

GOVERNMENT POLYTECHNIC, AMRAVATI

(An Autonomous Institute of Government of Maharashtra)

NBA Accredited Institute

Certificate



Name of Department: Computer Science and Engineering.

This is to certify that **Mr. Ayush Shashikant Bulbule** Identity Code **19CM007** has completed the practical work of the course **CM3407 Data Structure Using C** during the Academic year 2020-21.

Signature of the Teacher

Date:

who taught the examinee

Head of Department

Index

S.No	Name of Experiment	Date	Page	Remarks
1	Implement a 'C' program for performing following operations on Array: Creation, Insertion, Deletion, Display	20-9-2020	3	
2	Implement a 'C' program to search a particular data from the given Array using Linear Search.	24-9-2020	6	
3	Implement a 'C' program to search a particular data from the given Array using Binary Search	26-9-2020	8	
4	Implement a 'C' program to sort an array using Bubble Sort	28-9-2020		
5	Implement a 'C' program to sort an array using Selection Sort	30-9-2020		
6	Implement a 'C' program to sort an array using Insertion Sort	2-10-2020		
7	Write C program to Implement Merge Sort.	4-10-2020		
8	Write C program to Implement Quick Sort Algorithm	6-10-2020		
9	Write C program to perform PUSH and POP operations on stack using array.	15-10-2020		
10	Write C program to perform INSERT and DELETE operations on queue using array.	20-10-2020		
11	Write C program to reverse a list of given numbers	30-10-2020		
12	Write C program to implement a Circular Queue	2-11-2020		
13	Write C program to implement double ended queue (Deque)	4-11-2020		
14	Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Singly Linked List. Part – I	5-11-2020		
15	Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Circular Singly Linked List. Part – I	20-11-2020		
16	Write a C Program to design a stack using Linked List.	25-11-2020		
17	Write C program to implement Queue using Linked List	27-11-2020		
18	Write C program to Implement BST (Binary Search Tree) and traverse the tree (Inorder, Preorder, Post order).	13-12-2020		
19	Write C program to calculate height of the given Binary Tree.	15-12-2020		
20	Write C program to calculate number of nodes in Binary Search Tree	18-12-2020		

21	Write C program to find out largest nodes in a Binary Search Tree	22-12-2020		
22	Write C program to create a Graph of n vertices using an adjacency list. Also write code to read and print its information and finally to delete a desired node.	24-12-2020		
23	Write C program to implement the Breadth First Search algorithm	8-01-2021		
24	Write C program to implement the Depth First Search algorithm	10-01-2021		

Practical no 1.

Aim: Implement a 'C' program for performing following operations on Array: Creation, Insertion, Deletion, Display.

Theory:

Array is a container which can hold a certain items of same data type. Most of the data structures make use of arrays to implement their algorithm. Various operations can be done on Array. Some of them are Creation, Insertion, Deletion and Display. The Following Practical Demonstrates it:-

Algorithm:

Algorithm for Insertion operation :-

Step 1 : Start.

Step 2 : Input size and elements in array. Store it in some variable say size and arr.

Step 3 : Input a new element and position to insert in the array. Store it in some variable say num and pos.

Step 4 : To insert a new element in an array, shift elements from the given insert position to one position right. Hence, run a loop in descending order from size to pos to insert. The loop structure should look like for($i=size$; $i \geq pos$; $i--$).

Step 5 : Inside the loop copy previous element to current element by $arr[i] = arr[i - 1]$;

Step 6 : Finally, after performing shift operation. Copy the new element at its specified position i.e. $arr[pos - 1] = num$.

Step 7 : Exit.

Algorithm for Deletion operation :-

Step 1: Start.

Step 2: Input size and elements in array. Store it in some variable say size and arr.

Step 3: Input position to delete an element in the array. Store it in some variable say pos.

Step 4: Move to the specified location which you want to remove in the given array.

Step 5: Copy the next element to the current element of the array. Which is you need to perform $array[i] = array[i + 1]$.

Step 6: Repeat above steps till last element of array.

Step 7: Finally decrement the size of the array by one.

Step 8: Exit

Program:

```
#include <stdio.h>
#include <conio.h>
#define MAX_SIZE 100

//Function to print array
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf(" %d ", arr[i]);
    }
    printf("\n");
}

//Function to Insert new Element in array
int insertElement(int arr[], int num, int pos, int size)
{
    //If position is not valid
    if (pos > size + 1 || pos < 0)
    {
        printf("Position is not Valid please enter a position between
n 0 to %d", size);
    }
    else
    {
        //Making Room (Space For New Element at given position)
        for (int i = size; i >= pos; i--)
        {
            arr[i] = arr[i - 1];
        }
        //Insert element at given position
        arr[pos - 1] = num;
        return size + 1;
    }
}

//finding element in array
int findElement(int arr[], int num, int size)
{
    int i;
    for (i = 0; i < size; i++)
```

```

        if (arr[i] == num)
            return i;

    return -1;
}
//function to delete an element from array
int deleteElement(int arr[], int num, int size)
{
    int pos;
    pos = findElement(arr, num, size);
    if (pos == -1)
    {
        printf("Element not found!!");
        return size;
    }
    else
    {
        for (int i = pos; i < size; i++)
        {
            arr[i] = arr[i + 1];
        }
        return size - 1;
    }
}

int main()
{
    int arr[MAX_SIZE];
    int i, size, num, pos;

    // To Get size of array from user

    printf("Enter size of array: ");
    scanf("%d", &size);

    // Creating Array of Preferred Size
    for (i = 0; i < size; i++)
    {
        printf("Plese enter Elements in array: ");
        scanf("%d", &arr[i]);
    }
}

```

```

    printArray(arr, size); //Func call to print array

    printf("Element an elemet to be inserted: ");
    scanf("%d", &num);

    printf("Enter position: ");
    scanf("%d", &pos);

    size = insertElement(arr, num, pos, size);

    printf("Array after insertion: ");
    printArray(arr, size);

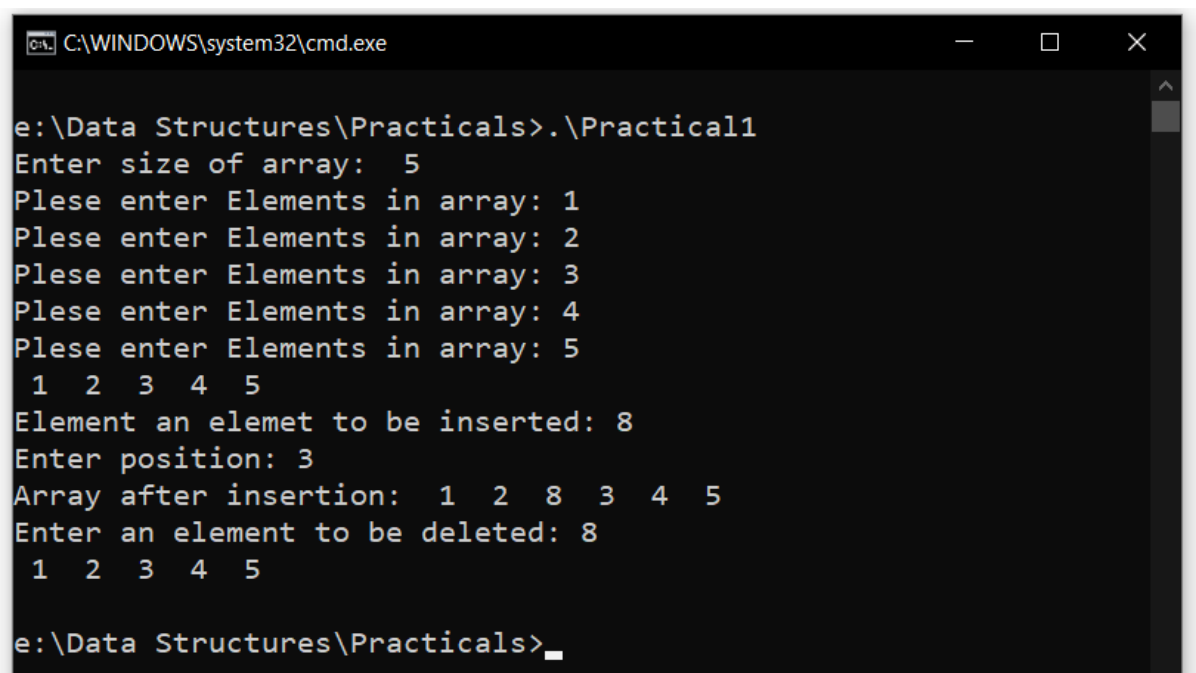
    //To delete an element
    printf("Enter an element to be deleted: ");
    scanf("%d", &num);

    size = deleteElement(arr, num, size);
    printArray(arr, size);

    return 0;
}
//19CM007

```

Output:



```

C:\WINDOWS\system32\cmd.exe
e:\Data Structures\Practicals>.\Practical1
Enter size of array: 5
Plese enter Elements in array: 1
Plese enter Elements in array: 2
Plese enter Elements in array: 3
Plese enter Elements in array: 4
Plese enter Elements in array: 5
1 2 3 4 5
Element an elemet to be inserted: 8
Enter position: 3
Array after insertion: 1 2 8 3 4 5
Enter an element to be deleted: 8
1 2 3 4 5
e:\Data Structures\Practicals>_

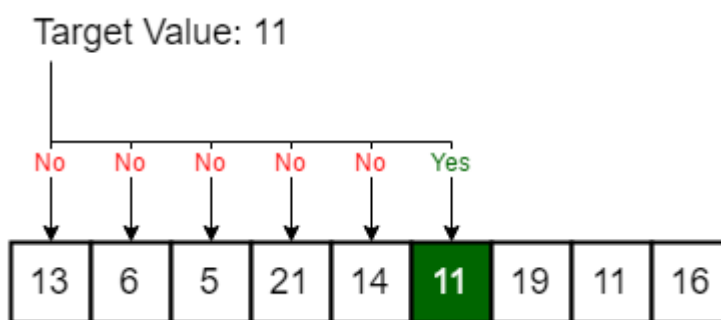
```

Practical no 2.

Aim: Implement a 'C' program to search a particular data from the given Array using Linear Search.

Theory:

Linear search is a simple searching algorithm which uses sequential. In Linear Search all elements of the array are checked sequentially and when the element is found it is returned with its location. The Following Practical Demonstrates it:-



Algorithm:

Step 1 : Let a is an array with n elements. To search data item variable item is used.

Step 2 : Read n numbers and store them in an array.

Step 3 : Input item to search.

Step 4 : Repeat for (i =0;i<=9;i++)

Step 5 : If (item==a[i]), then :

Write: "Item found at location"; break;

Step 6 : If(i>9), then :

Write:" Item not exist".

Step 7 : Exit.

Program:

```
//Implement a 'C' program to search a particular data from the given  
Array using: (i)Linear Search */
```

```
#include <stdio.h>  
#include <conio.h>  
#define MAX 100
```



```

//function to perform linear search
int linearSearch(int arr[], int n, int key)
{
    int result;
    for (int i = 0; i < n; i++)
        if (arr[i] == key)
            return i;
    return -1;
}

//Function to print array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main()
{
    int arr[MAX];
    int n, size, result;

    printf("Enter the Number of Elements in Array: ");
    scanf("%d", &size);

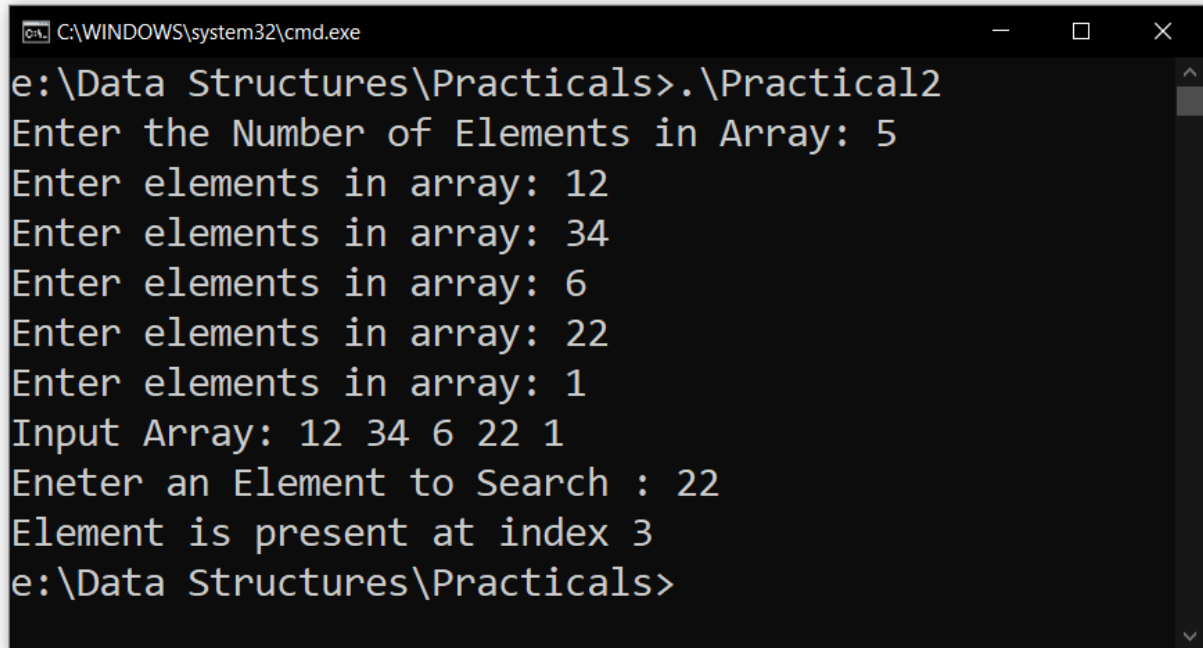
    for (int i = 0; i < size; i++)
    {
        printf("Enter elements in array: ");
        scanf("%d", &arr[i]);
    }
    printf("Input Array: ");
    printArray(arr, size);

    printf("Eneter an Element to Search : ");
    scanf("%d", &n);
    result = linearSearch(arr, size, n);
    (result == -
1) ? printf("Element not Found in Array") : printf("Element is prese
nt at index %d", result);

```

```
    return 0;  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
e:\Data Structures\Practicals>.\Practical2  
Enter the Number of Elements in Array: 5  
Enter elements in array: 12  
Enter elements in array: 34  
Enter elements in array: 6  
Enter elements in array: 22  
Enter elements in array: 1  
Input Array: 12 34 6 22 1  
Enter an Element to Search : 22  
Element is present at index 3  
e:\Data Structures\Practicals>
```

Practical no 3.

Aim: Implement a 'C' program to search a particular data from the given Array using Binary Search.

Theory:

Binary Search is a Fast Search Algorithm. Binary Search runs on the rule of divide and conquer. For this Algorithm the data must be Sorted. In Binary Search the element to be found is compared with the middle most element in the collection. If it is greater than the middle most the element is searched in the sub-array to the right side array and compared with the middle element of sub-array. This process goes on until the element is found or the size of sub-array becomes zero. The Following Practical Demonstrates it:-



Algorithm:

Step 1 : Let the first number be first and last number be last and middle number be middle.

Step 2 : Read n numbers and store it in an array.

Step 3 : Input search to find the value.

Step 4 : Then $\text{first}=0$, $\text{last}=n-1$ and $\text{middle}=(\text{first}+\text{last}) / 2$.

Step 5 : Repeat while ($\text{first} \leq \text{last}$)

Step 6 : If ($\text{array}[\text{middle}] < \text{search}$) then ($\text{first} < \text{last}$)

Step 7 : Else:if($\text{array}[\text{middle}] == \text{search}$)

Step 8 : Else($\text{last} = \text{middle}-1$)

$\text{middle} = (\text{first} + \text{last}) / 2$

Step 9 : If (first>last) the write:"Not found"

Step 10 : Exit.

Program:

//Implement a 'C' program to search a particular data from the given Array using Binary Search

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

int binarySearch(int arr[], int l, int r, int x)
{
    while (l < r)
    {
        int m = l + (r - l) / 2;

        if (arr[m] == x)
            return m;
        if (arr[m] < x)
            l = m + 1;
        else
        {
            r = m - 1;
        }
    }
    return -1;
}

//Function to print array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf(" %d ", arr[i]);
    }
    printf("\n");
}

int main()
{
```

```

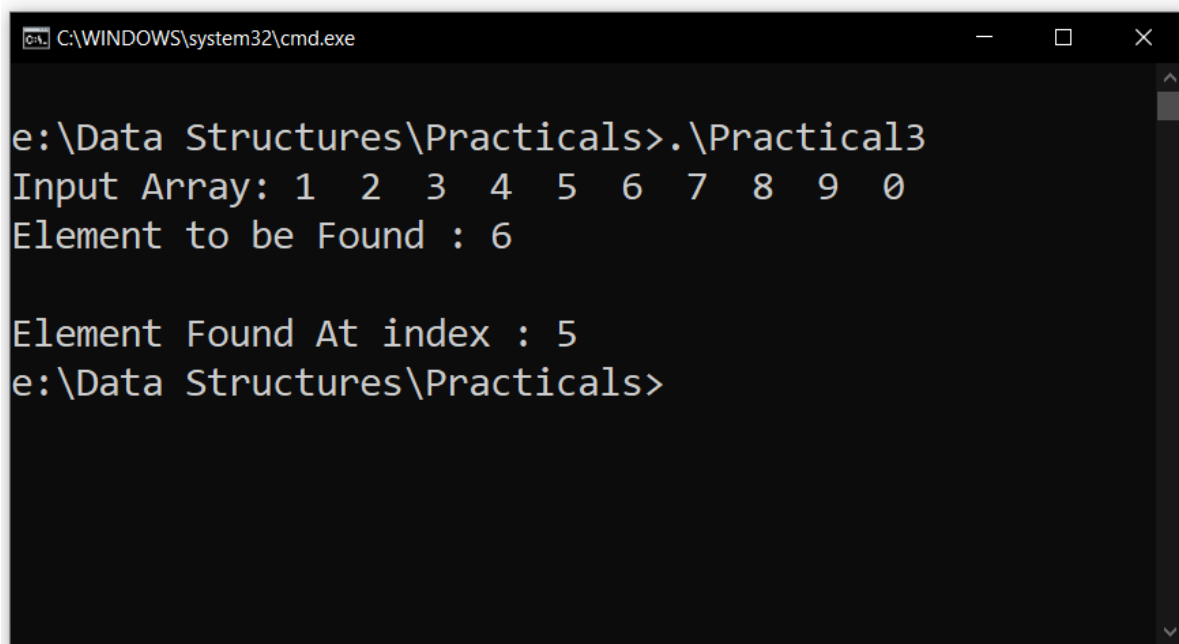
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}; //array
int n = sizeof(arr) / sizeof(arr[0]);      //number of elements
int x = 6;

printf("Input Array:");
printArray(arr, n);
printf("Element to be Found : %d\n\n", x);
int result = binarySearch(arr, 0, n, x); //Element to be found

(result == -
1) ? printf("Element Not Found!!") : printf("Element Found At index
: %d", result);
}
//19CM007

```

Output:



```

C:\WINDOWS\system32\cmd.exe
e:\Data Structures\Practicals>.\Practical3
Input Array: 1 2 3 4 5 6 7 8 9 0
Element to be Found : 6

Element Found At index : 5
e:\Data Structures\Practicals>

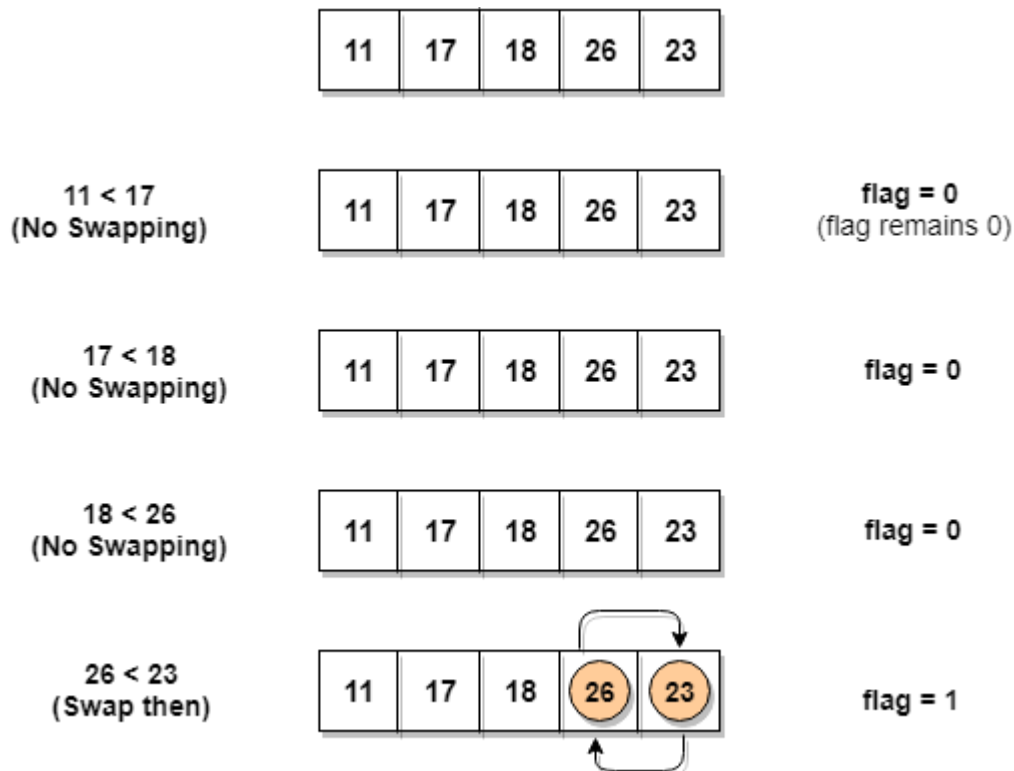
```

Practical no 4.

Aim: Implement a 'C' program to sort an array using Bubble Sort.

Theory:

Bubble Sort is a Simple Sorting algorithm. This algorithm is comparison based algorithm in which adjacent pair of element is compared and swapped if they are not in order. This algorithm is not suitable for large data.



Algorithm:

Step 1 : Start.

Step 2 : Read count numbers and store it in an array.

Step 3 : Repeat for($i = \text{count} - 2$; $i \geq 0$; $i--$)

Step 4 : Repeat for($j = 0$; $j \leq i$; $j++$)

Step 5 : If($\text{number}[j] > \text{number}[j+1]$)

Step 6 : Then $\text{temp} = \text{number}[j]$

$\text{number}[j] = \text{number}[j+1]$

$\text{number}[j+1] = \text{temp}$;

Step 7 : Write the sorted elements.

Step 8 : Exit.

Program:

//Implement a 'C' program to sort an array using Bubble Sort.

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

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void bubbleSort(int arr[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        for (int j = 0; j < size - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}

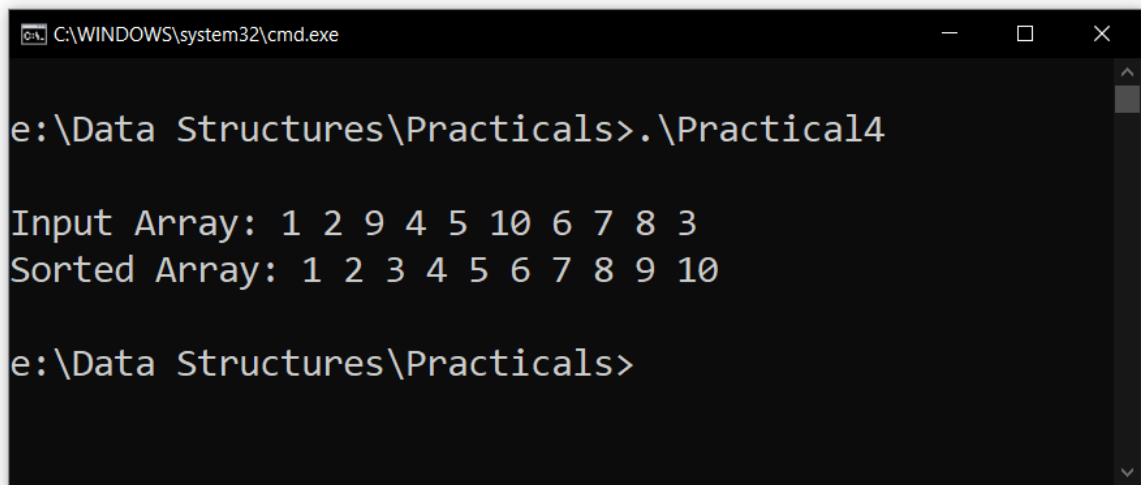
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main()
{
    int arr[] = {1, 2, 9, 4, 5, 10, 6, 7, 8, 3};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("\nInput Array: ");
```

```
    printArray(arr, size);  
  
    bubbleSort(arr, size);  
    printf("Sorted Array: ");  
    printArray(arr, size);  
}
```

Output:



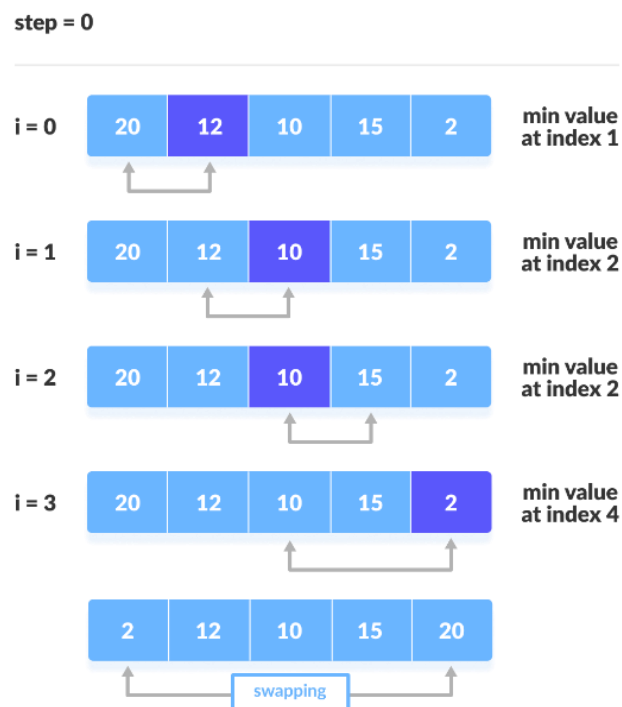
The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt is at "e:\Data Structures\Practicals>". The user has entered ".\Practical4", which has executed a program. The program's output is displayed on two lines: "Input Array: 1 2 9 4 5 10 6 7 8 3" and "Sorted Array: 1 2 3 4 5 6 7 8 9 10". The prompt is now at "e:\Data Structures\Practicals>" again.

Practical no 5.

Aim: Implement a 'C' program to sort an array using Selection Sort.

Theory:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Sorting works like it finds the minimum element in the unsorted part and swap it with the beginning.



Algorithm:

Step 1 : Start.

Step 2 : Read the length of the array.

Step 3 : Read the numbers.

Step 4 : Search the smallest number.

Step 5 : Repeat the loop until the array sorts.

Step 6 : Exit.

Program:

//Implement a 'C' program to sort an array using Selection Sort

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

void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

void selectionSort(int arr[], int size)
{
    int indexofMin, temp;
    for (int i = 0; i < size - 1; i++)
    {
        indexofMin = i;

        for (int j = i + 1; j < size; j++)
        {
            if (arr[j] < arr[indexofMin])
            {
                indexofMin = j;
            }

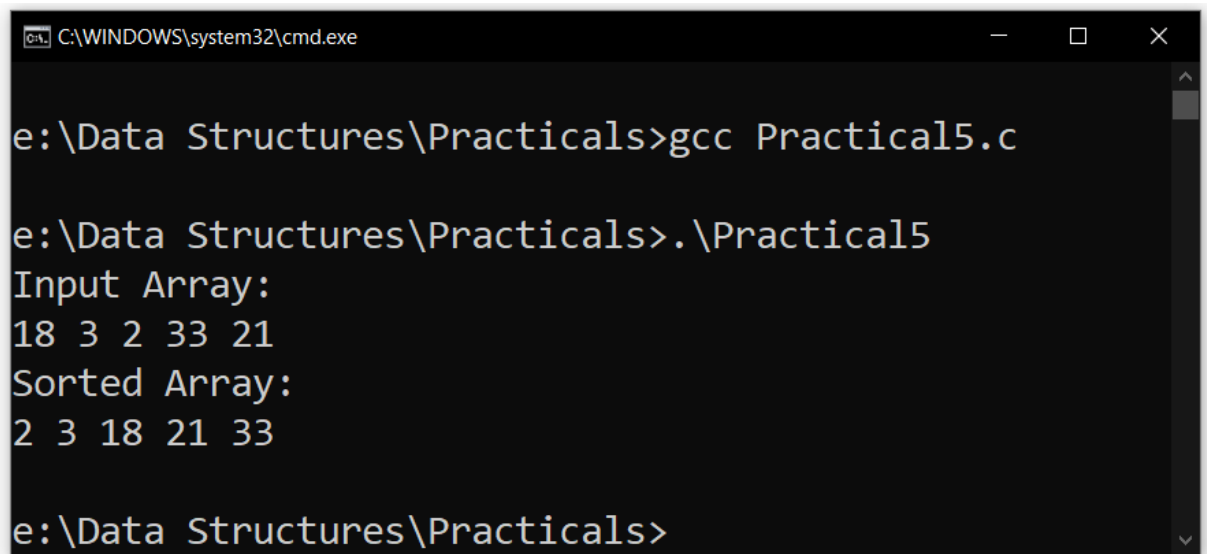
            temp = arr[i];
            arr[i] = arr[indexofMin];
            arr[indexofMin] = temp;
        }
    }
}

int main()
{
    int arr[] = {18, 3, 2, 33, 21};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Input Array: \n");
    printArray(arr, size);
}
```

```
    selectionSort(arr, size);  
    printf("Sorted Array: \n");  
  
    printArray(arr, size);  
  
    return 0;  
}
```

Output:



The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe". The prompt is at "e:\Data Structures\Practicals>". The user enters "gcc Practical5.c" and presses Enter. The prompt changes to "e:\Data Structures\Practicals>.\Practical5". The program outputs "Input Array:" followed by "18 3 2 33 21" on the next line. Then it outputs "Sorted Array:" followed by "2 3 18 21 33" on the next line. The prompt returns to "e:\Data Structures\Practicals>".

```
C:\WINDOWS\system32\cmd.exe  
e:\Data Structures\Practicals>gcc Practical5.c  
e:\Data Structures\Practicals>.\Practical5  
Input Array:  
18 3 2 33 21  
Sorted Array:  
2 3 18 21 33  
e:\Data Structures\Practicals>
```

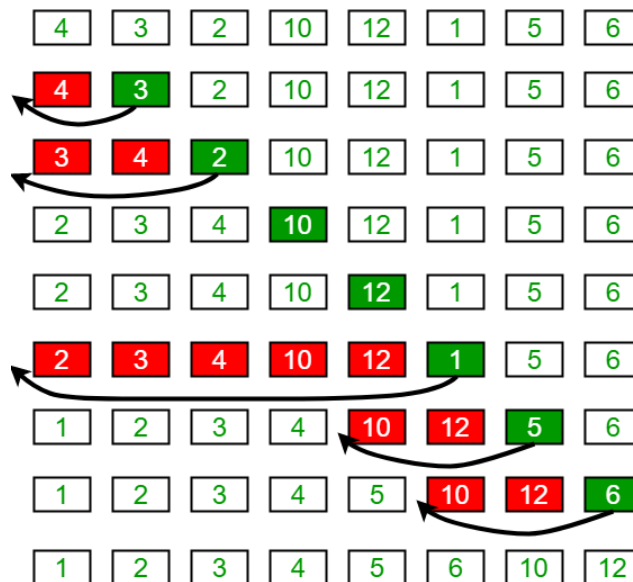
Practical no 6.

Aim: Implement a 'C' program to sort an array using Insertion Sort.

Theory:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Sorting works like it finds the minimum element in the unsorted part and swap it with the beginning.

Insertion Sort Execution Example



Algorithm:

Step 1 : Start.

Step 2 : Enter the number of elements .

Step 3 : Enter the numbers.

Step 4 : If the element is the first one, it is already sorted.

Step 5 : Move to the next element.

Step 6 : Compare the current element with all elements in the sorted array.

Step 7 : If the element in the sorted array is smaller then the current element, iterate to the next element. Otherwise, shift all the greater elements in the array by one position towards the right.

Step 8 : Insert the value at the correct position.

Step 9 : Repeat until the complete list is sorted.

Step 10 : Print the sorted elements.

Step 11 : Stop.

Program:

```
#include <stdio.h>
int main()
{
    int n, i, j, temp;
    int arr[30];

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    for (i = 1; i <= n - 1; i++)
    {
        j = i;
        while (j > 0 && arr[j - 1] > arr[j])
        {
            temp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = temp;
            j--;
        }
    }
    printf("Sorted list in ascending order:\n");
    for (i = 0; i <= n - 1; i++)
    {
        printf("%d\n", arr[i]);
    }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>gcc Practical6.c

E:\Data Structures\Practicals>.\a
Enter number of elements
5
Enter 5 integers
23 56 32 12 68
Sorted list in ascending order:
12 23 32 56 68
E:\Data Structures\Practicals>
```

Practical no 7.

Aim: Write C program to Implement Merge Sort

Theory:

Merge sort is a sorting technique based on divide and conquer technique. With the worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms.

Algorithm:

Step 1 : MergeSort(arr[], l, r), where l is the index of the first element & r is the index of the last element.

Step 2 : If $r > l$

Find the middle index of the array to divide it in two halves:

$m = (l+r)/2$

Step 3 : Call MergeSort for first half:

mergeSort(array, l, m)

Step 4 : Call mergeSort for second half:

mergeSort(array, m+1, r)

Step 5 : Recursively, merge the two halves in a sorted manner, so that only one sorted array is left:

merge(array, l, m, r)

Step 6 : Stop.

Program:

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

```
void merge_sort(int i, int j, int a[], int aux[])
```

```
{
    if (j <= i)
    {
        return;
    }
    int mid = (i + j) / 2;
    merge_sort(i, mid, a, aux);
    merge_sort(mid + 1, j, a, aux);
}
```

```

int pointer_left = i;
int pointer_right = mid + 1;
int k;
for (k = i; k <= j; k++)
{
    if (pointer_left == mid + 1)
    {
        aux[k] = a[pointer_right];
        pointer_right++;
    }
    else if (pointer_right == j + 1)
    {
        aux[k] = a[pointer_left];
        pointer_left++;
    }
    else if (a[pointer_left] < a[pointer_right])
    {
        aux[k] = a[pointer_left];
        pointer_left++;
    }
    else
    {
        aux[k] = a[pointer_right];
        pointer_right++;
    }
}

for (k = i; k <= j; k++)
{
    a[k] = aux[k];
}
}

int main()
{
    int a[100], aux[100], n, i, d, swap;

    printf("Enter number of elements in the array:\n");
    scanf("%d", &n);

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

```



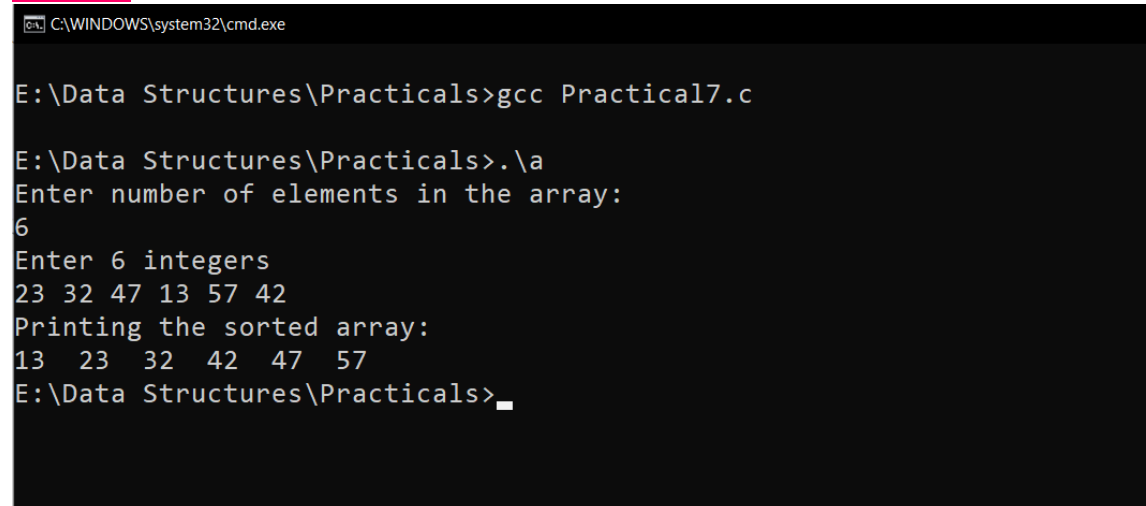
```
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    merge_sort(0, n - 1, a, aux);

    printf("Printing the sorted array:\n");

    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>gcc Practical7.c

E:\Data Structures\Practicals>.\a
Enter number of elements in the array:
6
Enter 6 integers
23 32 47 13 57 42
Printing the sorted array:
13 23 32 42 47 57
E:\Data Structures\Practicals>_
```

Practical no 8.

Aim: Write C program to Implement Quick Sort Algorithm

Theory:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Algorithm:

Step 1 : Start.

Step 2 : Enter the number of elements.

Step 3 : Enter the number.

Step 4 : Pick an element from the array, this element is called a pivot element.

Step 5 : Divide the unsorted array of elements in two arrays with values less than the pivot come in the first sub array, while all elements with values greater than the pivot come in the second sub-array (equal values can go either way). This step is called the partition operation.

Step 6 : Recursively repeat the step 5(until the sub-arrays are sorted) to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Step 7 : Print the sorted elements.

Step 8 : Stop.

Program:

```
#include <stdio.h>
void quicksort(int number[25], int first, int last)
{
    int i, j, pivot, temp;

    if (first < last)
    {
```

```

    pivot = first;
    i = first;
    j = last;

    while (i < j)
    {
        while (number[i] <= number[pivot] && i < last)
            i++;
        while (number[j] > number[pivot])
            j--;
        if (i < j)
        {
            temp = number[i];
            number[i] = number[j];
            number[j] = temp;
        }
    }
    temp = number[pivot];
    number[pivot] = number[j];
    number[j] = temp;
    quicksort(number, first, j - 1);
    quicksort(number, j + 1, last);
}

int main()
{
    int i, count, number[25];

    printf("How many elements are u going to enter?: ");
    scanf("%d", &count);

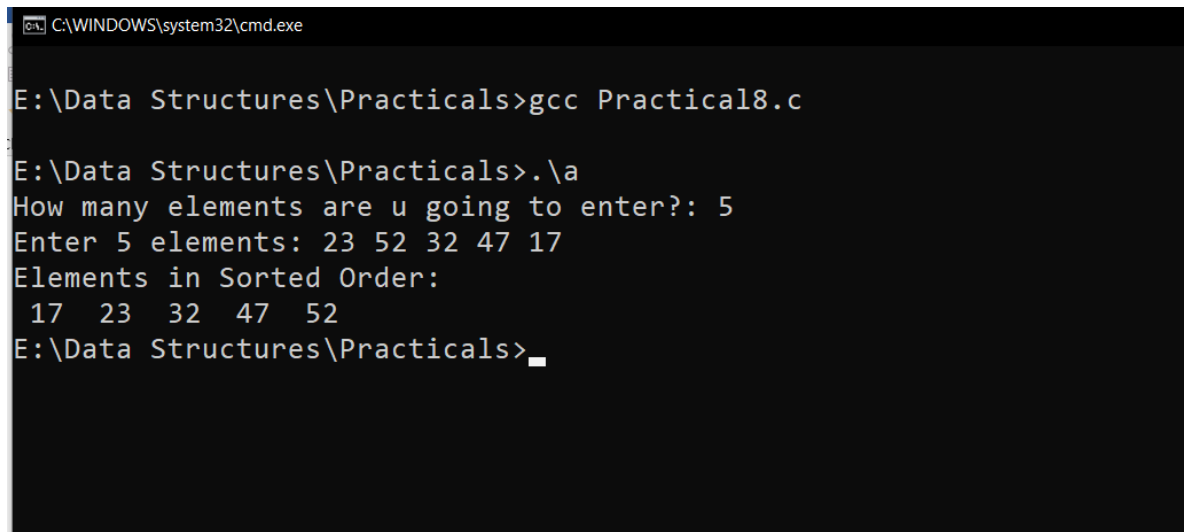
    printf("Enter %d elements: ", count);
    for (i = 0; i < count; i++)
        scanf("%d", &number[i]);

    quicksort(number, 0, count - 1);

    printf("Elements in Sorted Order:\n");
    for (i = 0; i < count; i++)
        printf(" %d ", number[i]);
}

```

Output:



```
C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>gcc Practical8.c

E:\Data Structures\Practicals>.\a
How many elements are u going to enter?: 5
Enter 5 elements: 23 52 32 47 17
Elements in Sorted Order:
17 23 32 47 52
E:\Data Structures\Practicals>_
```

Practical no 9.

Aim: Write C program to perform PUSH and POP operations on stack using array.

Theory:

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

push() – Pushing (storing) an element on the stack.

pop() – Removing (accessing) an element from the stack.

Algorithm:

Step 1 : Start.

Step 2 : Check if the stack is full.

Step 3 : If the stack is full, then display "Stack overflow".

Step 4 : If the stack is not full, increment top to the next location.

Step 5 : Assign data to the top element.

Step 6 : Exit.

pop()

Step 1 : Start.

Step 2 : Check if the stack is empty

Step 3 : If the stack is empty, then display "Stack Underflow".

Step 4 : If the stack is not empty, copy top in a temporary variable.

Step 5 : Decrement top to the previous location.

Step 6 : Delete the temporary variable.

Step 7 : Stop.

Program:

//Write C program to perform PUSH and POP operations on stack using array

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

struct stack
{
    int top;
    int size;
    int *arr;
};

int push(struct stack *s, int value)
{
    if (s->top == s->size)
    {
        printf("Stack Full!!");
        return 0;
    }
    else
    {
        s->top = s->top + 1;
        s->arr[s->top] = value;
        return 1;
    }
}

int pop(struct stack *s)
{
    if (s->top == -1)
    {
        printf("Stack Empty!!");
        return 0;
    }
    else
    {
        int value = s->arr[s->top];
        s->top = s->top - 1;
        s->size=s->size-1;
    }
}
```

```

        return value;
    }
}
int main()
{
    int i;
    struct stack *s;

    s->top = -1;
    s->size = 10;
    s->arr = (int *)malloc(s->size * sizeof(int));

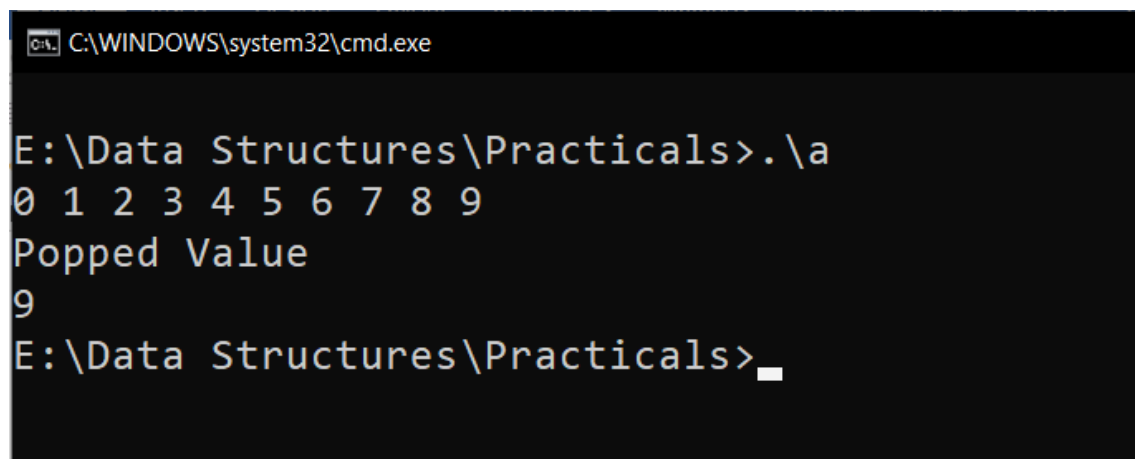
    for (int i = 0; i < s->size; i++)
    {
        push(s, i);
    }
    for (int i = 0; i < s->size; i++)
    {
        printf("%d ", s->arr[i]);
    }

    printf("\nPopped Value");
    int val = pop(s);
    printf("\n%d ", val);

    return 0;
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>.\a
0 1 2 3 4 5 6 7 8 9
Popped Value
9
E:\Data Structures\Practicals>

```

Practical no 10.

Aim: Write C program to perform INSERT and DELETE operations on queue using array

Theory:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

enqueue() – add (store) an item to the queue.

dequeue() – remove (access) an item from the queue.

Algorithm:

Step 1: Create a one dimensional array with above defined SIZE (int queue[SIZE])

Step 2 : Define two integer variables 'front' and 'rear' and initialize both with '-1'. (int front = -1, rear = -1).

Step 3 : Then implement the main method by displaying a menu of operations list and make suitable function calls to perform operations selected by the user on the queue.

Inserting value in the queue

Step 1 : Check whether the queue is FULL. (rear == SIZE-1)

Step 2 : If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3 : If it is NOT FULL, then increment rear value by one (rear++) and set queue[rear] = value.

Step 1 : Check whether the queue is EMPTY. (front == rear)

Step 2 : If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3 : If it is NOT EMPTY, then increment the front value by one (front ++). Then display queue[front] as a deleted element. Then check whether both front and rear are equal (front == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Delete a value from the queue

Step 1 : Check whether the queue is EMPTY. (front == rear)

Step 2 : If it is EMPTY, then display "Queue is EMPTY!!!" and terminate the function.

Step 3 : If it is NOT EMPTY, then define an integer variable 'i' and set 'i = front+1'.

Step 4 : Display 'queue[i]' value and increment 'i' value by one (i++). Repeat the same until 'i' value reaches to rear (i <= rear)

Program:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void enQueue(int value)
{
    if (rear == SIZE - 1)
        printf("\nQueue is Full!!! Insertion is not possible!!!");
    else
    {
        if (front == -1)
            front = 0;
        rear++;
        queue[rear] = value;
        printf("\nInsertion success!!!");
    }
}

void deQueue()
{
    if (front == rear)
        printf("\nQueue is Empty!!! Deletion is not possible!!!");
    else
    {
        printf("\nDeleted : %d", queue[front]);
        front++;
        if (front == rear)
            front = rear = -1;
    }
}

void display()
{
    if (rear == -1)
        printf("\nQueue is Empty!!!");
    else
    {
        int i;
        printf("\nQueue elements are:\n");
        for (i = front; i <= rear; i++)
```

```

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

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

main()
{
    int value, choice;

    while (1)
    {
        printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");
        printf("\nOption of your Choise: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                printf("Enter the value to be insert: ");
                scanf("%d", &value);
                enQueue(value);
                break;
            case 2:
                deQueue();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("\nWrong selection!!! Try again!!!");
        }
    }
}

```

Output:

```

#include <stdio.h>
#define SIZE 5

void enQueue(int);

```

```

void deQueue();
void display();

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

int main()
{
    //deQueue is not possible on empty queue
    deQueue();

    //enQueue 5 elements
    enQueue(1);
    enQueue(2);
    enQueue(3);
    enQueue(4);
    enQueue(5);
    enQueue(6);

    display();

    deQueue();

    display();

    return 0;
}

void enQueue(int value)
{
    if (rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else
    {
        if (front == -1)
            front = 0;
        rear++;
        items[rear] = value;
        printf("\nInserted -> %d", value);
    }
}

void deQueue()

```

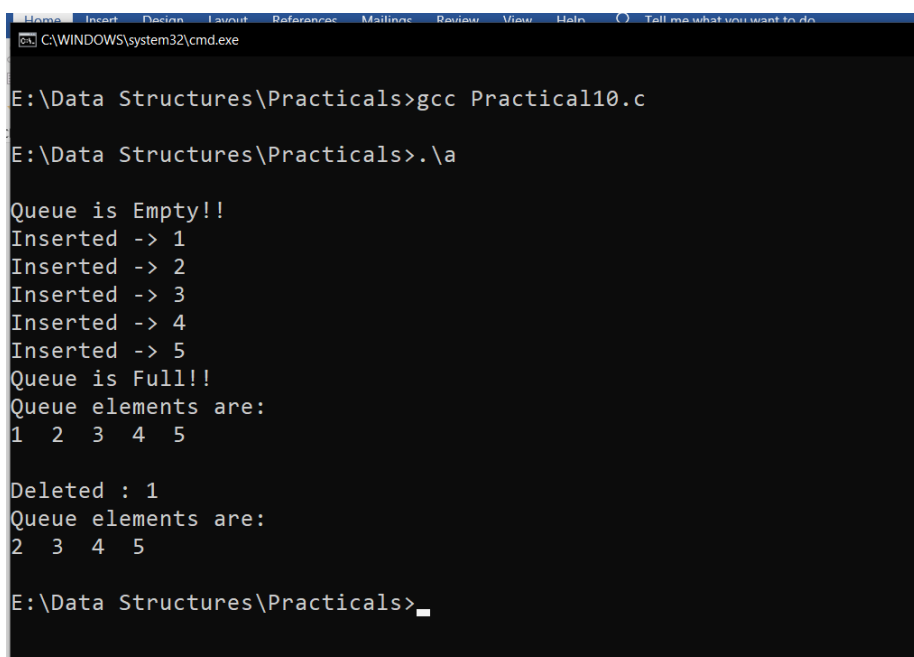
```

{
    if (front == -1)
        printf("\nQueue is Empty!!");
    else
    {
        printf("\nDeleted : %d", items[front]);
        front++;
        if (front > rear)
            front = rear = -1;
    }
}

void display()
{
    if (rear == -1)
        printf("\nQueue is Empty!!!");
    else
    {
        int i;
        printf("\nQueue elements are:\n");
        for (i = front; i <= rear; i++)
            printf("%d  ", items[i]);
    }
    printf("\n");
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>gcc Practical10.c
E:\Data Structures\Practicals>.\a

Queue is Empty!!
Inserted -> 1
Inserted -> 2
Inserted -> 3
Inserted -> 4
Inserted -> 5
Queue is Full!!
Queue elements are:
1 2 3 4 5

Deleted : 1
Queue elements are:
2 3 4 5

E:\Data Structures\Practicals>_

```

Practical no 11.

Aim: Write C program to reverse a list of given numbers

Algorithm

Step 1 : Start.

Step 2 : Read the numbers.

Step 3 : Print numbers by giving reverse conditon in for loop.

Step 4 : Stop.

Program:

```
#include <stdio.h>
main()
{
    int num[50];
    int i, N;

    printf("Enter the number of elements : ");
    scanf("%d", &N);

    printf("Enter the elements : ");
    for (i = 0; i < N; i++)
        scanf("%d", &num[i]);

    printf("Reverse list order : ");
    for (i = N - 1; i >= 0; i--)
        printf("%d\t", num[i]);
}
```

Practical no 12.

Aim Write C program to implement a Circular Queue

Theory:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

enqueue() – add (store) an item to the queue.

dequeue() – remove (access) an item from the queue.

Algorithm:

Step 1 : Start

Step 2 : Check whether queue is Full – Check $((\text{rear} == \text{SIZE}-1 \ \&\& \ \text{front} == 0) \ || \ (\text{rear} == \text{front}-1))$.

Step 3 : If it is full then display Queue is full. If queue is not full then, check if $(\text{rear} == \text{SIZE} - 1 \ \&\& \ \text{front} != 0)$ if it is true then set $\text{rear}=0$ and insert element.

Step 4 : Check whether queue is Empty means check $(\text{front} == -1)$.

Step 5 : If it is empty then display Queue is empty. If queue is not empty then step 3

Step 6 : Check if $(\text{front} == \text{rear})$ if it is true then set $\text{front} = \text{rear} = -1$ else check if $(\text{front} == \text{size}-1)$, if it is true then set $\text{front}=0$ and return the element.

Step 7 : Stop

Program:

```
// Write C program to implement a Circular Queue
```

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

```
struct queue
{
    int size;
    int f;
    int r;
    int *arr;
```

```

};
int isFull(struct queue *q)
{
    if ((q->f == q->r + 1) || (q->f == 0 && q->r == q->size - 1))
        return 1;
    return 0;
}
int isEmpty(struct queue *q)
{
    if (q->f == -1)
        return 1;
    return 0;
}

void enqueue(struct queue *q, int val)
{
    if (isFull(q))
    {
        printf("This q is Full!");
    }
    else
    {
        if (q->f == -1)
            q->f = 0;
        q->r = (q->r + 1) % q->size;
        q->arr[q->r] = val;
        printf("\n Inserted -> %d", val);
    }
}

int dequeue(struct queue *q)
{
    int val;
    if (isEmpty(q))
    {
        printf("\n Queue is empty !! \n");
        return (-1);
    }
    else
    {
        val = q->arr[q->f];
        if (q->f == q->r)
        {

```

```

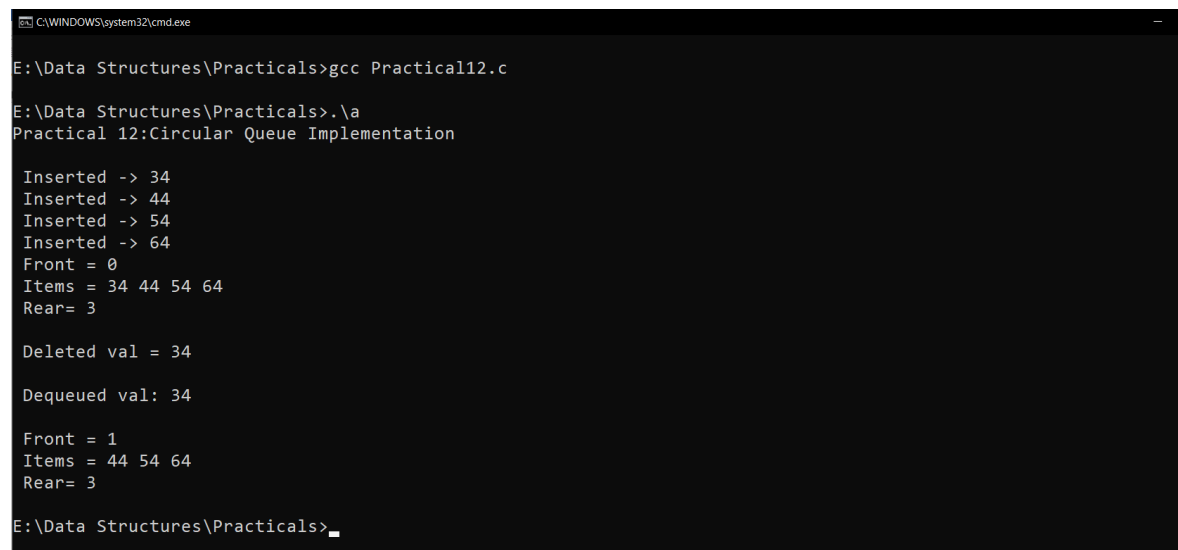
        q->f = -1;
        q->r = -1;
    }
    // Q has only one val, so we reset the
    // queue after dequeing it. ?
    else
    {
        q->f = (q->f + 1) % q->size;
    }
    printf("\n Deleted val = %d \n", val);
    return (val);
}
}
void printQueue(struct queue *q)
{
    int i;
    if (isEmpty(q))
        printf(" \n Empty Queue\n");
    else
    {
        printf("\n Front = %d ", q->f);
        printf("\n Items = ");
        for (i = q->f; i != q->r; i = (i + 1) % q->size)
        {
            printf("%d ", q->arr[i]);
        }
        printf("%d ", q->arr[i]);
        printf("\n Rear= %d \n", q->r);
    }
}
}
int main()
{
    struct queue q;
    q.f = q.r = -1;
    q.arr = (int *)malloc(q.size * sizeof(int));
    printf("Practical 12: Circular Queue Implementation\n");
    enqueue(&q, 34);
    enqueue(&q, 44);
    enqueue(&q, 54);
    enqueue(&q, 64);
    printQueue(&q);
}

```



```
    int val = dequeue(&q);  
    printf("\n Dequeued val: %d\n", val);  
    printQueue(&q);  
  
    return 0;  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
E:\Data Structures\Practicals>gcc Practical12.c  
E:\Data Structures\Practicals>.\a  
Practical 12:Circular Queue Implementation  
  
Inserted -> 34  
Inserted -> 44  
Inserted -> 54  
Inserted -> 64  
Front = 0  
Items = 34 44 54 64  
Rear= 3  
  
Deleted val = 34  
  
Dequeued val: 34  
  
Front = 1  
Items = 44 54 64  
Rear= 3  
E:\Data Structures\Practicals>_
```

Practical no 13.

Aim: Write C program to implement double ended queue (Deque)

Theory:

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).

Algorithm:

1. Insert at the Front

This operation adds an element at the front.

Step 1 : Check the position of front.

Step 2 : If $\text{front} < 1$, reinitialize $\text{front} = n-1$ (last index).

Step 3 : Else, decrease front by 1.

Step 4 : Add the new key 5 into $\text{array}[\text{front}]$.

2. Insert at the Rear

This operation adds an element to the rear.

Step 1 : Check if the array is full.

Step 2 : If the deque is full, reinitialize $\text{rear} = 0$.

Step 3 : Else, increase rear by 1

Step 4 : Add the new key 5 into $\text{array}[\text{rear}]$.

3. Delete from the Front

Step 1 : Check if the deque is empty.

Step 2 : If the deque is empty (i.e. $\text{front} = -1$), deletion cannot be performed (underflow condition).

Step 3 : If the deque has only one element (i.e. $\text{front} = \text{rear}$), set $\text{front} = -1$ and $\text{rear} = -1$.

Step 4 : Else if front is at the end (i.e. $\text{front} = n - 1$), set go to the front $\text{front} = 0$.

Step 5 : Else, $\text{front} = \text{front} + 1$.

4. Delete from the Rear

This operation deletes an element from the rear.

Step 1 : Check if the deque is empty.

Step 2 : If the deque is empty (i.e. $\text{front} = -1$), deletion cannot be performed (underflow condition).

Step 3 : If the deque has only one element (i.e. $\text{front} = \text{rear}$), set $\text{front} = -1$ and $\text{rear} = -1$, else follow the steps below.

Step 4 : If rear is at the front (i.e. $\text{rear} = 0$), set go to the front $\text{rear} = n - 1$.

Step 5 : Else, rear = rear - 1.

5. Check Empty

a) This operation checks if the deque is empty. If front = -1, the deque is empty.

6. Check Full

b) This operation checks if the deque is full. If front = 0 and rear = n - 1 OR front = rear + 1, the deque is full.

Program:

```
//Write C program to implement double ended queue (Dequeue)
// Deque implementation in C
```

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

```
#define MAX 10
```

```
int count(int *arr)
```

```
{
```

```
    int c = 0, i;
```

```
    for (i = 0; i < MAX; i++)
```

```
    {
```

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

```
            c++;
```

```
    }
```

```
    return c;
```

```
}
```

```
void addFront(int *arr, int item, int *pfront, int *prear)
```

```
{
```

```
    int i, k, c;
```

```
    if (*pfront == 0 && *prear == MAX - 1)
```

```
    {
```

```
        printf("\nDeque is full.\n");
```

```
        return;
```

```
    }
```

```
    if (*pfront == -1)
```

```
    {
```

```
        *pfront = *prear = 0;
```

```

        arr[*pfront] = item;
        return;
    }

    if (*prear != MAX - 1)
    {
        c = count(arr);
        k = *prear + 1;
        for (i = 1; i <= c; i++)
        {
            arr[k] = arr[k - 1];
            k--;
        }
        arr[k] = item;
        *pfront = k;
        (*prear)++;
    }
    else
    {
        (*pfront)--;
        arr[*pfront] = item;
    }
}

void addRear(int *arr, int item, int *pfront, int *prear)
{
    int i, k;

    if (*pfront == 0 && *prear == MAX - 1)
    {
        printf("\nDeque is full.\n");
        return;
    }

    if (*pfront == -1)
    {
        *prear = *pfront = 0;
        arr[*prear] = item;
        return;
    }

    if (*prear == MAX - 1)

```

```

    {
        k = *pfront - 1;
        for (i = *pfront - 1; i < *prear; i++)
        {
            k = i;
            if (k == MAX - 1)
                arr[k] = 0;
            else
                arr[k] = arr[i + 1];
        }
        (*prear)--;
        (*pfront)--;
    }
    (*prear)++;
    arr[*prear] = item;
}

int delFront(int *arr, int *pfront, int *prear)
{
    int item;

    if (*pfront == -1)
    {
        printf("\nDeque is empty.\n");
        return 0;
    }

    item = arr[*pfront];
    arr[*pfront] = 0;

    if (*pfront == *prear)
        *pfront = *prear = -1;
    else
        (*pfront)++;

    return item;
}

int delRear(int *arr, int *pfront, int *prear)
{
    int item;

```

```

    if (*pfront == -1)
    {
        printf("\nDeque is empty.\n");
        return 0;
    }

    item = arr[*prear];
    arr[*prear] = 0;
    (*prear)--;
    if (*prear == -1)
        *pfront = -1;
    return item;
}

void display(int *arr)
{
    int i;

    printf("\n front: ");
    for (i = 0; i < MAX; i++)
        printf(" %d", arr[i]);
    printf(" :rear");
}

int main()
{
    int arr[MAX];
    int front, rear, i, n;

    front = rear = -1;
    for (i = 0; i < MAX; i++)
        arr[i] = 0;

    addRear(arr, 5, &front, &rear);
    addFront(arr, 12, &front, &rear);
    addRear(arr, 11, &front, &rear);
    addFront(arr, 5, &front, &rear);
    addRear(arr, 6, &front, &rear);
    addFront(arr, 8, &front, &rear);

    printf("\nElements in a deque: ");
    display(arr);
}

```

```

    i = delFront(arr, &front, &rear);
    printf("\nremoved item: %d", i);

    printf("\nElements in a deque after deletion: ");
    display(arr);

    addRear(arr, 16, &front, &rear);
    addRear(arr, 7, &front, &rear);

    printf("\nElements in a deque after addition: ");
    display(arr);

    i = delRear(arr, &front, &rear);
    printf("\nremoved item: %d", i);

    printf("\nElements in a deque after deletion: ");
    display(arr);

    n = count(arr);
    printf("\nTotal number of elements in deque: %d", n);
}

```

Output:

```

E:\Data Structures\Practicals>.\a
Elements in a deque:
front:  8 5 12 5 11 6 0 0 0 0 :rear
removed item: 8
Elements in a deque after deletion:
front:  0 5 12 5 11 6 0 0 0 0 :rear
Elements in a deque after addition:
front:  0 5 12 5 11 6 16 7 0 0 :rear
removed item: 7
Elements in a deque after deletion:
front:  0 5 12 5 11 6 16 0 0 0 :rear
Total number of elements in deque: 6
E:\Data Structures\Practicals>

```

Practical no 14.

Aim: Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Singly Linked List.

Theory:

A linked list data structure includes a series of connected nodes. Here, each node store the data and the address of the next node.

Algorithm:

1. Create a class Node which has two attributes: data and next. Next is a pointer to the next node in the list.
2. Create another class which has two attributes: head and tail.
3. addNode() will add a new node to the list:
 - a. Create a new node.
 - b. It first checks, whether the head is equal to null which means the list is empty.
 - c. If the list is empty, both head and tail will point to the newly added node.
 - d. If the list is not empty, the new node will be added to end of the list such that tail's next will point to a newly added node. This new node will become the new tail of the list.
4. countNodes() will count the nodes present in the list:
 - . Define a node current which will initially point to the head of the list.
 - a. Declare and initialize a variable count to 0.
 - b. Traverse through the list till current point to null.
 - c. Increment the value of count by 1 for each node encountered in the list.
5. display() will display the nodes present in the list:
 - . Define a node current which will initially point to the head of the list.
 - a. Traverse through the list till current points to null.
 - b. Display each node by making current to point to node next to it in each iteration.

Program:


```

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

struct Node
{
    int data;
    struct Node *next;
};

//Traversal Code
void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element = %d\n", ptr->data);
        ptr = ptr->next;
    }
}

//case 1:
struct Node *insertAtBegging(struct Node *head, int data)
{
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));

    ptr->next = head;
    ptr->data = data;
    return ptr;
}

//case 2:
struct Node *insertAtIndex(struct Node *head, int data, int index)
{
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
    int i = 0;
    struct Node *p = head;
    while (i != index - 1)
    {
        p = p->next;
        i++;
    }
    ptr->data = data;
    ptr->next = p->next;
    p->next = ptr;
}

```

```

        return head;
    }
    //case 3:
    struct Node *insertAtEnd(struct Node *head, int data)
    {
        struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
        struct Node *p = head;
        ptr->data = data;

        while (p->next != NULL)
        {
            p = p->next;
        }
        p->next = ptr;
        ptr->next = NULL;
        return head;
    }
    //case 4:
    struct Node *insertAfter(struct Node *head, int data, struct Node *p
    revNode)
    {
        struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
        struct Node *p = prevNode;

        ptr->data = data;
        ptr->next = prevNode->next;
        prevNode->next = ptr;
        return head;
    }
    //case 1:
    struct Node *deleteBegginig(struct Node *head)
    {
        struct Node *ptr = head;

        head = head->next;

        free(ptr);

        return head;
    }

```

```
//case 2:
struct Node *deleteAtIndex(struct Node *head, int index)
{
    struct Node *p = head;
    struct Node *q = p->next;

    for (int i = 0; i < index - 1; i++)
    {
        p = p->next;
        q = q->next;
    }
    p->next = q->next;
    free(q);

    return head;
}
```

```
// case 3:
struct Node *deleteEnd(struct Node *head)
{
    struct Node *p = head;
    struct Node *q = p->next;

    while (q->next != NULL)
    {
        p = p->next;
        q = q->next;
    }
    p->next = NULL;

    free(q);
    return head;
}
```

```
//case 4:Deleting a element with given value
struct Node *deleteValue(struct Node *head, int value)
{
    struct Node *p = head;
    struct Node *q = p->next;

    while (q->data != value && q->next != NULL)
```

```

    {
        p = p->next;
        q = q->next;
    }
    if (q->data == value)
    {
        p->next = q->next;
        free(q);
    }
    return head;
}
int main()
{
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *forth;

    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    forth = (struct Node *)malloc(sizeof(struct Node));

    //Linking HEad
    head->data = 7;
    head->next = NULL;

    printf("Linked List Before Insertion: \n");
    //Before Inserting
    linkedListTraversal(head);

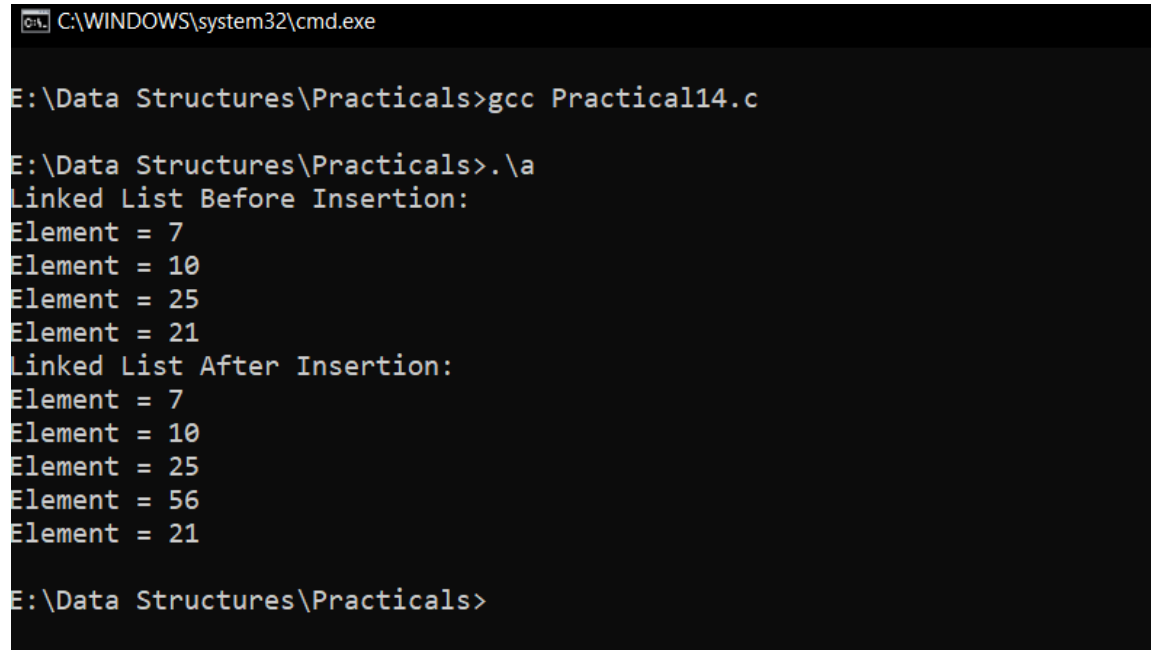
    // head = insertBegging(head, 56);
    // head = insertAtIndex(head, 56,2);
    // head = insertAtEnd(head, 56);
    head = insertAfter(head, 56, third);

    printf("Linked List After Insertion: \n");
    //After inserting at Begginig
    linkedListTraversal(head);

```

```
    return 0;  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
  
E:\Data Structures\Practicals>gcc Practical14.c  
  
E:\Data Structures\Practicals>.\a  
Linked List Before Insertion:  
Element = 7  
Element = 10  
Element = 25  
Element = 21  
Linked List After Insertion:  
Element = 7  
Element = 10  
Element = 25  
Element = 56  
Element = 21  
  
E:\Data Structures\Practicals>
```

Practical no 15.

Aim: Write C program to perform the operations (Insert, Delete, Traverse, and Search) on Circular Singly Linked List. Part – I

Theory:

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.

Algorithm:

Deletion in circular singly linked list at beginning

Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR → NEXT != HEAD

Step 4: SET PTR = PTR → next

[END OF LOOP]

Step 5: SET PTR → NEXT = HEAD → NEXT

Step 6: FREE HEAD

Step 7: SET HEAD = PTR → NEXT

Step 8: EXIT

Program:

//Write C program to perform the operations (Insert, Delete, Traverse, and Search) on

//Circular Singly Linked List.Part – I

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

```
#include <conio.h>
```

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

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node *next;
```

```
};
```

```

void linkedListTraversal(struct Node *head)
{
    struct Node *ptr = head;

    do
    {
        printf("Element is: %d\n", ptr->data);
        ptr = ptr->next;
    } while (ptr != head);
}

struct Node *insertAtBeginning(struct Node *head, int data)
{
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
    struct Node *p = head->next;
    ptr->data = data;

    while (p->next != head)
    {
        p = p->next;
    }
    //Now p is pointing the last list element
    p->next = ptr;
    ptr->next = head;
    head = ptr;
    return head;
}

struct Node *insertAfter(struct Node *head, int data, struct Node *p
revNode)
{
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));

    ptr->data = data;
    ptr->next = prevNode->next;
    prevNode->next = ptr;
    return head;
}

struct Node *insertAtEnd(struct Node *head, int data)
{
    struct Node *ptr = (struct Node *)malloc(sizeof(struct Node));
    struct Node *p = head->next;

```

```

    ptr->data = data;

    while (p->next != head)
    {
        p = p->next;
    }
    p->next = ptr;
    ptr->next = head;

    return head;
}
int main()
{
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *forth;

    //Allocate Memory to the linked list of nodes
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    forth = (struct Node *)malloc(sizeof(struct Node));

    head->data = 7;
    head->next = second;

    second->data = 16;
    second->next = third;

    third->data = 22;
    third->next = forth;

    forth->data = 48;
    forth->next = head;

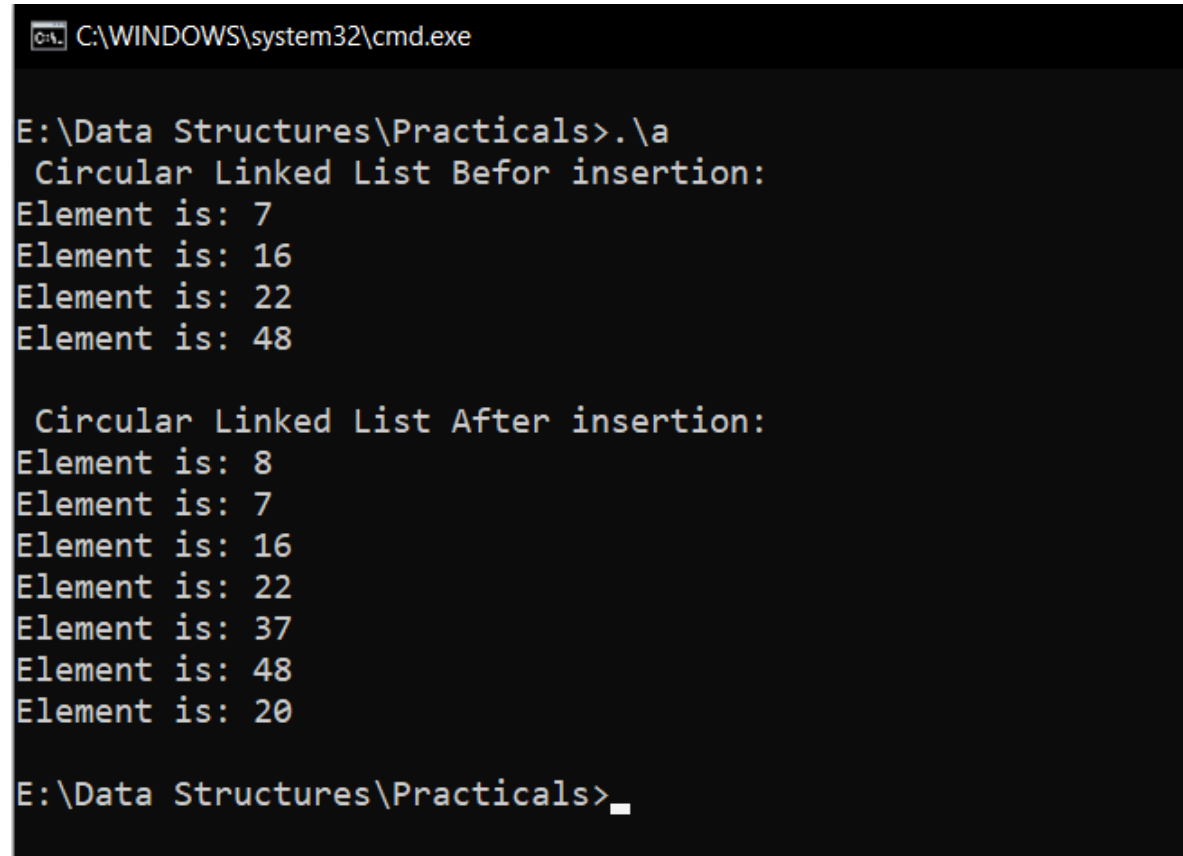
    printf(" Circular Linked List Befor insertion:  \n");
    linkedListTraversal(head);
    head = insertAtBegining(head, 8);
    head = insertAfter(head, 37, third);
    head = insertAtEnd(head, 20);
    printf("\n Circular Linked List After insertion:  \n");

```



```
    linkedListTraversal(head);  
    return 0;  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
  
E:\Data Structures\Practicals>.\a  
Circular Linked List Befor insertion:  
Element is: 7  
Element is: 16  
Element is: 22  
Element is: 48  
  
Circular Linked List After insertion:  
Element is: 8  
Element is: 7  
Element is: 16  
Element is: 22  
Element is: 37  
Element is: 48  
Element is: 20  
  
E:\Data Structures\Practicals>_
```

Practical no 16.

Aim: Write a C Program to design a stack using Linked List.

Theory:

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete ();
void display();
void main()
{
    int choice;
    while (choice != 4)
    {
        printf("\n*****Main Menu*****\n");
        printf("\n=====
=====
\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Displa
y the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                insert();
                break;
            case 2:
```

```

        delete ();
        break;
    case 3:
        display();
        break;
    case 4:
        exit(0);
        break;
    default:
        printf("\nEnter valid choice??\n");
    }
}
}
void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *)malloc(sizeof(struct node));
    if (ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d", &item);
        ptr->data = item;
        if (front == NULL)
        {
            front = ptr;
            rear = ptr;
            front->next = NULL;
            rear->next = NULL;
        }
        else
        {
            rear->next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}

```

```

    }
}
void delete ()
{
    struct node *ptr;
    if (front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front->next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if (front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {
        printf("\nprinting values ..... \n");
        while (ptr != NULL)
        {
            printf("\n%d\n", ptr->data);
            ptr = ptr->next;
        }
    }
}

```

Output:

C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>.\a
Stack Using Linked List

44 33 22 11

Top element is 44

22 11

Top element is 22

E:\Data Structures\Practicals>

Practical no 17.

Aim: Write C program to implement Queue using Linked List

Theory:

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

Program:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete ();
void display();
void main()
{
    int choice;
    while (choice != 4)
    {
        printf("Queue using Linked List\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                insert();
                break;
            case 2:
                delete ();
```

```

        break;
    case 3:
        display();
        break;
    case 4:
        exit(0);
        break;
    default:
        printf("\nEnter valid choice??\n");
    }
}
}
void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *)malloc(sizeof(struct node));
    if (ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d", &item);
        ptr->data = item;
        if (front == NULL)
        {
            front = ptr;
            rear = ptr;
            front->next = NULL;
            rear->next = NULL;
        }
        else
        {
            rear->next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
    }
}

```

```

}
void delete ()
{
    struct node *ptr;
    if (front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front->next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if (front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {
        printf("\nprinting values ..... \n");
        while (ptr != NULL)
        {
            printf("\n%d\n", ptr->data);
            ptr = ptr->next;
        }
    }
}

```

Output:


```
C:\WINDOWS\system32\cmd.exe - .\a

E:\Data Structures\Practicals>gcc Practical17.c

E:\Data Structures\Practicals>.\a
Queue using Linked List

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
2
Queue using Linked List

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2
Queue using Linked List

1.insert an element
2.Delete an element
```

Practical no 18.

Aim: Write C program to Implement BST (Binary Search Tree) and traverse the tree (Inorder, Preorder, Post order).

Theory:

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.

Algorithm:

Program:

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

struct node
{
    int data;

    struct node *leftChild;
    struct node *rightChild;
};

struct node *root = NULL;

void insert(int data)
{
    struct node *tempNode = (struct node *)malloc(sizeof(struct node
));
    struct node *current;
    struct node *parent;

    tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild = NULL;

    //if tree is empty
    if (root == NULL)
    {
```

```

        root = tempNode;
    }
    else
    {
        current = root;
        parent = NULL;

        while (1)
        {
            parent = current;

            //go to left of the tree
            if (data < parent->data)
            {
                current = current->leftChild;

                //insert to the left
                if (current == NULL)
                {
                    parent->leftChild = tempNode;
                    return;
                }
            } //go to right of the tree
            else
            {
                current = current->rightChild;

                //insert to the right
                if (current == NULL)
                {
                    parent->rightChild = tempNode;
                    return;
                }
            }
        }
    }
}

struct node *search(int data)
{
    struct node *current = root;
    printf("Visiting elements: ");

```

```

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

    //go to left tree
    if (current->data > data)
    {
        current = current->leftChild;
    }
    //else go to right tree
    else
    {
        current = current->rightChild;
    }

    //not found
    if (current == NULL)
    {
        return NULL;
    }
}

return current;
}

void pre_order_traversal(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->data);
        pre_order_traversal(root->leftChild);
        pre_order_traversal(root->rightChild);
    }
}

void inorder_traversal(struct node *root)
{
    if (root != NULL)
    {
        inorder_traversal(root->leftChild);

```

```

        printf("%d ", root->data);
        inorder_traversal(root->rightChild);
    }
}

void post_order_traversal(struct node *root)
{
    if (root != NULL)
    {
        post_order_traversal(root->leftChild);
        post_order_traversal(root->rightChild);
        printf("%d ", root->data);
    }
}

int main()
{
    int i;
    int array[7] = {27, 14, 35, 10, 19, 31, 42};

    for (i = 0; i < 7; i++)
        insert(array[i]);

    i = 31;
    struct node *temp = search(i);

    if (temp != NULL)
    {
        printf("[%d] Element found.", temp->data);
        printf("\n");
    }
    else
    {
        printf("[ x ] Element not found (%d).\n", i);
    }

    i = 15;
    temp = search(i);

    if (temp != NULL)
    {
        printf("[%d] Element found.", temp->data);

```

```

        printf("\n");
    }
    else
    {
        printf("[ x ] Element not found (%d).\n", i);
    }

    printf("\nPreorder traversal: ");
    pre_order_traversal(root);

    printf("\nInorder traversal: ");
    inorder_traversal(root);

    printf("\nPost order traversal: ");
    post_order_traversal(root);

    return 0;
}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
E:\Data Structures\Practicals>gcc Practical18.c

E:\Data Structures\Practicals>.\a
Visiting elements: 27 35 [31] Element found.
Visiting elements: 27 14 19 [ x ] Element not found (15).

Preorder traversal: 27 14 10 19 35 31 42
Inorder traversal: 10 14 19 27 31 35 42
Post order traversal: 10 19 14 31 42 35 27
E:\Data Structures\Practicals>_

```

Practical no 19.

Aim: Write C program to calculate height of the given Binary Tree

Theory:

Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1.

Algorithm:

Program:

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

struct node
{
    int data;
    struct node *left;
    struct node *right;
};

int maxDepth(struct node *node)
{
    if (node == NULL)
        return -1;
    else
    {
        /* compute the depth of each subtree */
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        /* use the larger one */
        if (lDepth > rDepth)
            return (lDepth + 1);
        else
            return (rDepth + 1);
    }
}
```

```

}

struct node *newNode(int data)
{
    struct node *node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}

int main()
{
    struct node *root = newNode(1);

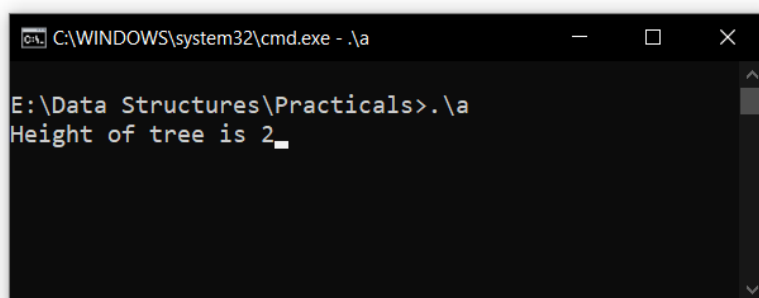
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Height of tree is %d", maxDepth(root));

    getchar();
    return 0;
}

```

Output:



The screenshot shows a Windows command prompt window with the title bar "C:\WINDOWS\system32\cmd.exe - .\a". The command prompt is at the directory "E:\Data Structures\Practicals>". The user has entered the command ".\a", and the output displayed is "Height of tree is 2_".

Practical no 20.

Aim: Write C program to calculate number of nodes in Binary Search Tree

Theory:

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.

Algorithm:

1. If tree is empty then return 0
2. Else
 - (a) Get the size of left subtree recursively i.e., call
size(tree->left-subtree)
 - (a) Get the size of right subtree recursively i.e., call
size(tree->right-subtree)
 - (c) Calculate size of the tree as following:
tree_size = size(left-subtree) + size(right-subtree) + 1
 - (d) Return tree_size

Program:

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

struct node
{
    int data;
    struct node *left;
    struct node *right;
};

struct node *newNode(int data)
{
    struct node *node = (struct node *)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
```

```

        return (node);
    }

    /* Computes the number of nodes in a tree. */
    int size(struct node *node)
    {
        if (node == NULL)
            return 0;
        else
            return (size(node->left) + 1 + size(node->right));
    }

    /* Driver program to test size function*/
    int main()
    {
        struct node *root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
        root->left->left = newNode(4);
        root->left->right = newNode(5);

        printf("Size of the tree is %d", size(root));
        getchar();
        return 0;
    }

```

Output:

```

C:\WINDOWS\system32\cmd.exe - .\a
E:\Data Structures\Practicals>.\a
Height of tree is 2

E:\Data Structures\Practicals>gcc Practical20.c

E:\Data Structures\Practicals>.\a
Size of the tree is 5_

```

Practical no 21.

Aim: Write C program to find out largest nodes in a Binary Search Tree.

Theory:

In this program, we will find out the largest node in the given binary tree. We first define variable **max** that will hold root's data. Then, we traverse through the left sub-tree to find the largest node. Compare it with **max** and store the maximum of two in a variable **max**. Then, we traverse through the right subtree to find the largest node and compare it with **max**. In the end, **max** will have the largest node.

Algorithm:

Step 1 : Start

Step 2 : Define the class **Node** which has three attributes namely: **data**, **left**, and **right**.

Here, left represents the left child of the node and right represents the right child of the node.

Step 3 : Assign the data part of the node with an appropriate value and assign left and right to null.

Step 4 : Define another class which has an attribute root.

Step 5 : **Root** represents the root node of the tree which is initialized to null.

Step 6 : **largestElement()** will find out the largest node in the binary tree:

- a. It checks whether the **root is null**, which means the tree is empty.
- b. If the tree is not empty, define a variable **max** that will store temp's data.
- c. Find out the maximum node in the left subtree by calling **largestElement()** recursively. Store that value in **leftMax**. Compare the value of **max** with **leftMax** and store the maximum of two to **max**.
- d. Find out the maximum node in right subtree by calling **largestElement()** recursively. Store that value in **rightMax**. Compare the value of **max** with **rightMax** and store the maximum of two to **max**.
- e. In the end, **max** will hold the largest node in the binary tree.

Step 7 : Stop

Program:

```
//Write C program to find out largest nodes in a Binary Search Tree
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
```

```

//Represent a node of binary tree
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//Represent the root of binary tree
struct node *root = NULL;

//createNode() will create a new node
struct node *createNode(int data)
{
    //Create a new node
    struct node *newNode = (struct node *)malloc(sizeof(struct
node));
    //Assign data to newNode, set left and right children to NU
LL
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;

    return newNode;
}

//largestElement() will find out the largest node in the binary
tree
int largestElement(struct node *temp)
{
    //Check whether tree is empty
    if (root == NULL)
    {
        printf("Tree is empty\n");
        return 0;
    }
    else

```

```

{
    int leftMax, rightMax;
    //Max will store temp's data
    int max = temp->data;
    //It will find largest element in left subtree
    if (temp->left != NULL)
    {
        leftMax = largestElement(temp->left);
        //Compare max with leftMax and store greater value
into max
        max = (max > leftMax) ? max : leftMax;
    }

    //It will find largest element in right subtree
    if (temp->right != NULL)
    {
        rightMax = largestElement(temp->right);
        //Compare max with rightMax and store greater value
into max
        max = (max > rightMax) ? max : rightMax;
    }
    return max;
}

}

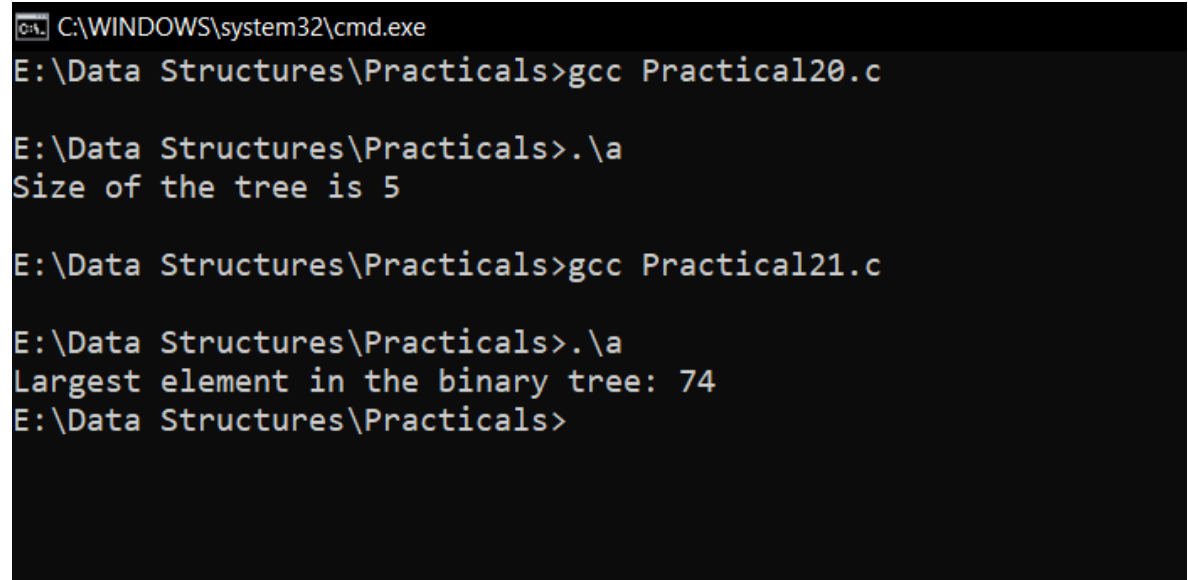
int main()
{
    //Add nodes to the binary tree
    root = createNode(15);
    root->left = createNode(20);
    root->right = createNode(35);
    root->left->left = createNode(74);
    root->right->left = createNode(55);
    root->right->right = createNode(6);

    //Display largest node in the binary tree
    printf("Largest element in the binary tree: %d", largestElement(root));
}

```

```
    return 0;  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
E:\Data Structures\Practicals>gcc Practical20.c  
  
E:\Data Structures\Practicals>.\a  
Size of the tree is 5  
  
E:\Data Structures\Practicals>gcc Practical21.c  
  
E:\Data Structures\Practicals>.\a  
Largest element in the binary tree: 74  
E:\Data Structures\Practicals>
```

Practical no 22.

Aim: Write C program to create a Graph of n vertices using an adjacency list. Also write code to read and print its information and finally to delete a desired node.

Theory:

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.

Program:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
typedef struct node
{
    struct node *next;
    int vertex;
} node;

node *G[20];

int n;
void read_graph();
int in_degree();
int out_degree();
void insert(int vi, int vj);

void main()
{
    int i_degree, o_degree, i;
    read_graph();
    for (i = 0; i < n; i++)
    {
        i_degree = in_degree(i);
        o_degree = out_degree(i);
```

```

        printf("\nNode No=%d \t outdegree=%d", i, i_degree, o_d
egree);
    }
}

```

```

void read_graph()
{
    int i, vi, vj, no_of_edges;
    printf("\nEnter no of vertices :");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        G[i] = NULL;
        printf("\nEnter no of edges : ");
        scanf("%d", &no_of_edges);
        for (i = 0; i < no_of_edges; i++)
        {
            printf("\nEnter an edge(u,v) :");
            scanf("%d%d", &vi, &vj);
            insert(vi, vj);
        }
    }
}

```

```

void insert(int vi, int vj)
{
    node *p, *q;
    q = (node *)malloc(sizeof(node));
    q->vertex = vj;
    q->next = NULL;
    if (G[vi] == NULL)
        G[vi] = q;
    else
    {
        p = G[vi];
        while (p->next != NULL)
            p = p->next;
        p->next = q;
    }
}

```



```

    }
}

int out_degree(int v)
{
    node *p;
    int o_degree = 0;
    p = G[v];
    while (p != NULL)
    {
        o_degree++;
        p = p->next;
    }
    return (o_degree);
}

int in_degree(int v)
{
    node *p;
    int in_degree, i;
    in_degree = 0;
    for (i = 0; i < n; i++)
    {
        p = G[i];
        while (p != NULL)
        {
            if (p->vertex == v)
                in_degree++;
            p = p->next;
        }
    }
    return (in_degree);
}

```

Output:

```
C:\WINDOWS\system32\cmd.exe
E:\Data Structures\Practicals>gcc Practical22.c
E:\Data Structures\Practicals>.\a
Enter no of vertices :6
Enter no of edges : 8
Enter an edge(u,v) :1 2
Enter an edge(u,v) :1 4
Enter an edge(u,v) :3 1
Enter an edge(u,v) :0 2
Enter an edge(u,v) :5 0
Enter an edge(u,v) :0 3
Enter an edge(u,v) :3 4
Enter an edge(u,v) :4 5
Node No=0      outdegree=1
Node No=1      outdegree=1
Node No=2      outdegree=2
Node No=3      outdegree=1
Node No=4      outdegree=2
Node No=5      outdegree=1
E:\Data Structures\Practicals>
```

Practical no 23.

Aim: Write C program to implement the Breadth First Search algorithm

Theory:

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

Algorithm:

Step 1 : Each vertex or node in the graph is known. For instance, you can mark the node as V.

Step 2 : In case the vertex V is not accessed then add the vertex V into the BFS Queue

Step 3 : Start the BFS search, and after completion, Mark vertex V as visited.

Step 4 : The BFS queue is still not empty, hence remove the vertex V of the graph from the queue.

Step 5 : Retrieve all the remaining vertices on the graph that are adjacent to the vertex V

Step 6 : For each adjacent vertex let's say V1, in case it is not visited yet then add V1 to the BFS queue

Step 7 : BFS will visit V1 and mark it as visited and delete it from the queue.

Program:

```
//Write C program to implement the Breadth First Search algorithm
```

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40

struct queue
{
    int items[SIZE];
    int front;
    int rear;
};
```

```

struct queue *createQueue();
void enqueue(struct queue *q, int);
int dequeue(struct queue *q);
void display(struct queue *q);
int isEmpty(struct queue *q);
void printQueue(struct queue *q);

struct node
{
    int vertex;
    struct node *next;
};

struct node *createNode(int);

struct Graph
{
    int numVertices;
    struct node **adjLists;
    int *visited;
};

// BFS algorithm
void bfs(struct Graph *graph, int startVertex)
{
    struct queue *q = createQueue();

    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);

    while (!isEmpty(q))
    {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);

        struct node *temp = graph->adjLists[currentVertex];
    }
}

```

```

        while (temp)
        {
            int adjVertex = temp->vertex;

            if (graph->visited[adjVertex] == 0)
            {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}

// Creating a node
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Creating a graph
struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node *));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
}

```

```

    }

    return graph;
}

// Add edge
void addEdge(struct Graph *graph, int src, int dest)
{
    // Add edge from src to dest
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Create a queue
struct queue *createQueue()
{
    struct queue *q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}

// Check if the queue is empty
int isEmpty(struct queue *q)
{
    if (q->rear == -1)
        return 1;
    else
        return 0;
}

// Adding elements into queue

```

```

void enqueue(struct queue *q, int value)
{
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else
    {
        if (q->front == -1)
            q->front = 0;
        q->rear++;
        q->items[q->rear] = value;
    }
}

```

```

// Removing elements from queue
int dequeue(struct queue *q)
{
    int item;
    if (isEmpty(q))
    {
        printf("Queue is empty");
        item = -1;
    }
    else
    {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear)
        {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
    return item;
}

```

```

// Print the queue
void printQueue(struct queue *q)
{

```

```

    int i = q->front;

    if (isEmpty(q))
    {
        printf("Queue is empty");
    }
    else
    {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++)
        {
            printf("%d ", q->items[i]);
        }
    }
}

int main()
{
    struct Graph *graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);

    bfs(graph, 0);

    return 0;
}

```

Output:

C:\WINDOWS\system32\cmd.exe

E:\Data Structures\Practicals>gcc Practical23.c

E:\Data Structures\Practicals>.\a

Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3

E:\Data Structures\Practicals>_

Practical no 24.

Aim: Write C program to implement the Depth First Search algorithm

Theory:

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

Algorithm:

Step 1 : Push the source node to the stack.

Step 2 : Maintain a data structure to mark if a node is visited or not, say,
boolean[] visited = new boolean[total_nodes_in_graph]

Step 3 : Mark source node S as visited: visited[S] = True

Step 4 : While stack is not empty repeat steps 5 - 8 below

Step 5 : Pop node, say, A from the stack

Step 6 : If visited[A] is True go to step 5, else go to step 7.

Step 7 : Mark visited[A] = True.

Step 8 : Check if A is the node that we need. If yes, break dfs. Else, push all the adjacent nodes of A which are not visited into the stack.

Program:

//Write C program to implement the Breadth First Search algorithm

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

struct node
{
    int vertex;
    struct node *next;
};

struct node *createNode(int v);
```

```

struct Graph
{
    int numVertices;
    int *visited;
    struct node **adjLists;
};

// DFS algo
void DFS(struct Graph *graph, int vertex)
{
    struct node *adjList = graph->adjLists[vertex];
    struct node *temp = adjList;

    graph->visited[vertex] = 1;
    printf("Visited %d \n", vertex);

    while (temp != NULL)
    {
        int connectedVertex = temp->vertex;

        if (graph->visited[connectedVertex] == 0)
        {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

// Create a node
struct node *createNode(int v)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Create graph

```

```

struct Graph *createGraph(int vertices)
{
    struct Graph *graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;

    graph->adjLists = malloc(vertices * sizeof(struct node *));

    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

// Add edge
void addEdge(struct Graph *graph, int src, int dest)
{
    // Add edge from src to dest
    struct node *newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph *graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {

```

```

        struct node *temp = graph->adjLists[v];
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main()
{
    struct Graph *graph = createGraph(4);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);

    printGraph(graph);

    DFS(graph, 2);

    return 0;
}

```

Output:

```
C:\WINDOWS\system32\cmd.exe  
E:\Data Structures\Practicals>gcc Practical24.c
```

```
E:\Data Structures\Practicals>.\a
```

```
Adjacency list of vertex 0  
2 -> 1 ->
```

```
Adjacency list of vertex 1  
2 -> 0 ->
```

```
Adjacency list of vertex 2  
3 -> 1 -> 0 ->
```

```
Adjacency list of vertex 3  
2 ->
```

```
Visited 2
```

```
Visited 3
```

```
Visited 1
```

```
Visited 0
```

```
E:\Data Structures\Practicals>_
```