nMENU:");

        printf("\n1.INSERT AT END\n2.INSERT AT FRONT\n3.INSERT AT ANY POSITION\n4.DELETE FROM FRONT\n5.DELETE FROM END\n6.DELETE FROM ANY POSITION\n7.DISPLAY LIST\n8.DISPLAY LIST IN REVERSE\n9.EXIT");

        printf("\nEnter Your Choice:");

        scanf("%d",&ch);

        switch(ch)

        {

            case 1:

            printf("Enter the element to be inserted: ");

            scanf("%d",&item);

            start=endinsert(start,item);

            displist(start);

            break;

            case 2:

            printf("Enter the element to be inserted:");

            scanf("%d",&item);

            start=frontinsert(start,item);

            displist(start);

            break;

            case 3:

            if(start==NULL)

                start=endinsert(start,item);

            else

            {

                printf("Enter the insertion position: ");

                scanf("%d",&pos);

                max=getmax(start);

                //Checking validity of given position

                if(pos<1||pos>max+1)

                    printf("\nInvalid Position\n");

                else

                {

                    printf("Enter the element to be inserted:");

                    scanf("%d",&item);

                    start=insertatany(start,item,pos);

                }

            }

            displist(start);

            break;

            case 4:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                start=frontdelete(start);

                displist(start);

            }

            break;

            case 5:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                if(start->next==NULL)

                    start=frontdelete(start);

                else

                    enddelete(start);

                displist(start);

            }

            break;

            case 6:

            if(start==NULL)

                printf("LIST EMPTY");

            else

            {

                printf("Enter the deletion position: ");

                scanf("%d",&pos);

                max=getmax(start);

                if(pos<1 || pos>max)

                    printf("Invalid position");

                else

                {

                    start=deletefromany(start,pos);

                    displist(start);

                }

            }

            break;

            case 7:

                displist(start);

            break;

            case 8:

                disprev(start);

            break;

            case 9:

                exit(0);

            break;

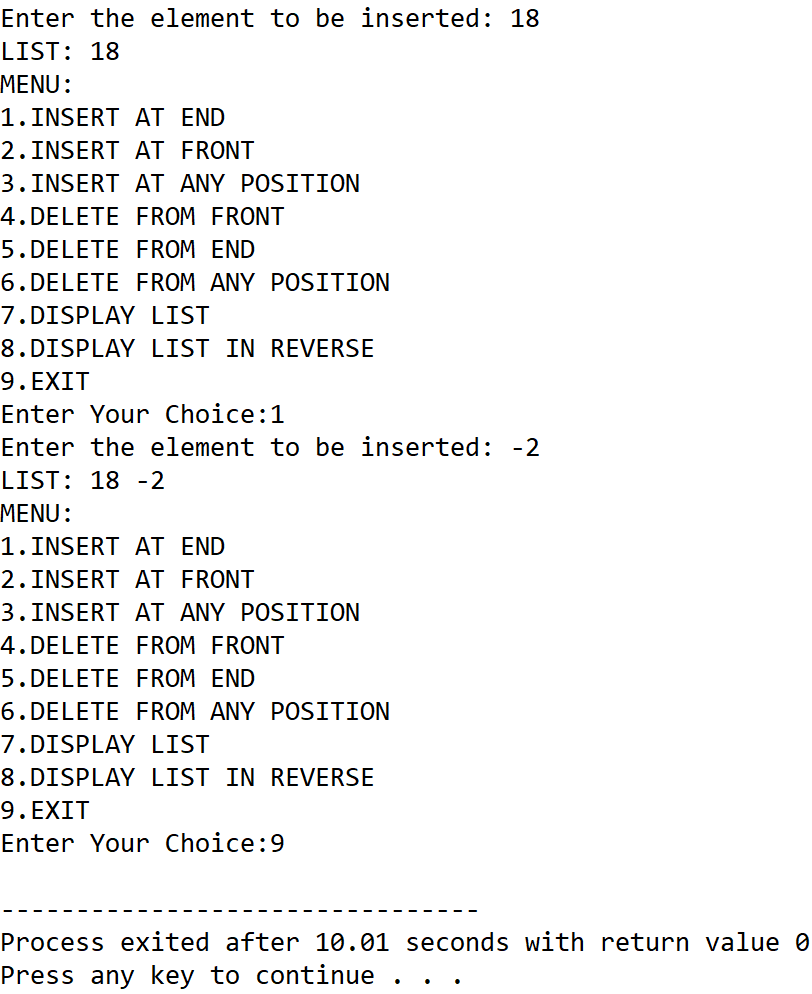
        }

    }

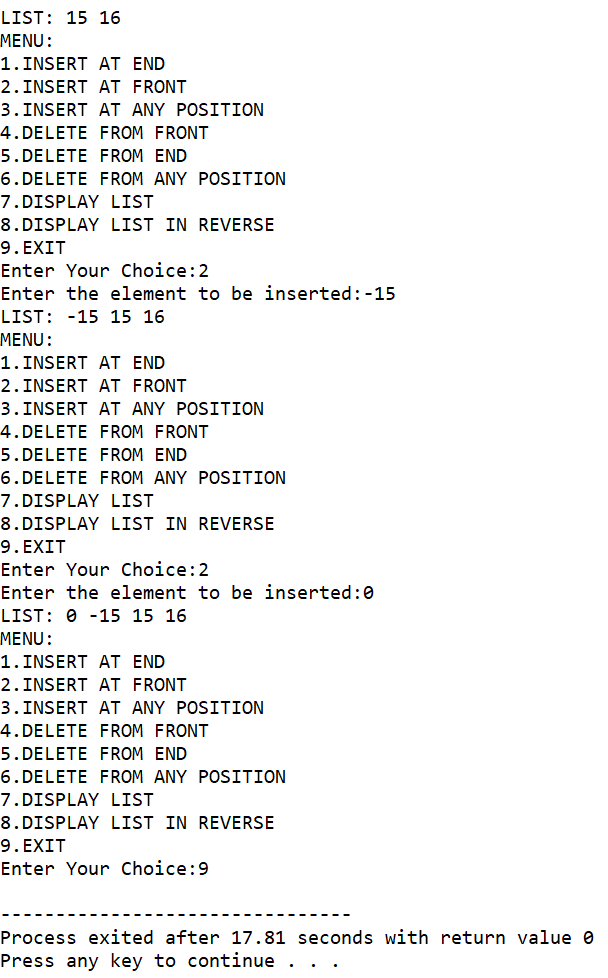
}

**4.OUTPUT**

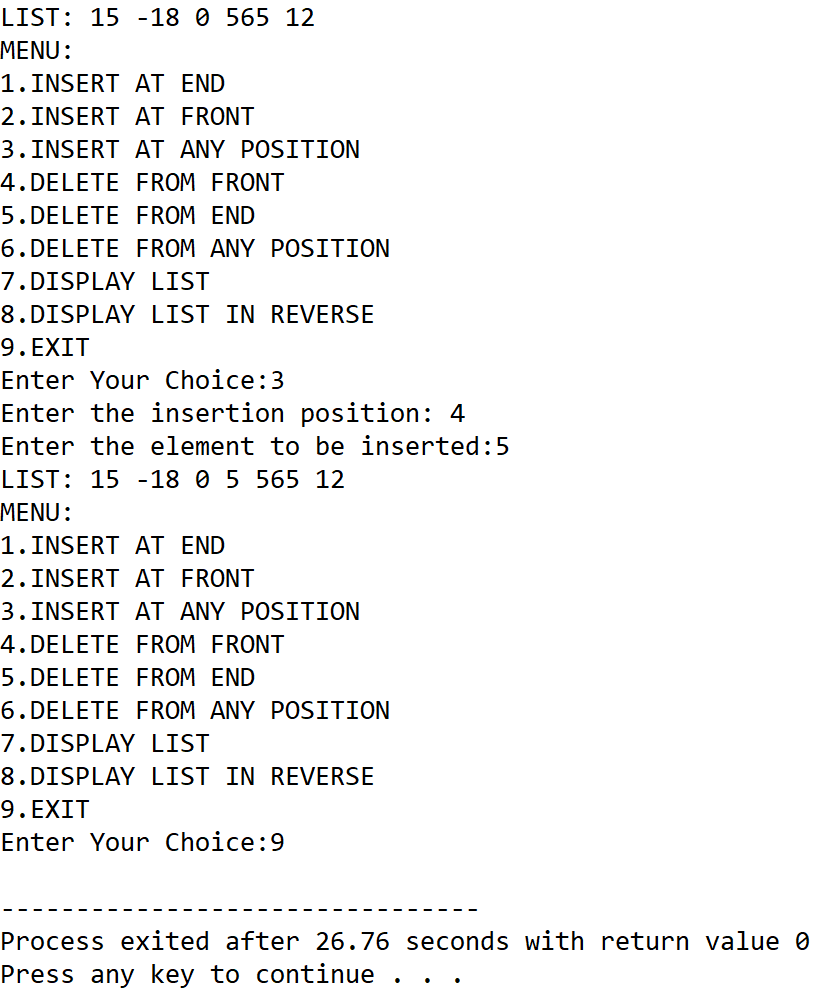
**SET 1:** Insert at end



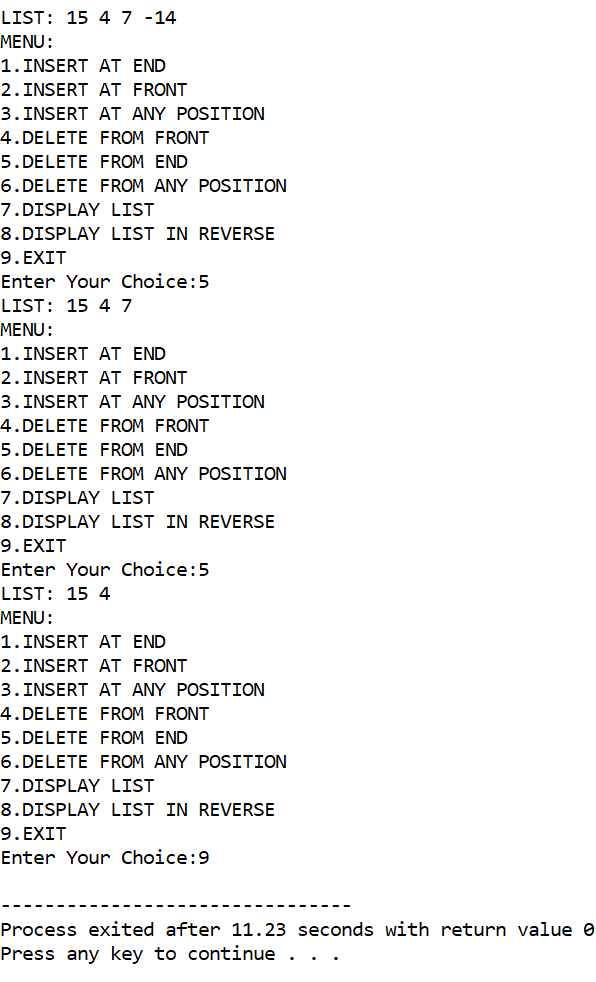
**SET 2:** Insert at front



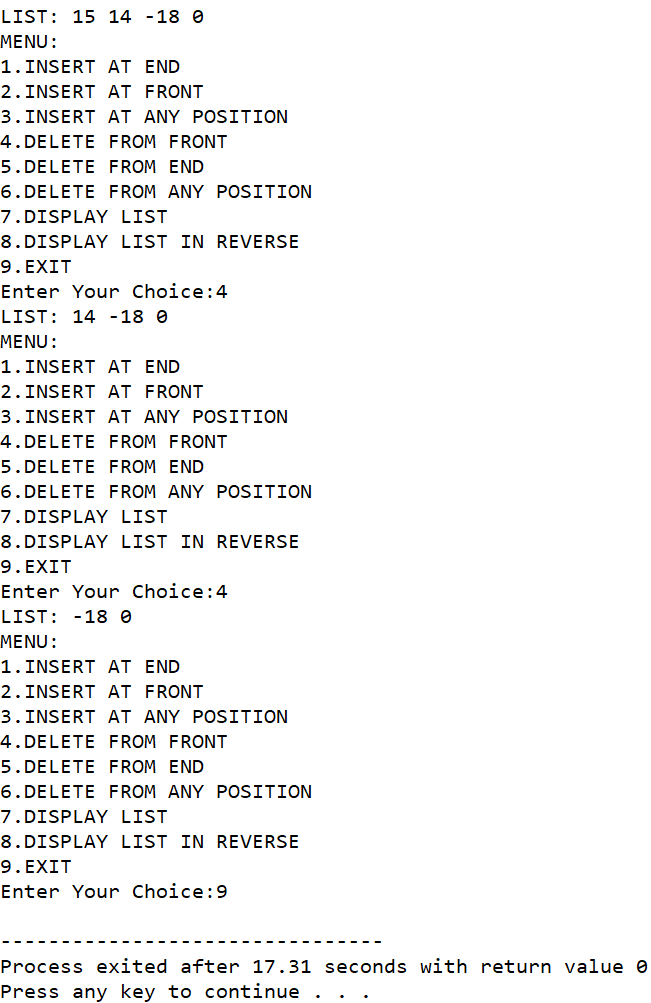
**SET 3:** Insert at any position



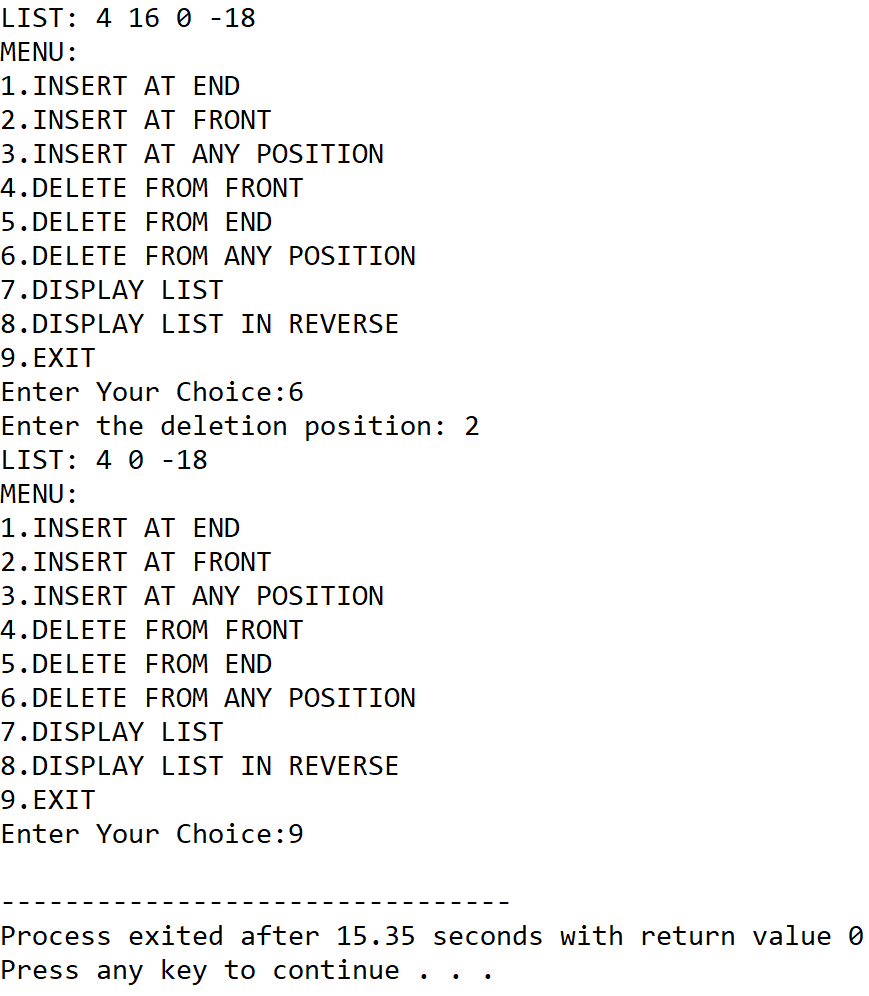
**SET 4:** Delete from end



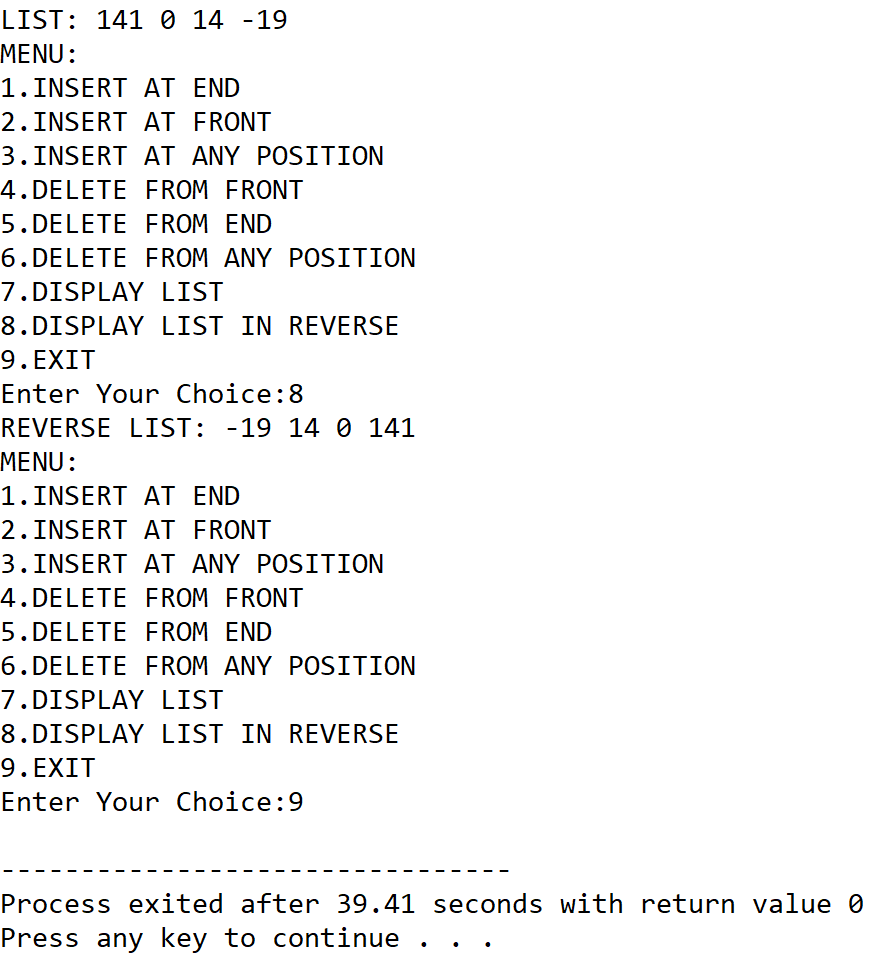
**SET 5:** Delete from front



**SET 6:** Delete from any position



**SET 7:** Display list in reverse order



**5.DISCUSSIONS**

**Variable Description**

* **start:** pointer to the first node of the linked list.
* **item:** The item to be inserted into a node.
* **pos:** to hold user input position of a node.
* **max:** The total number of nodes in the list.
* **end:** pointer to the last element of the list.
* **ptr,temp,temp2,temp3**: pointers to the nodes of the list
* **count:** loop counter

**Limitations**

* Unlike arrays, extra variables need to be stored as pointers to refer to other nodes in the list, thus more memory is required to create the data structure.
* The nodes are not indexed intrinsically and thus, loops need to be used to find the location of a node.

**Uses**

* The program can be used to perfrom various operations including insertion and deletion of nodes. For example, it can find implementation in a employee database where a list of employee data needs to be stored and the list needs to be updated with time by adding and deleting nodes at various positions throughout the list.

**Future Scope:**

* A separate field can be added to instrinsically store the index of a node for easier maintainance of very large lists.

**Teacher’s Signature**

**ASSIGNMENT – 22**

**1.PROBLEM STATEMENT**

Write a program in C to merge two sorted lists into a single list.

**2.ALGORITHMS**

Algorithm **Create\_Node**

**Input:** The data ‘item’ which is to be inserted into the node

**Output:** A node constructed in memory with item inserted as the data

**Remarks:** Node is allocated only if sufficient memory is available

**Steps:**

1. A node is allocated in memory and it’s address is stored in a pointer ptr
2. ptr.DATA=item
3. ptr.LINK=null
4. Return **ptr**
5. **Stop**

Algorithm **Insert\_Node**

**Input:** The address ‘start’ of the first node of the list and the data ‘item’ to be inserted

**Output:** A node inserted into the ‘start’ linked list with ‘item’ stored in it.

**Remarks:** Node is allocated only if sufficient memory is available.

**Steps:**

1. Ptr=**Create\_Node**(item)
2. **If**(start=null) **then** //for an empty list
3. start=ptr //ptr becomes start
4. end=ptr //ptr becomes end
5. **Else**
6. end.LINK=ptr //point end link to ptr
7. end=ptr //update end to ptr
8. **EndIf**
9. **Return** start

Algorithm **Merge\_Lists**

**Input:** The address of the first nodes both the lists, namely ‘start1’ and ‘start3’ and the address of the first node ‘start3’ of the linked list in which the merged result is to be stored.

**Output:** The contents of the the list ‘start1’ and ‘start2’ merged in ascending order within ‘start3’.

**Remarks:** The lists ‘start1’ and ‘start2’ must be in sorted order.

**Steps:**

1. Temp1=start1,temp2=start2
2. **While**(temp≠null **AND** temp≠null) **do** //untill either list exhausts
3. **If**(temp1.DATA≤temp2.DATA) **then** //temp1.DATA is greater or equal
4. start3=**Insert\_Node**(start3,temp1.DATA) //insert in start3
5. Temp1=temp1.LINK //shift temp1 to next node
6. **Else**
7. Start3=**Insert\_Node**(start3,temp2.DATA)
8. Temp2=temp2.LINK
9. **EndIf**
10. **EndWhile**
11. **If**(Temp1=null) **then** //when first list exhausts
12. **While**(temp2≠null) **do** //copy the second list as it is
13. Start3=**Insert\_Node**(start3,temp2.DATA)
14. **EndWhile**
15. **Else**
16. **While**(temp1≠null) **do** //copy the first list as it is
17. Start3=**Insert\_Node**(start3,temp1.DATA)
18. Temp1=temp1.LINK
19. **Endwhile**
20. **EndIf**
21. **Return** start3
22. Stop

**3.SOURCE CODE**

#include<stdlib.h>

#include<stdio.h>

//structure of a node of a linked list

typedef struct node

{

    int data;

    struct node \*link;

}node;

//function to create a new node in memory

node\* createnode(int item)

{

    node\* ptr;

    ptr=(node\*)malloc(sizeof(node));

    ptr->data=item; //store item in data part

    ptr->link=NULL; //initialise ptr link to NULL

    return ptr;

}

//function to display a linked list

void displist(node \*start)

{

    node \*temp;

    temp=start;

    while(temp!=NULL)

    {

        printf("%d ",temp->data);

        temp=temp->link;

    }

}

node \*end; //global pointer to point to the last element of the list

//function to insert a new node at the end of a linked list

node\* insertnode(node\* start,int item)

{

    node \*ptr,\*temp=start;

    ptr=createnode(item);

    if(start==NULL) //if list is empty

{

        start=ptr;

end=ptr

}

    else

    {

        end->link=ptr //insert ptr at end

end=ptr //update end

    }

    return start;

}

//function to take input in a list

node\* getlist(node \*start,int len)

{

    int i,item;

    printf("Enter %d elements: ",len);

    for(i=0;i<len;i++)

    {

        scanf("%d",&item);

        start=insertnode(start,item);

    }

    return start;

}

//function to merge two sorted lists into a single list

node\* mergelist(node\* start1,node\* start2,node\* start3)

{

    node \*temp1=start1,\*temp2=start2;

    int i,j,k;

    while(temp1!=NULL && temp2!=NULL) //while both lists don’t exhaust

        {

            if((temp1->data)<=(temp2->data))

            {

                start3=insertnode(start3,temp1->data);

                temp1=temp1->link;

            }

            else

            {

                start3=insertnode(start3,temp2->data);

                temp2=temp2->link;

            }

        }

    if(temp1==NULL) //if list1 ends while list2 still remains

    {

        while(temp2!=NULL) // copy all the data of list2 to merged list

        {

            start3=insertnode(start3,temp2->data);

            temp2=temp2->link;

        }

    }

    else // if list2 ends while list1 still remains

    {

        while(temp1!=NULL) //copy all data of list1 to merged list

        {

            start3=insertnode(start3,temp1->data);

            temp1=temp1->link;

        }

    }

    return start3;

}

int main(void)

{

    node \*start1=NULL,\*start2=NULL,\*start3=NULL;

    int len1,len2,len3=len1+len2;

printf("To merge two sorted lists into a single list: \n");

    printf("Enter the length of the first list: ");

    scanf("%d",&len1);

    start1=getlist(start1,len1);

    printf("\nEntered List: ");

    displist(start1);

    printf("\nEnter the length of the second list: ");

    scanf("%d",&len2);

    start2=getlist(start2,len2);

    printf("\nEntered List: ");

    displist(start2);

    start3=mergelist(start1,start2,start3);

    printf("\nMerged List: ");

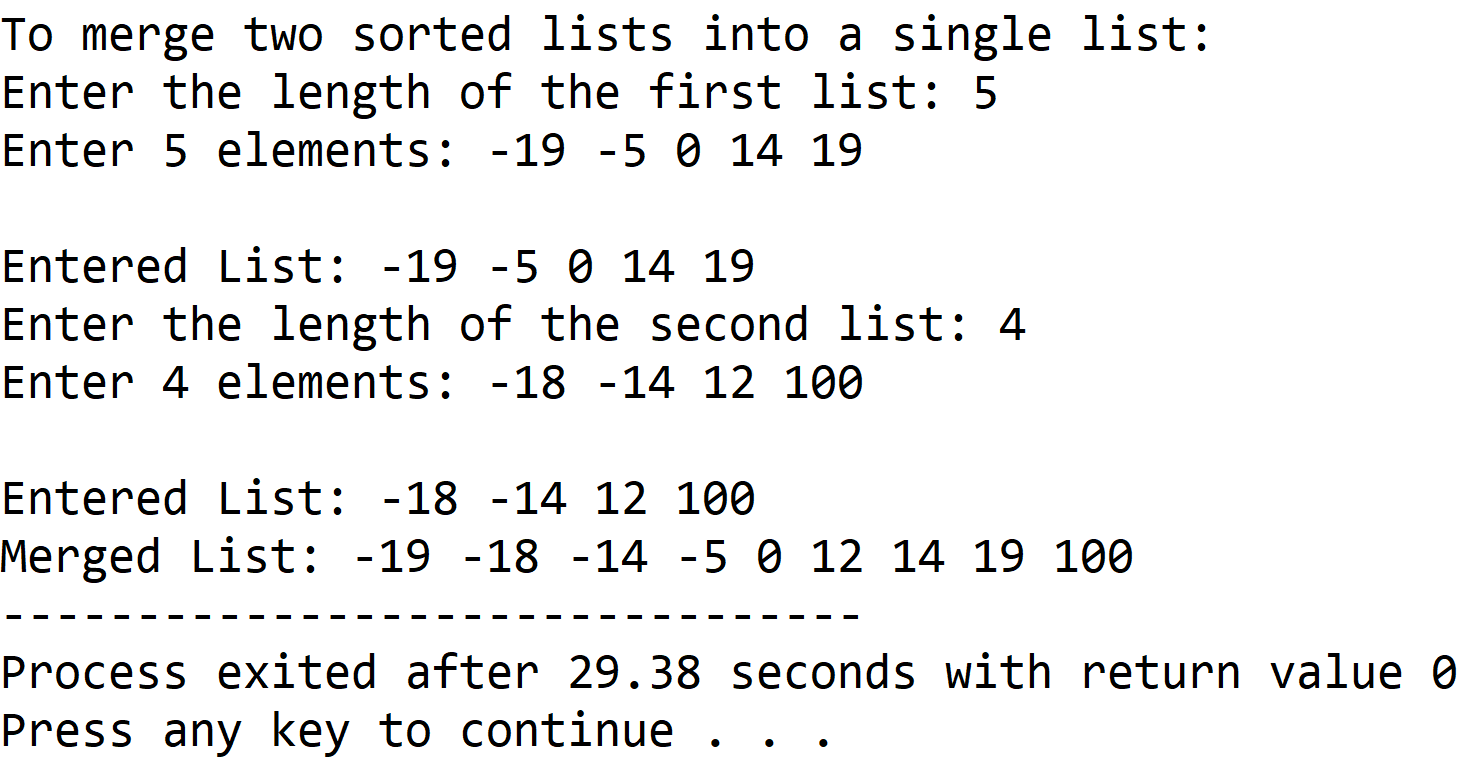
    displist(start3);

    return 0;

}

**4.OUTPUT**

**SET 1:** Merging of two sorted lists



**5.DISCUSSIONS**

**Variable Description**

* **start1,start2,start3:** header nodes of the first, second and third lists repectively.
* **len1,len2,len3:**Number of elements in first,second and third list respectively.
* **temp1,temp2,ptr:** pointers to nodes.
* **Item:** to hold user entered data during insertion of nodes.
* **I,j,k:** loop counters.

**Limitations:**

* Unlike arrays, the nodes need extra memory locations as pointers to refer to other nodes in the linked list, this leads to more usage of memory.
* There is no intrinsic indexing of the nodes, thus iterative constructs have to be used to find the location of a node.
* There is no provision to traverse backwards in the list.

**Uses**

* The program can be used to merge any two sorted list of integers into a single sorted list. It is of huge importance in several sorting procedures such as merge sort.

**Future Scope**

* A doubly linked list can be used to add the provision of backwards traversal in the list
* A extra index field can be added to each node to indicate the location of each node.

**Teacher’s Signature**

**ASSIGNMENT – 23**

**1.PROBLEM STATEMENT**

Write a program in C to implement Binary Search Tree and include insertion, searching and display operations.

**2.ALGORITHMS**

Algorithm **Create\_Node**

**Input:** The element ‘item’ which is to be inserted into the tree.

**Output:** A node of the tree is created in memory with it’s data section holding ‘item’ and it’s address returned in a pointer ‘ptr’.

**Remarks:** Each node has three sections: a DATA section to hold integer data, a LCHILD section to point to the left subtree and a RCHILD section to point to the right subtree.

**Steps:**

1. A node is allocated in memory and its memory address is returned in a pointer ‘ptr’
2. ptr.DATA=item
3. ptr.LCHILD=ptr.RCHILD=null
4. **Return** ptr
5. **Stop**

Algorithm **Insert\_In\_BST**

**Input:** The address of the root node ‘root’ of the tree and the integer ‘item’ which is to be inserted.

**Output:** The element ‘item’ inserted at it’s appropriate position in the binary search tree.

**Remarks:** Recursive procedure

**Steps:**

1. **If**(root=null) **then** //if tree is empty
2. ptr=**Create\_Node**(item) //set root to ptr
3. **Return** ptr
4. **EndIf**
5. **If**(item<root.DATA) //if key is less than root data
6. Root.LCHILD=**Insert\_In\_BST**(root.LCHILD,item) //go to left subtree
7. **Else If**(item>root.DATA)
8. Root.RCHILD=**Insert\_In\_BST**(root.RCHILD,item)//go to right subtree
9. **Return** root
10. **Stop**

Algorithm **Display\_Inorder**

**Input:** The address of the root node of the tree which is to be displayed

**Output:** The contents of the tree displayed in inorder traversal

**Remarks:** Inorder traversal: LCHILD🡪root🡪RCHILD

**Steps:**

1. **If**(root≠null) **then** //tree is not empty
2. **Display\_Inorder**(root.LCHILD) //recursively print right subtree
3. **Print** the root.DATA element //print root data
4. **Display\_Inorder**(root.RCHILD) //recursively print left subtree
5. **EndIf**
6. **Return**
7. **Stop**

Algorithm **Display\_preorder**

**Input:** The address of the root node of the tree which is to be displayed

**Output:** The contents of the tree displayed in preorder traversal

**Remarks:** Inorder traversal: root🡪LCHILD🡪RCHILD

**Steps:**

1. **If**(root≠null) **then** //tree is not empty
2. **Print** the root.DATA element //print root data
3. **Display\_Inorder**(root.LCHILD) //recursively print left subtree
4. **Display\_Inorder**(root.RCHILD) //recursively print right subtree
5. **EndIf**
6. **Return**
7. **Stop**

Algorithm **Display\_postorder**

**Input:** The address of the root node of the tree which is to be displayed

**Output:** The contents of the tree displayed in postorder traversal

**Remarks:** Inorder traversal:LCHILD🡪RCHILD🡪root

**Steps:**

1. **If**(root≠null) **then**
2. **Display\_Inorder**(root.LCHILD) //recursively print left subtree
3. **Display\_Inorder**(root.RCHILD) //recursively print right subtree
4. **Print** the root.DATA element //print root data
5. **EndIf**
6. **Return**
7. **Stop**

Algorithm **Search\_BST**

**Input:** The address of the root node ‘root’ of the tree and the element ‘item’ which is to be inserted.

**Output:** If the element is found at a node, the node address is returned, otherwise null is returned.

**Remarks:**The key should be an integer

**Steps:**

1. **If**(root=null **OR** root.DATA=item) **then** //if root data matches key
2. **Return** root
3. **EndIf**
4. **If**(item>root.DATA) **then** //if key is greater than root data
5. **Return Search\_BST**(root.RCHILD,item) //search right subtree
6. **Else**
7. **Return Search\_BST**(root.LCHILD,item) //search left subtree
8. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

#include<stdbool.h>

//structure to represent nodes of tree

typedef struct node

{

    int data;

    struct node\* lchild;

    struct node\* rchild;

}node;

//function to allocate node in heap

node\* createnode(int item)

{

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

    ptr->lchild=ptr->rchild=NULL;

    ptr->data=item;

    return ptr;

}

//function to insert a node in BST

node\* insert(node\* root,int item)

{

    node \*ptr;

    if(root==NULL) //if tree is empty

    {

        ptr=createnode(item);

        return ptr;

    }

    if(item<root->data)

        root->lchild=insert(root->lchild,item);

    else if(item>root->data)

        root->rchild=insert(root->rchild,item);

    return root;

}

//function to traverse BST by inorder

void disp\_inorder(node \*root)

{

    if(root!=NULL) //if tree is not empty

    {

        disp\_inorder(root->lchild);

        printf("%d ",root->data);

        disp\_inorder(root->rchild);

    }

}

//function to traverse BST by preorder

void disp\_preorder(node \*root)

{

    if(root!=NULL) //if tree is not empty

    {

        printf("%d ",root->data);

        disp\_preorder(root->lchild);

        disp\_preorder(root->rchild);

    }

}

//function to traverse BST by preorder

void disp\_postorder(node \*root)

{

    if(root!=NULL) //if tree is not empty

    {

        disp\_postorder(root->lchild);

        disp\_postorder(root->rchild);

        printf("%d ",root->data);

    }

}

//function to search an item in BST

node\* search\_BST(node \*root,int item)

{

    if(root==NULL || root->data==item)

        return root;

    if(item>root->data)

        return search\_BST(root->rchild,item);

    return search\_BST(root->lchild,item);

}

int main(void)

{

    bool found=false;

    node \*root=NULL,\*root2=NULL;

    int ch,item;

    printf("To perfrom the following operations on a Binary Search Tree: ");

    while(1)

    {

        printf("\nMENU: ");

        printf("\n1.INSERT\n2.SEARCH\n3.DISPLAY\n4.EXIT");

        printf("\nEnter your choice: ");

        scanf("%d",&ch);

        switch(ch)

        {

            case 1:

            printf("Enter the item to be inserted: ");

            scanf("%d",&item);

            root=insert(root,item);

            printf("Element of BST: \n");

            if(root!=NULL)

            {

                printf("INORDER TRAVERSAL: ");

                disp\_inorder(root);

                printf("\nPREORDER TRAVERSAL: ");

                disp\_preorder(root);

                printf("\nPOSTORDER TRAVERSAL: ");

                disp\_postorder(root);

            }

            else

                printf("Tree is empty");

            break;

            case 2:

            printf("Enter the key to be searched: ");

            scanf("%d",&item);

            if(root!=NULL)

                root2=search\_BST(root,item);

            else

                printf("Tree is empty");

            if(root2==NULL)

                printf("Element %d not found in the tree",item);

            else

                printf("Element %d found in the tree",item);

            break;

            case 3:

            if(root!=NULL)

            {

                printf("INORDER TRAVERSAL: ");

                disp\_inorder(root);

                printf("\nPREORDER TRAVERSAL: ");

                disp\_preorder(root);

                printf("\nPOSTORDER TRAVERSAL: ");

                disp\_postorder(root);

            }

            else

                printf("Tree is empty");

            break;

            case 4:

            exit(1);

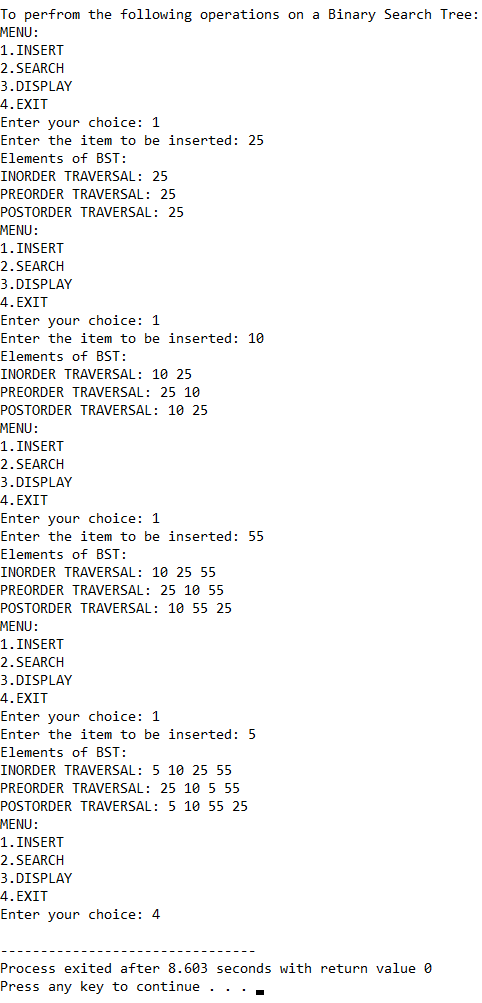
        }

    }

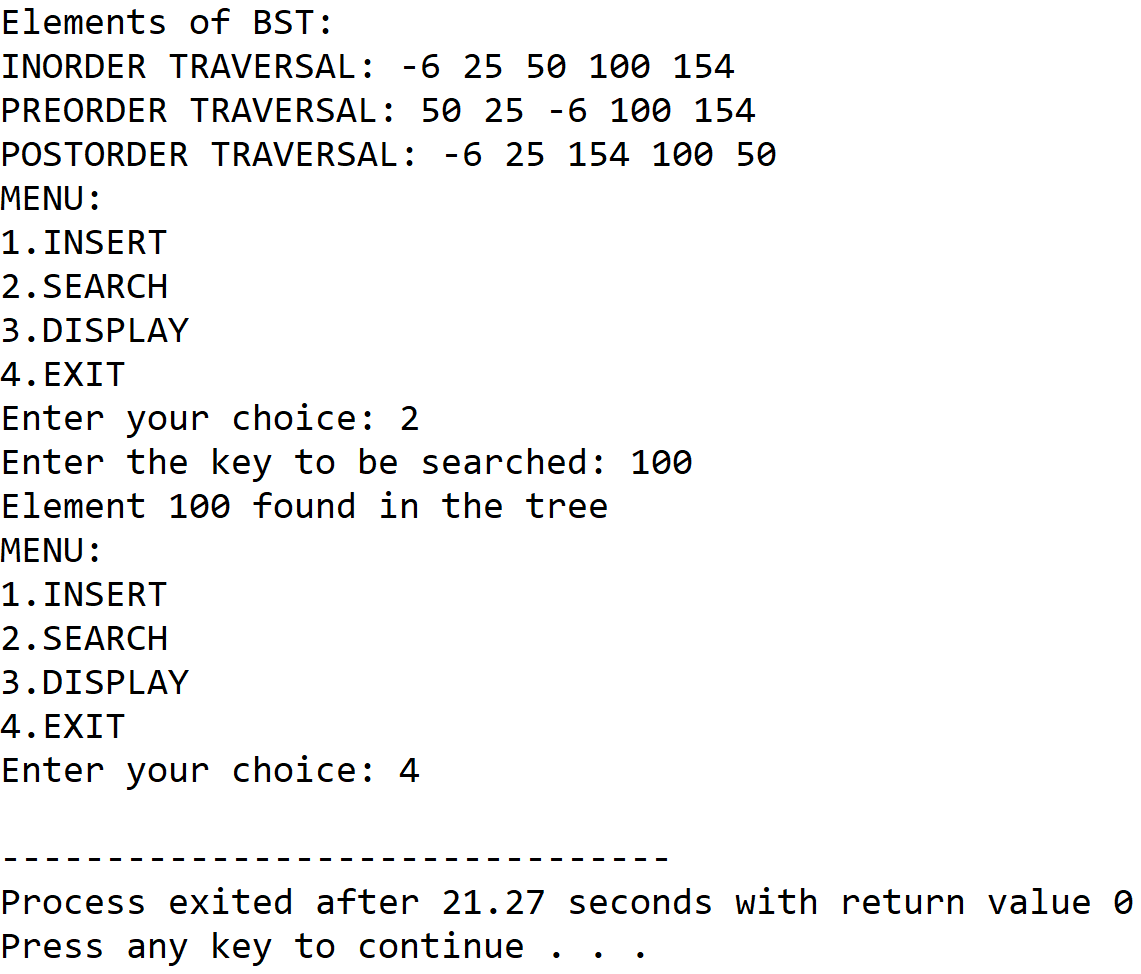
}

**4.OUTPUT**

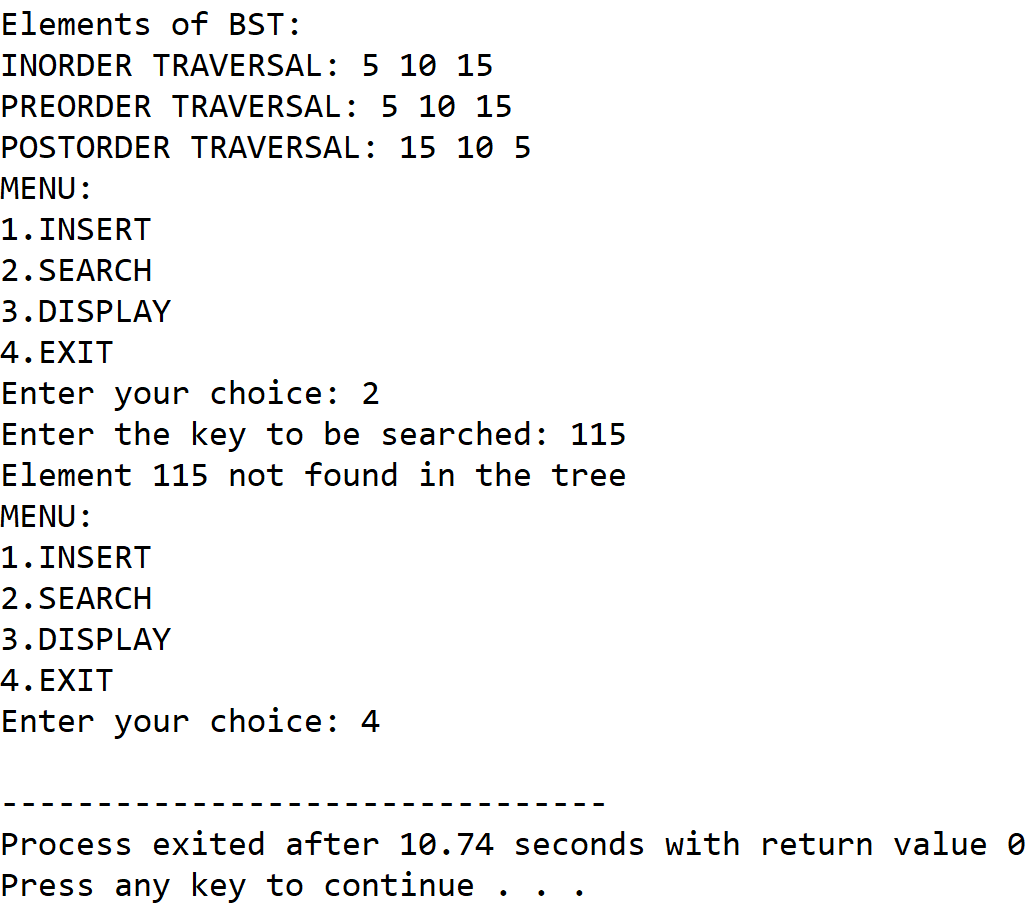
**SET 1:** Insertion

****

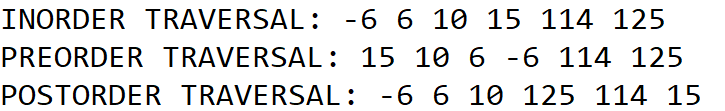
**SET 2:** Search successful



**SET 3:** Search unsuccessful



**SET 4**: Display



**5.DISCUSSIONS**

**Variable Description**

* **root,root2:** pointers to hold root nodes of the binary search tree
* **item:** to hold the integer to be inserted into the tree
* **ptr:** pointer to nodes of the tree

**Limitations**

* The left child and right child pointers of the leaf nodes never point to any subtree, and thus lead to wastage of space

**Uses**

* The program can be used to build a binary search tree with integer data, it is a very efficient datastructure specifically for searching a integer data from a large number of nodes. For example, it can be used to implement a database which requires frequent searching of data.

**Future Scope**

* The lchild and rchild pointers of the leaf nodes can be pointed to the root node of the tree to facilitate circular traversal.

**Teacher’s Signature**