Department of Information Technology
Delhi Technological University

Data Structure Lab (IT-201)

Session 2020-21(IT-3rd Semester Section-2)

Submitted By:
Name: VARUN KUMAR
Roll No.:- 2K19/IT/140
Group:- G3

Submitted To:-
Mr. Jasraj Meena
Assistant Professor
Dept. of IT, DTU

Index

S. No.	Name of the Program	Date of Experiment	Signature	Grade/Remarks
1.	Write a program to Implement Linear Search in C programming language	19/08/20		
2.	Write a program to Implement Binary Search in C programming language. Assume the list is already sorted.	19/08/20		
3	Write a program to insert an element at the mid position in the One-dimensional array	26/08/20		
4	Write a program to delete a given row in the two-dimensional array.	26/08/20		
5	Write a program Implement a stack data structure and perform its operations.	02/09/20		
6	Write a program Implement two stack using single array.	02/09/20		
7	Write a program to find the minimum element of the stack in constant time with using extra space.	09/09/20		
8	Write a program to find the minimum element of the stack in constant time without using extra space.	09/09/20		
9	Write a program to implement Queue Data structure.	16/09/20		
10	Write a program to reverse first k elements of a given Queue.	16/09/20		
11	Write a program to check weather given string is	23/09/20		

	Palindrome or not using DEQUEUE.			
12	Implement Tower of Hanoi Problem using Stack	23/09/20		
13	Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.	07/10/20		
14	Write a program to split a given linked list into two sub-list as Front sub-list and Back sub-list, if odd number of element then add last element into front list.	07/10/20		
15	Given a Sorted doubly linked list of positive integers and an integer, then finds all the pairs (sum of two nodes data part) that is equal to the given integer value. Example: Double Linked List 2, 5, 7, 8, 9, 10, 12, 16, 19, 25, and P=35 then pairs will be Pairs will be (10, 25), (16, 19).	14/10/20		
16	Write a program to implement the Binary Tree using linked list and perform In-order traversal.	14/10/20		
17	Write a Program to check weather given tree is Binary Search Tree or not.	21/10/20		
18	Write a program to implement insertion in AVL tree.	21/10/20		
19	Write a Algorithm to count the number of leaf nodes in a a AVL tree.	28/10/20		
20	Write a program to Delete a key from the AVL tree.	28/10/20		

21	Write a program to implement Stack Data Structure using Queue	16/11/20		
22	Write a program to implement Queue Data Structure using Stack	16/11/20		
23	Write a program to implement Graph Data Structure and Its traversal BFS and DFS	25/11/20		

Program-1

1. Objective: -

Write a program to Implement Linear Search in C programming language

2. Theory:-

Linear search in C to find whether a number is present in an array. If it's present, then at what location it occurs. It is also known as a sequential search. It is straightforward and works as follows: we compare each element with the element to search until we find it or the list ends. Linear search for multiple occurrences and using a function.

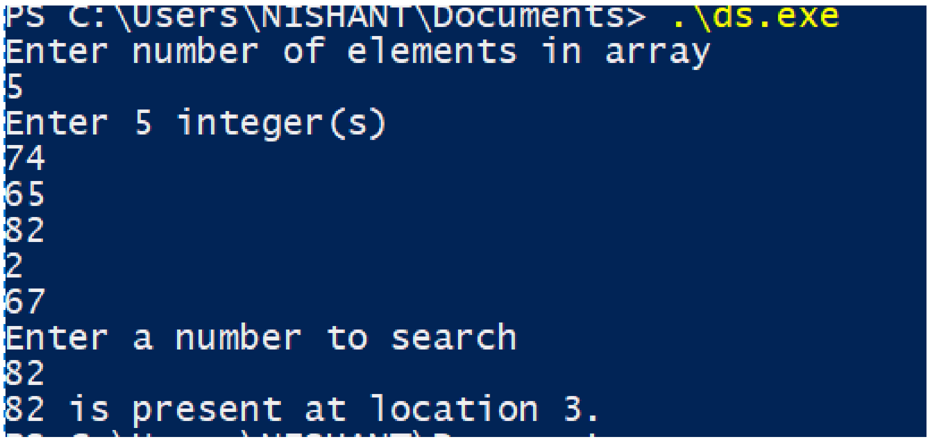
3. Algorithm:-

```
1. Set i to 1
2. if i > n then go to step 7
3. if A[i] = x then go to step 6
4. Set i to i + 1
5. Go to Step 2
6. Print Element x Found at index i and go to step 8
7. Print element not found
8. Exit
```

4. Code: -

```
#include <stdio.h>
int main()
{
  int array[100], search, i, n;

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

  printf("Enter %d integer(s)\n", n);

  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);
  printf("Enter a number to search\n");
  scanf("%d", &search);

for (i = 0; i < n; i++)
  {
    if (array[i] == search)
    {
      printf("%d is present at location %d.\n", search, i+1);
      break;
    }
  }
  if (i == n)
    printf("%d isn't present in the array.\n", search);

  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter number of elements in array
5
Enter 5 integer(s)
74
65
82
2
67
Enter a number to search
82
82 is present at location 3.
```

6. Applications:-

Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an un-ordered list. When many values have to be searched in the same list, it often pays to pre-process the list in order to use a faster method.

Program-2

1. Objective: -

Write a program to Implement Binary Search in C programming language. Assume the list is already sorted.

2. Theory:-

Binary search is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array. Binary search runs in logarithmic time in the worst case, making O(logn) comparisons , where n is no. of elements in the array. It is faster than linear search except for small arrays.

3. Algorithm:-

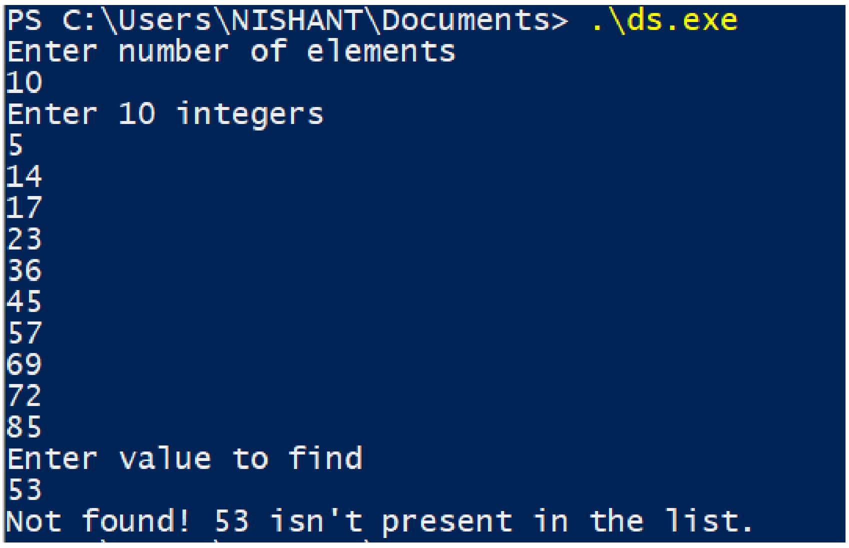
We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

4. Code: -

```
#include <stdio.h>
int main()
{
  int i, first, last, middle, n, search, array[100];

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

  printf("Enter %d integers\n", n);
  for (i = 0; i < n; i++)
    scanf("%d", &array[i]);

  printf("Enter value to find\n");
  scanf("%d", &search);

  first = 0;
  last = n - 1;
  middle = (first+last)/2;

  while (first <= last) {
    if (array[middle] < search)
      first = middle + 1;
    else if (array[middle] == search) {
      printf("%d found at location %d.\n", search, middle+1);
      break;
    }
    else
      last = middle - 1;

    middle = (first + last)/2;
  }
  if (first > last)
    printf("Not found! %d isn't present in the list.\n", search);
  return 0;
}
```

5. Input/Output:-


```
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter number of elements
10
Enter 10 integers
5
14
17
23
36
45
57
69
72
85
Enter value to find
53
Not found! 53 isn't present in the list.
```

6. Applications:-

Binary search can be used to access ordered data quickly when memory space is tight.

Find presence and absence of an element in a sorted array in O(logn) time complexity.

Program-3

1. Objective: -

Write a program to insert an element at the mid position in the One-dimensional array.

2. Theory:-

An array, is a data structure consisting of a collection
of elements (values or variables), each identified by at least one array index or key.

A one-dimensional array (or single dimension array) is a type of linear array.
Accessing its elements involves a single subscript which can either represent a row or column index

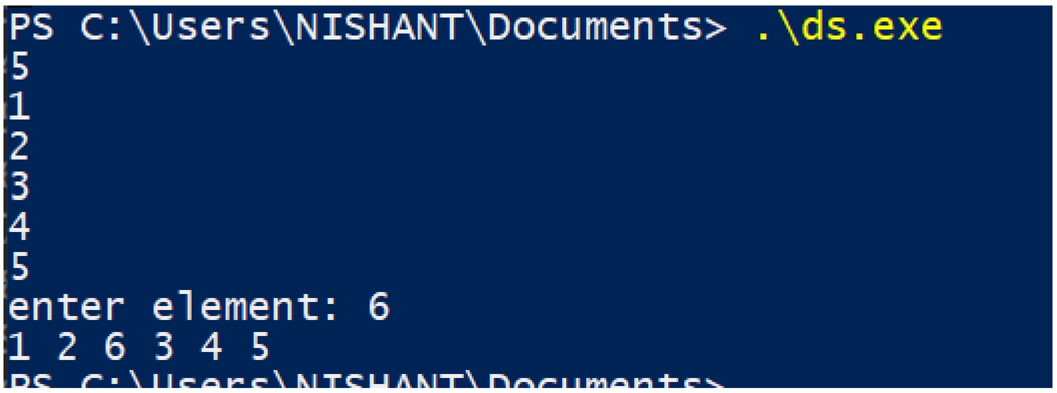
3. Algorithm:-

1. First get the element to be inserted, say x
2. Then get the position at which this element is to be inserted, say pos
3. Then shift the array elements from this position to one position forward, and do this for all the other elements next to pos.
4. Insert the element x now at the position pos, as this is now empty.

4. Code: -

```
#include <stdio.h>

int main(){

int n; scanf("%d",&n);

int a[n+1];

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

printf("enter element: " );

int element; scanf("%d",&element);
if(n%2==0){
int mid=n/2,idx=n;

while(idx!=mid){
a[idx]=a[idx-1];
idx--;
}

a[mid]=element;


}

else{
int mid=n/2,idx=n;

while(idx!=mid){
        a[idx]=a[idx-1];
        idx--;
}
a[mid]=element;

}


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


return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
5
1
2
3
4
5
enter element: 6
1 2 6 3 4 5
PS C:\Users\NISHANT\Documents>
```

6. Applications:-

Single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.

Program-4

1. Objective: -

Write a program to delete a given row in the two-dimensional array.

2. Theory:-

The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure.

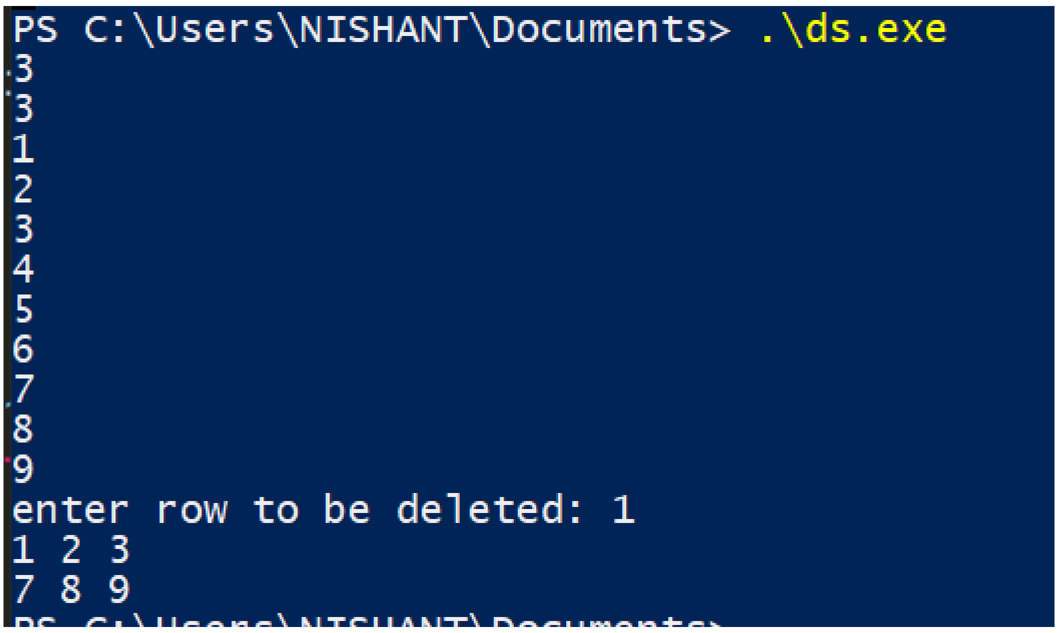
3. Algorithm:-

1. Take the values of an array
2. Ask the user to input the number (index) of the row that has to be deleted.
3. Using for loop iteration, we compare if the row number and the user input number are matching or not.
4. If they are matching and if the row number is less than the size of an array, we are printing the next row. Else, we are printing the row as it is.

4. Code: -

```
#include <stdio.h>


int main(){


int row,col; scanf("%d%d",&row,&col);


int mat[row][col];
for(int i=0;i<row;i++){
        for(int j=0;j<col;j++) scanf("%d",&mat[i][j]);

}

int row_deleted;

printf("enter row to be deleted: ");

scanf("%d",&row_deleted);

for(int i=row_deleted ; i<row-1 ; i++){

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

mat[i][j]=mat[i+1][j];

}

}

for(int i=0;i<row-1;i++){
        for(int j=0;j<col;j++) printf("%d ",mat[i][j]);
                printf("\n");

}
return 0;
}
```

5. **Input/Output:-**



```
PS C:\Users\NISHANT\Documents> .\ds.exe
3
3
1
2
3
4
5
6
7
8
9
enter row to be deleted: 1
1 2 3
7 8 9
```

6. **Applications:-**

Two dimensional arrays are best used and useful for representing information in two dimensional grid. Tic-Tac-Toe game and tracking golf scores are applications of two-dimensional arrays.

Program-5

1. Objective: -

Write a program to Implement a stack data structure and perform its operations.

2. Theory:-

A stack is an abstract data type that serves as a collection of elements, with two main principal

operations - push, which adds an element to the collection, and pop, which removes the most recently added element that was not yet removed.

The order in which elements come off a stack gives rise to its alternative name, LIFO (last in, first out). Additionally, a peek operation may give access to the top without modifying the stack.

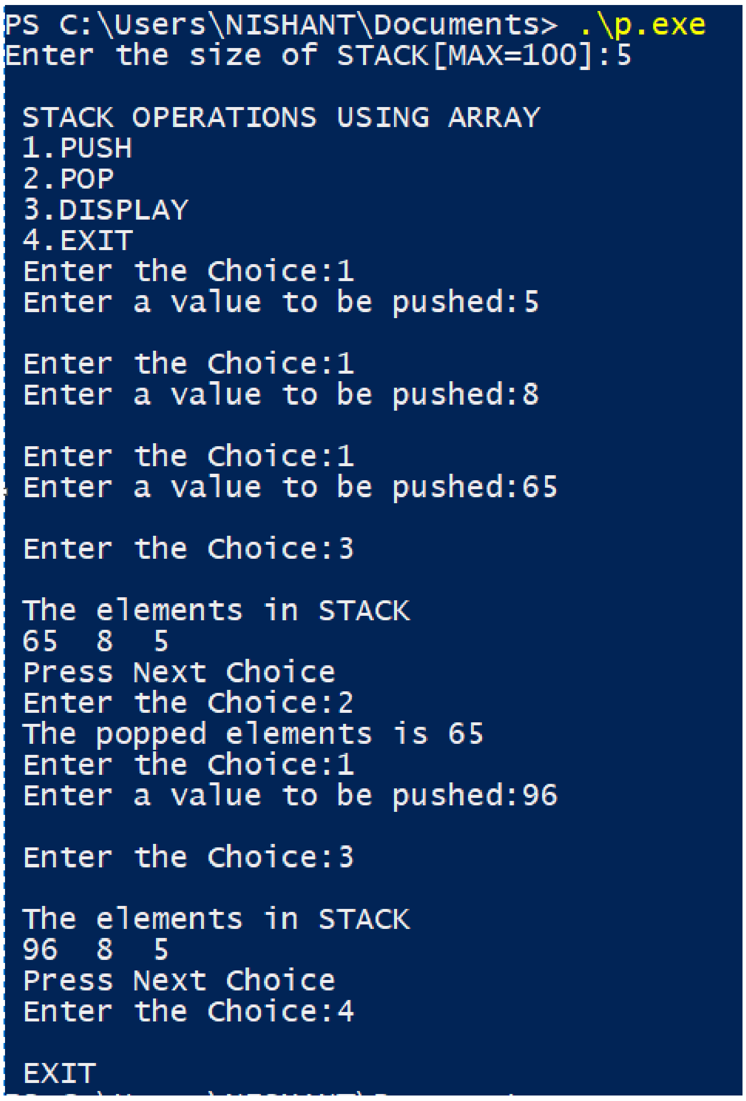
3. Algorithm:-

```
1. Start
2. Declare  Stack[MAX]
   //push:
3. Check if the stack is full or not by comparing top with (MAX-1)
   If the stack is full, Then print "Stack Overflow" i.e, stack is full and cannot be
   pushed with another element
4. Else, the stack is not full
    Increment top by 1 and Set, a[top] = x

    which pushes the element x into the address pointed by top.

    //pop:

5. Check if the stack is empty or not by comparing top with base of array i.e 0
6. If top is less than 0, then stack is empty, print "Stack Underflow"
7. Else, If top is greater than zero the stack is not empty, then store the value
   pointed by top in a variable x=a[top] and decrement top by 1. The popped
   element is x.
8. Stop
```

4. Code: -

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;

void push()
{
   if(top>=n-1)
   {
      printf(" STACK is over flow");

   }
   else
   {
      printf(" Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      stack[top]=x;
   }
}
void pop()
{
   if(top<=-1)
   {
      printf(" Stack is under flow");
   }
   else
   {
      printf(" The popped elements is %d",stack[top]);
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK\n");
      for(i=top; i>=0; i--)
         printf(" %d ",stack[i]);
      printf("\n Press Next Choice");
   }
   else
   {
        printf(" The STACK is empty");
    }

}
int main()
{
    top=-1;
    printf("Enter the size of STACK[MAX=100]:");
    scanf("%d",&n);
    printf("\n STACK OPERATIONS USING ARRAY");
    printf("\n 1.PUSH\n 2.POP\n 3.DISPLAY\n 4.EXIT");
    do
    {
        printf("\n Enter the Choice:");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                display();
                break;
            }
            case 4:
            {
                printf("\n EXIT \n");
                break;
            }
            default:
            {
                printf ("\n Please Enter a Valid Choice(1/2/3/4)");
            }

        }
    }
    while(choice!=4);
    return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe
Enter the size of STACK[MAX=100]:5

 STACK OPERATIONS USING ARRAY
 1.PUSH
 2.POP
 3.DISPLAY
 4.EXIT
 Enter the Choice:1
 Enter a value to be pushed:5

 Enter the Choice:1
 Enter a value to be pushed:8

 Enter the Choice:1
 Enter a value to be pushed:65

 Enter the Choice:3

 The elements in STACK
 65   8   5
 Press Next Choice
 Enter the Choice:2
 The popped elements is 65
 Enter the Choice:1
 Enter a value to be pushed:96

 Enter the Choice:3

 The elements in STACK
 96   8   5
 Press Next Choice
 Enter the Choice:4

 EXIT
```

6. Applications:-

It is used Expression evaluation and syntax parsing, backtracking and compile time memory management as some of its applications

Program-6

1. Objective: -

Write a program to Implement two stack using single array.

2. Theory:-

When a stack is created using single array, we can not able to store large amount of data, thus this problem is rectified using more than one stack in the same array of sufficient array. This technique is called as Multiple Stack.

3. Algorithm:-

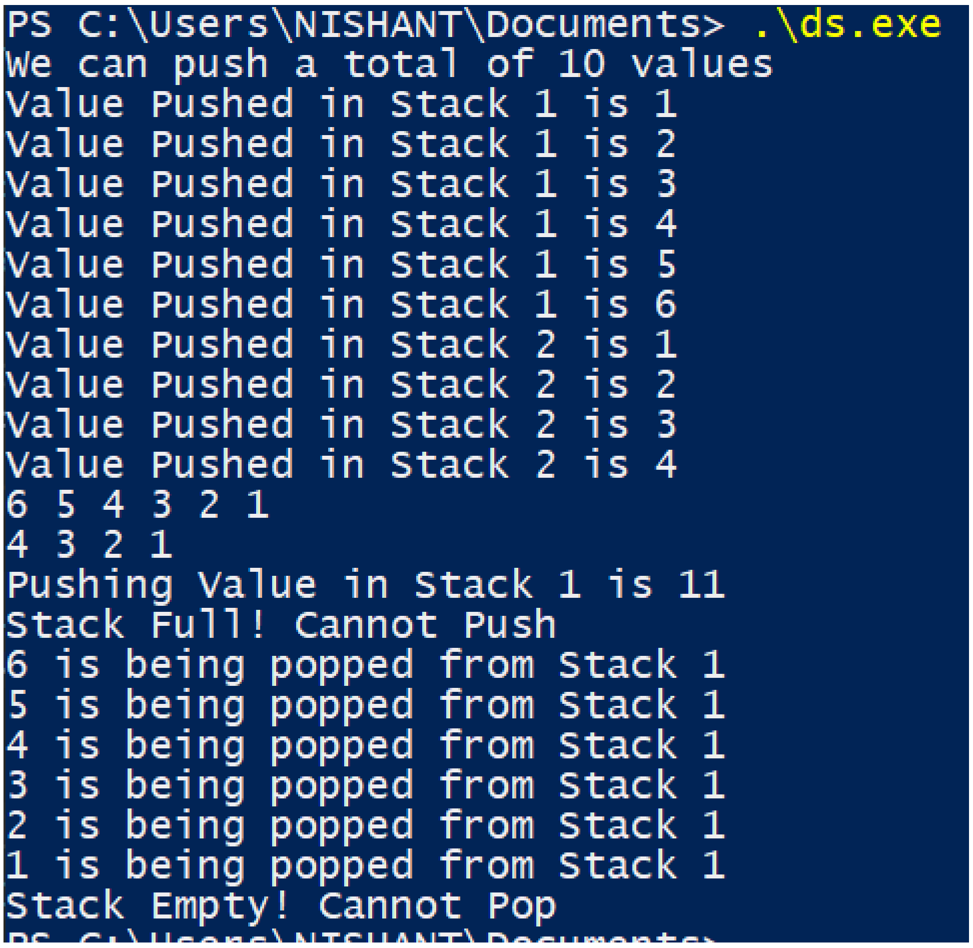
1. Start
2. Stack1 starts from the leftmost element, the first element in stack1 is pushed at index 0.
3. The stack2 starts from the rightmost corner, the first element in stack2 is pushed at index (n-1).
4. If top 1 > top2-1 Stacks overflow for push operation
5. If top 1< 0 for stack 1 and top2 > size for stack2 respective stack is underflow
6. Stop

4. Code: -

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


int ar[SIZE];
int top1 = -1;
int top2 = SIZE;

void push_stack1 (int data)
{
  if (top1 < top2 - 1)
  {
    ar[++top1] = data;
  }
  else
  {
    printf ("Stack Full! Cannot Push\n");
  }
}
void push_stack2 (int data)
{
  if (top1 < top2 - 1)
  {
    ar[--top2] = data;
  }
  else
  {
    printf ("Stack Full! Cannot Push\n");
  }
}

void pop_stack1 ()
{
  if (top1 >= 0)
  {
    int popped_value = ar[top1--];
    printf ("%d is being popped from Stack 1\n", popped_value);
  }
  else
  {
    printf ("Stack Empty! Cannot Pop\n");
  }
}
void pop_stack2 ()
{
  if (top2 < SIZE)
  {
    int popped_value = ar[top2++];
    printf ("%d is being popped from Stack 2\n", popped_value);
  }
  else
  {
    printf ("Stack Empty! Cannot Pop\n");
  }
}

void print_stack1 ()
```

```
{
  int i;
  for (i = top1; i >= 0; --i)
  {
    printf ("%d ", ar[i]);
  }
  printf ("\n");
}
void print_stack2 ()
{
  int i;
  for (i = top2; i < SIZE; ++i)
  {
    printf ("%d ", ar[i]);
  }
  printf ("\n");
}

int main()
{
  int ar[SIZE];
  int i;
  int num_of_ele;

  printf ("We can push a total of 10 values\n");


  for (i = 1; i <= 6; ++i)
  {
    push_stack1 (i);
    printf ("Value Pushed in Stack 1 is %d\n", i);
  }
  for (i = 1; i <= 4; ++i)
  {
    push_stack2 (i);
    printf ("Value Pushed in Stack 2 is %d\n", i);
  }

  print_stack1 ();
  print_stack2 ();

  printf ("Pushing Value in Stack 1 is %d\n", 11);
  push_stack1 (11);
  num_of_ele = top1 + 1;
  while (num_of_ele)
  {
    pop_stack1 ();
    --num_of_ele;
  }

  pop_stack1 ();

  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
We can push a total of 10 values
Value Pushed in Stack 1 is 1
Value Pushed in Stack 1 is 2
Value Pushed in Stack 1 is 3
Value Pushed in Stack 1 is 4
Value Pushed in Stack 1 is 5
Value Pushed in Stack 1 is 6
Value Pushed in Stack 2 is 1
Value Pushed in Stack 2 is 2
Value Pushed in Stack 2 is 3
Value Pushed in Stack 2 is 4
6 5 4 3 2 1
4 3 2 1
Pushing Value in Stack 1 is 11
Stack Full! Cannot Push
6 is being popped from Stack 1
5 is being popped from Stack 1
4 is being popped from Stack 1
3 is being popped from Stack 1
2 is being popped from Stack 1
1 is being popped from Stack 1
Stack Empty! Cannot Pop
```

6. Applications:-

Used in Back/Forward stacks on browsers, Undo/Redo stacks in Excel or Word, Activation records of method calls.

Program-7

1. Objective: -

Write a program to find the minimum element of the stack in constant time with using extra space.

2. Theory:-

We will make an auxiliary stack which stores the minimum element of given stack and operations on this stack will depend on deletion or insertion of minimum element in given stack

3. Algorithm:-

1. Start
2. Declare Stack[MAX]
   //Push
3. push element x to the first stack (the stack with actual elements)
4. compare x with the top element of the second stack (the auxiliary stack). Let the top element be y.
5. If x is smaller than y then push x to the auxiliary stack.
6. If x is greater than y then push y to the auxiliary stack.
   //Pop()
7. pop the top element from the auxiliary stack.
8. pop the top element from the actual stack and return it.
   //getMin()
9. Return the top element of the auxiliary stack.
10. Stop

4. Code: -

```
#include <stdio.h>

int a[100],b[100],top=-1,n;


void push()
```

```
{
 int x;
 if(top==n-1)
 printf("Stack is FULL\n");
 else if(top==-1)
 {
   top++;
   printf("Enter the element to be pushed: ");
   scanf("%d",&x);
   a[top]=b[top]=x;
 }
 else
 {
   top++;
   printf("Enter the element to be pushed: ");
   scanf("%d",&x);
   a[top]=x;
   if(x>=b[top-1])
   b[top]=b[top-1];
   else
   b[top]=x;
 }
}

void pop()
{
 if(top==-1)
 printf("Top is EMPTY\n");
 else
```

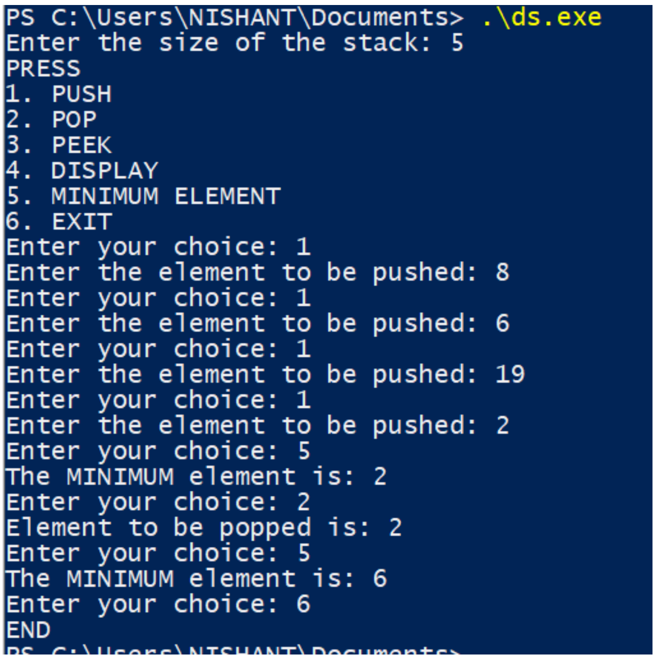
```
  {
    printf("Element to be popped is: %d\n",a[top]);
    top--;
  }
}

void peek()
{
  if(top==-1)
  printf("Top is EMPTY\n");
  else
    printf("Element at the top is: %d\n",a[top]);
}

void display()
{
  if(top==-1)
  printf("Top is EMPTY\n");
  else
  { int i;
    printf("Elements in the stack are\n");
    for(i=top;i!=-1;i--)
    {
      printf("%d ",a[i]);
    }
    printf("\n");
  }
}
```

```
void min()
{
  if(top==-1)
  printf("Top is EMPTY\n");
  else
  {
    printf("The MINIMUM element is: %d\n",b[top]);
  }
}

int main() {
  int choice;
  printf("Enter the size of the stack: ");
  scanf("%d",&n);
  printf("PRESS\n");
  printf("1. PUSH\n2. POP\n3. PEEK\n4. DISPLAY\n5. MINIMUM ELEMENT\n6.
EXIT\n");
  do
  {
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1:
      {
        push();
        break;
      }
      case 2:
```

```
  {
    pop();
    break;
  }
  case 3:
  {
    peek();
    break;
  }
  case 4:
  {
    display();
    break;
  }
  case 5:
  {
    min();
    break;
  }
  case 6:
  {
    printf("END\n");
    break;
  }
  default:
  {
    printf("Enter correct choice\n");
  }
}
```

```
  }
  while(choice!=6);
  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter the size of the stack: 5
PRESS
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. MINIMUM ELEMENT
6. EXIT
Enter your choice: 1
Enter the element to be pushed: 8
Enter your choice: 1
Enter the element to be pushed: 6
Enter your choice: 1
Enter the element to be pushed: 19
Enter your choice: 1
Enter the element to be pushed: 2
Enter your choice: 5
The MINIMUM element is: 2
Enter your choice: 2
Element to be popped is: 2
Enter your choice: 5
The MINIMUM element is: 6
Enter your choice: 6
END
PS C:\Users\NISHANT\Documents>
```

6. Applications:-

We will get minimum element in stack in a constant time

Maximum operations of stack will take constant time.

Program-8

1. Objective: -

Write a program to find the minimum element of the stack in constant time without using extra space.

2. Theory:-

We define a variable minEle that stores the current minimum element in the stack. when minimum element is removed, we push (2x – minEle) into the stack instead of x so that previous minimum element can be retrieved using current minEle and its value stored in stack.

3. Algorithm:-

```
1. Start
2. Declare  Stack[MAX]
   //push
3. If stack is empty, insert x into the stack and make minEle equal to x.
4. If stack is not empty, compare x with minEle. Two cases arise:
     a. If x is greater than or equal to minEle, simply insert x.
     b. If x is less than minEle, insert (2*x – minEle) into the stack and make
        minEle equal to x.
   //pop
5. Remove element from top. Let the removed element be y. Two cases arise
     a. If y is greater than or equal to minEle, the minimum element in the stack
        is still minEle.
     b. If y is less than minEle, the minimum element now becomes (2*minEle
        – y), so update (minEle = 2*minEle – y)
6. Stop
```

4. Code: -

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

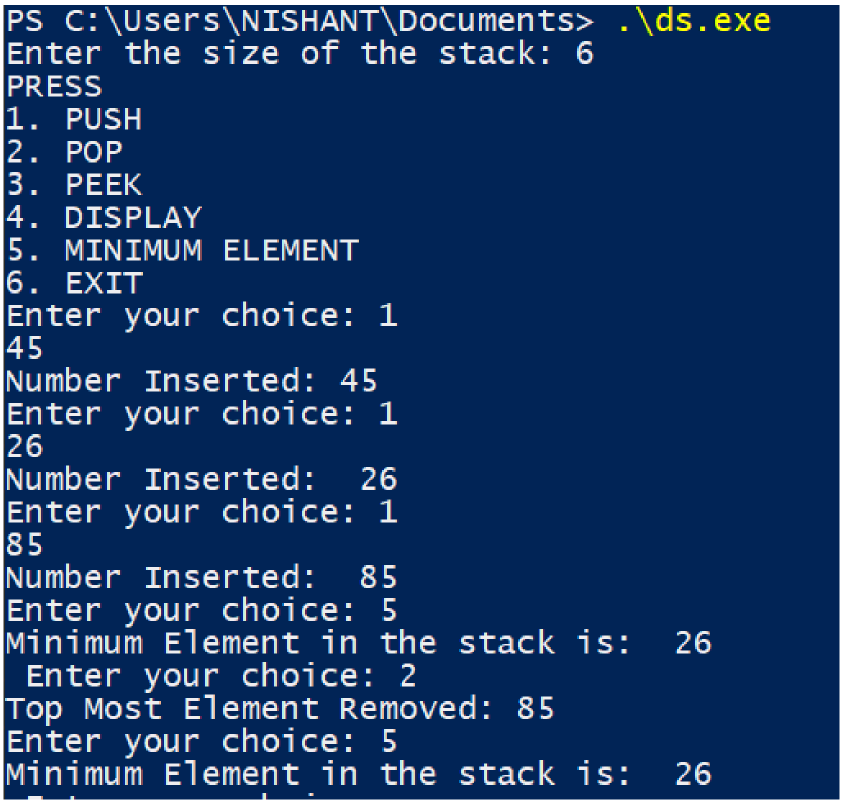
```
int a[100],minEle,top=-1,n;

void push(int x)
{
  // Insert new number into the stack
        if (top==-1)
        {
            minEle = x;
            top++;
            a[top]=x;
            printf  ("Number Inserted: %d\n",x);
            return;
        }

        // If new number is less than minEle
        if (x < minEle)
        {
           top++;
            a[top]=2*x - minEle;
            minEle = x;
        }

        else{
           top++;
           a[top]=x;
        }

        printf(  "Number Inserted:  %d\n",x);
}
```

```
void pop()
{
  if (top==-1)
        {
            printf ("Stackintf is empty\n");
            return;
        }

        printf( "Top Most Element Removed: ");
        int t = a[top];

        top--;

        // Minimum will change as the minimum element
        // of the stack is being removed.
        if (t < minEle)
        {
            printf ("%d\n",minEle);
            minEle = 2*minEle - t;
        }

        else
            printf ("%d\n",t);
}

void peek()
{
  if (top==-1)
```

```
        {
           printf ("Stack is empty ");
           return;
        }

        int t = a[top]; // Top element.

        printf ("Top Most Element is: ");

        // If t < minEle means minEle stores
        // value of t.
        (t < minEle)? printf("%d",minEle ): printf("%d",t);;

    }

void display()
{
  if(top==-1)
  printf("Top is EMPTY\n");
  else
  { int i;
    printf("Elements in the stack are\n");
    for(i=top;i!=-1;i--)
    {
      printf("%d ",a[i]);
    }
    printf("\n");
  }
}
```

```
void min()
{
  if (top==-1)
          printf ("Stack is empty\n");


      // variable minEle stores the minimum element
      // in the stack.
      else
         printf("Minimum Element in the stack is:  %d\n " ,minEle);


}


int main() {
  int choice;
  printf("Enter the size of the stack: ");
  scanf("%d",&n);
  printf("PRESS\n");
  printf("1. PUSH\n2. POP\n3. PEEK\n4. DISPLAY\n5. MINIMUM ELEMENT\n6.
EXIT\n");
  do
  {
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
      case 1:
      {    int x; scanf("%d",&x);
  push(x);
  break;
}
case 2:
{
  pop();
  break;
}
case 3:
{
  peek();
  break;
}
case 4:
{
  display();
  break;
}
case 5:
{
  min();
  break;
}
case 6:
{
  printf("END\n");
  break;
}
default:
      {
        printf("Enter correct choice\n");
      }
    }
  }
  while(choice!=6);
  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter the size of the stack: 6
PRESS
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. MINIMUM ELEMENT
6. EXIT
Enter your choice: 1
45
Number Inserted: 45
Enter your choice: 1
26
Number Inserted:  26
Enter your choice: 1
85
Number Inserted:  85
Enter your choice: 5
Minimum Element in the stack is:  26
 Enter your choice: 2
Top Most Element Removed: 85
Enter your choice: 5
Minimum Element in the stack is:  26
```

6. Applications:-

We will get minimum element in stack in a constant time
Maximum operations of stack will take constant time
Extra space will not be used

Program-9

1. Objective: -

Write a program to implement Queue Data structure.

2. Theory:-

QUEUE is a simple data structure, which has FIFO (First In First Out) property in which Items are removed in the same order as they are entered.
QUEUE has two pointer FRONT and REAR, Item can be pushed by REAR End and can be removed by FRONT End.
Operations on A Queue

Enqueue- adding an element in the queue if there is space in the queue.
Dequeue- Removing elements from a queue if there are any elements in the queue
Front- get the first item from the queue.
Rear- get the last item from the queue.
isEmpty/isFull - checks if the queue is empty or full.

3. Algorithm:-

1. Start
2. Ask the user for the operation like insert, delete, display and exit.
3. According to the option entered, access its respective function using switch statement. Use the variables front and rear to represent the first and last element of the queue.
4. In the function insert(), firstly check if the queue is full. If it is, then print the output as “Queue Overflow”. Otherwise take the number to be inserted as input and store it in the variable add_item. Copy the variable add_item to the array queue_array[] and increment the variable rear by 1.
5. In the function delete(), firstly check if the queue is empty. If it is, then print the output as “Queue Underflow”. Otherwise print the first element of the array queue_array[] and decrement the variable front by 1.
6. In the function display(), using for loop print all the elements of the array starting from front to rear.
7. Stop

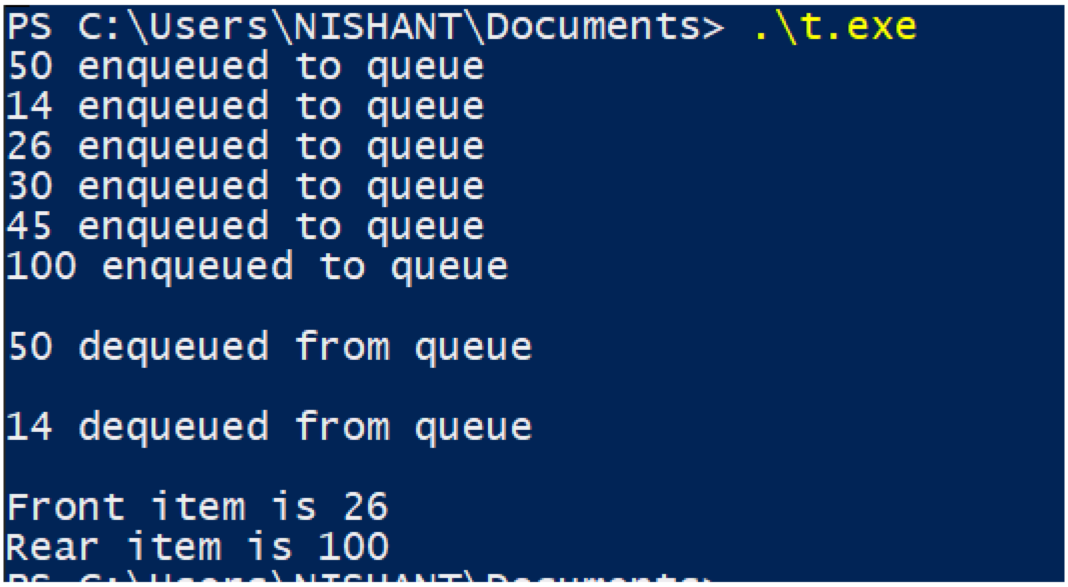
4. Code: -

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

struct Queue {
   int front, rear, size;
   unsigned capacity;
   int* array;
};

struct Queue* createQueue(unsigned capacity)
{
   struct Queue* queue = (struct Queue*)malloc(
      sizeof(struct Queue));
   queue->capacity = capacity;
   queue->front = queue->size = 0;

   queue->rear = capacity - 1;
   queue->array = (int*)malloc(
      queue->capacity * sizeof(int));
   return queue;
}

int isFull(struct Queue* queue)
{
   return (queue->size == queue->capacity);
}

int isEmpty(struct Queue* queue)
{
   return (queue->size == 0);
}

void enqueue(struct Queue* queue, int item)
{
   if (isFull(queue))
      return;
   queue->rear = (queue->rear + 1)
            % queue->capacity;
   queue->array[queue->rear] = item;
   queue->size = queue->size + 1;
```

```
    printf("%d enqueued to queue\n", item);
}

int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)
                % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

int front(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

int rear(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}

int main()
{
    struct Queue* queue = createQueue(100);

    enqueue(queue, 50);
    enqueue(queue, 14);
    enqueue(queue, 26);
    enqueue(queue, 30);
    enqueue(queue, 45);
    enqueue(queue, 100);

    printf("\n%d dequeued from queue\n\n",
        dequeue(queue));
```

```
printf("%d dequeued from queue\n\n",
        dequeue(queue));

    printf("Front item is %d\n", front(queue));
    printf("Rear item is %d\n", rear(queue));

    return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\t.exe
50 enqueued to queue
14 enqueued to queue
26 enqueued to queue
30 enqueued to queue
45 enqueued to queue
100 enqueued to queue

50 dequeued from queue

14 dequeued from queue

Front item is 26
Rear item is 100
```

6. Applications:-

A queue is used in scheduling processes in a CPU.
It is used in data transfer between processes.
It holds multiple jobs which are then called when needed.

Program-10

1. Objective: -

Write a program to reverse first k elements of a given Queue.

2. Theory:-

It is a simple data structure, which has FIFO (First In First Out) property in which Items are removed in the same order as they are entered. Queue has two pointer FRONT and REAR, Item can be pushed by REAR End and can be removed by FRONT End.
Given an integer k and a queue of integers, we need to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.

3. Algorithm:-

1. Create an empty stack.
2. One by one dequeue items from given queue and push the dequeued items to stack.
3. Enqueue the contents of stack at the back of the queue
4. Dequeue (size-k) elements from the front and enque them one by one to the same queue.

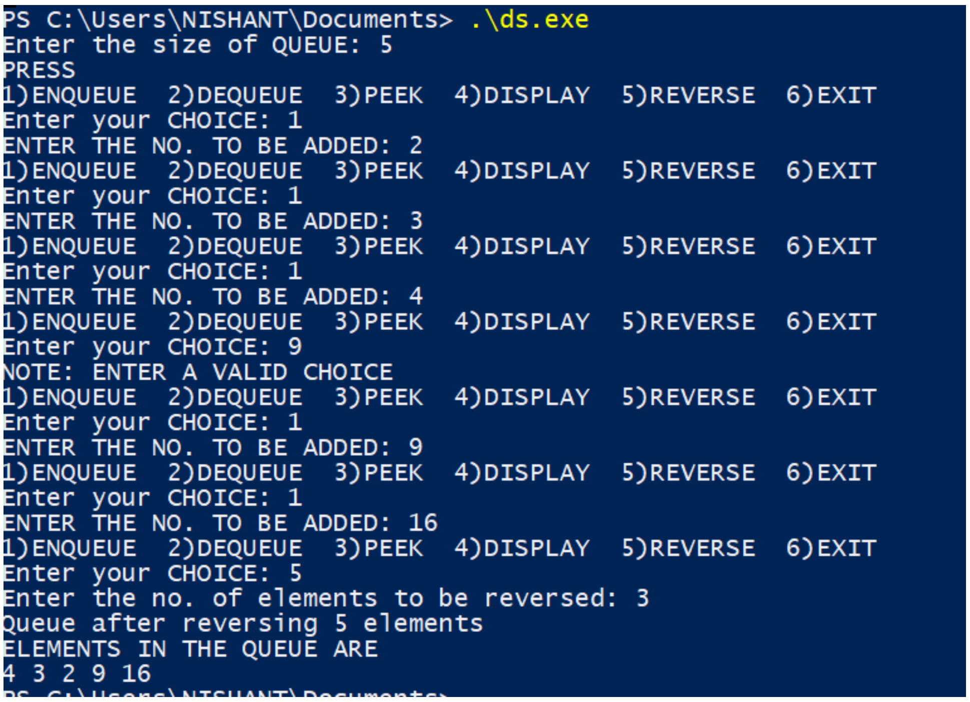
4. Code: -

```
#include <stdio.h>

int rear=-1,front=-1,a[20];

int x;

void enqueue()

{

  if(rear==x-1)

  printf("NOTE: STACK IS FULL\n");

  else if(rear==-1&&front==-1)
  {
    rear=front=0;
    printf("ENTER THE NO. TO BE ADDED: ");
    scanf("%d",&a[rear]);
  }
  else
  {
    rear++;
    printf("ENTER THE NO. TO BE ADDED: ");
    scanf("%d",&a[rear]);
  }
}

void dequeue()
{
  if(rear==-1&&front==-1)
  printf("NOTE: STACK IS EMPTY\n");
  else if(rear==front)
  {
    printf("NO. TO BE REMOVED IS: %d\n",a[front]);
    rear=front=-1;
  }
  else
  {
    printf("NO. TO BE REMOVED IS: %d\n",a[front]);
    front++;
  }
}
```

```
void peek()
{
  if(front==-1)
  printf("NOTE: STACK IS EMPTY\n");
  else
  {
    printf("NO. AT FRONT IS: %d\n",a[front]);
  }
}

void display()
{
  if(front==-1)
  printf("NOTE: STACK IS EMPTY\n");
  else
  {
    int i;
    printf("ELEMENTS IN THE QUEUE ARE\n");
    for(i=front;i<=rear;i++)
    printf("%d ",a[i]);
    printf("\n");
  }
}

void reverse()
{
  if(front==-1)
  printf("NOTE: STACK IS EMPTY\n");
  else
  {
    int t;
    printf("Enter the no. of elements to be reversed: ");
    scanf("%d",&t);
    if(t>rear-front+1)
    printf("ERROR\n");
    else
    {
      int b[t],i,j;
      for(i=front,j=0;j<t;j++)
      {
        b[j]=a[i];
        i++;
      }
      for(i=front,j=t-1;j>=0;j--)
      {
        a[i]=b[j];
        i++;
      }
      printf("Queue after reversing %d elements\n",x);
      display();
    }
  }
}

int main() {
  int n;
  printf("Enter the size of QUEUE: ");
  scanf("%d",&x);
  printf("PRESS\n");
  do
  {
      printf("1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE
6)EXIT\n");
      printf("Enter your CHOICE: ");
      scanf("%d",&n);
      switch(n)
    {
      case 1:
      {
        enqueue();
        break;
      }
      case 2:
      {
        dequeue();
        break;
      }
      case 3:
      {
        peek();
        break;
      }
      case 4:
      {
        display();
        break;
      }
```

```
        case 5:
        {
          reverse();
          break;
        }
        case 6:
        {
          printf("*!Wrong!*\n");
          break;
        }
        default:
        {
          printf("NOTE: ENTER A VALID CHOICE\n");
          break;
        }
      }
    }
    while(n!=5);
    return 0;

}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter the size of QUEUE: 5
PRESS
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 1
ENTER THE NO. TO BE ADDED: 2
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 1
ENTER THE NO. TO BE ADDED: 3
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 1
ENTER THE NO. TO BE ADDED: 4
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 9
NOTE: ENTER A VALID CHOICE
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 1
ENTER THE NO. TO BE ADDED: 9
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 1
ENTER THE NO. TO BE ADDED: 16
1)ENQUEUE  2)DEQUEUE  3)PEEK  4)DISPLAY  5)REVERSE  6)EXIT
Enter your CHOICE: 5
Enter the no. of elements to be reversed: 3
Queue after reversing 5 elements
ELEMENTS IN THE QUEUE ARE
4 3 2 9 16
```

6. Applications:-

To reverse the elements in an queue in various types of problems.

Program-11

1. Objective: -

Write a program to check weather given string is Palindrome or not using DEQUEUE.

2. Theory:-

A palindrome is a string that reads the same forward and backward, for example, radar, toot, and madam. We would construct an algorithm to input a string of characters and check whether it is a palindrome.
A dequeue data structure is the one in which input is allowed from behind but the elements can be deleted from both front as well as the rear positions.

3. Algorithm:-

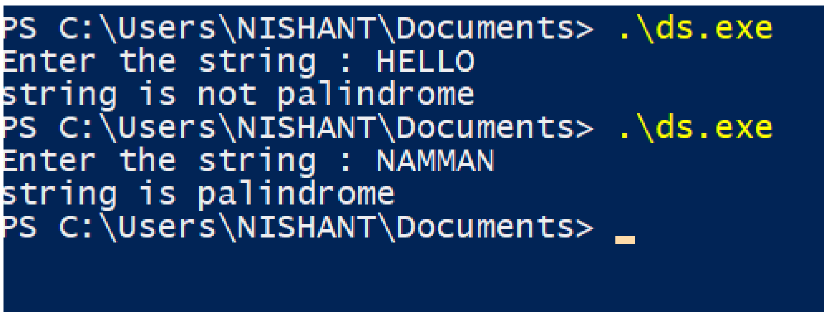
1. Start
2. Take a string as a character array and push to the stack, enqueue to the queue.
3. Pop the element from the stack and dequeue the element from Queue.
4. Compare each character. If the character is not equal, return false.
5. If both characters are equal, return true.
6. Stop

4. Code: -

```
# include<stdio.h>
#include <string.h>
# define MAX 30

int deque_arr[MAX];
int left = -1;
int right = -1;

void insert_right(char added_item)
{

  if((left == 0 && right == MAX-1) || (left == right+1))
    return;
  if (left == -1)
  { left = 0;
    right = 0;}
  else
  if(right == MAX-1)
    right = 0;
  else
    right = right+1;
  deque_arr[right] = added_item ;
}


void insert_left(char added_item){
  if((left == 0 && right == MAX-1) || (left == right+1))
    return;
  if (left == -1)
  { left = 0;
    right = 0;   }
  else
  if(left== 0)
    left=MAX-1;
  else
    left=left-1;
  deque_arr[left] = added_item ;   }

void delete_left()
{ if (left == -1)
    return ;
  if(left == right)
  { left = -1;
    right=-1;  }
  else
    if(left == MAX-1)
      left = 0;
    else
      left = left+1;
}

void delete_right()
{if (left == -1)
    return ;
  if(left == right)
  { left = -1;
    right=-1;  }
  else
    if(right == 0)
      right=MAX-1;
    else
      right=right-1;  }

int front(){
return deque_arr[left];

}

int back(){
  return deque_arr[right];
}


bool empty(){

if(left==-1 and right==-1) return 1;
else return 0;
}

int main()
{
char s[20];

   printf("Enter the string : ");

gets(s);

for(int i=0 ;i<strlen(s);i++){
insert_right(s[i]);
}

bool palindrome=1;

while(!empty()){
char f=front();
char b=back();

if(f!=b) palindrome=0;

delete_left();
delete_right();

}


if(palindrome) printf("string is palindrome\n");
```

```
else printf("string is not palindrome\n");



  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter the string : HELLO
string is not palindrome
PS C:\Users\NISHANT\Documents> .\ds.exe
Enter the string : NAMMAN
string is palindrome
PS C:\Users\NISHANT\Documents> _
```

6. Applications:-

To Check whether the string is a palindrome or not and various other problems.

Program-12

1. Objective: -

Write a program to implement Tower of Hanoi Problem using Stack

2. Theory:-

Tower of Hanoi is a mathematical puzzle. It consists of three poles and a number of disks of different sizes which can slide onto any poles. The puzzle starts with the disk in a neat stack in ascending order of size in one pole, the smallest at the top thus making a conical shape. The objective of the puzzle is to move all the disks from one pole (say ‘source pole’) to another pole (say ‘destination pole’) with the help of the third pole (say auxiliary pole).
The puzzle has the following two rules:

1. You can’t place a larger disk onto smaller disk
2. Only one disk can be moved at a time

3. Algorithm:-

```
1. Start
2. Calculate the total number of moves required i.e. "$2^n$ - 1" here n is the number
   of disks.
3. If number of disks (i.e. n) is even then interchange destination pole and
   auxiliary pole.
4. for i = 1 to total number of moves:
5. if i%3 == 1:
   legal movement of top disk between source pole and destination pole
6. if i%3 == 2:
   legal movement top disk between source pole and auxiliary pole
7. if i%3 == 0:
   legal movement top disk between auxiliary pole and destination pole
8. Stop
```

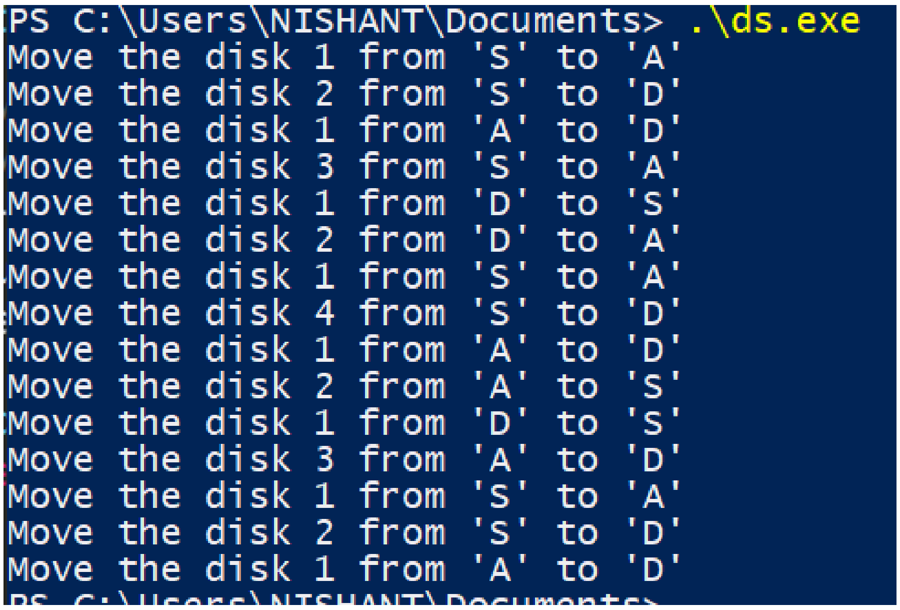
4. Code: -

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

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

struct Stack
{
unsigned capacity;
int top;
int *array;
};

struct Stack* createStack(unsigned capacity)
{
  struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
  stack -> capacity = capacity;
  stack -> top = -1;
  stack -> array =
    (int*) malloc(stack -> capacity * sizeof(int));
  return stack;
}

int isFull(struct Stack* stack)
{
return (stack->top == stack->capacity - 1);
}

int isEmpty(struct Stack* stack)
{
return (stack->top == -1);
}

void push(struct Stack *stack, int item)
{
  if (isFull(stack))
    return;
  stack -> array[++stack -> top] = item;
}

int pop(struct Stack* stack)
{
  if (isEmpty(stack))
    return INT_MIN;
  return stack -> array[stack -> top--];
```

```
}

void moveDisk(char fromPeg, char toPeg, int disk)
{
  printf("Move the disk %d from \'%c\' to \'%c\'\n",
    disk, fromPeg, toPeg);
}

void moveDisksBetweenTwoPoles(struct Stack *src,
      struct Stack *dest, char s, char d)
{
  int pole1TopDisk = pop(src);
  int pole2TopDisk = pop(dest);

  if (pole1TopDisk == INT_MIN)
  {
    push(src, pole2TopDisk);
    moveDisk(d, s, pole2TopDisk);
  }

  else if (pole2TopDisk == INT_MIN)
  {
    push(dest, pole1TopDisk);
    moveDisk(s, d, pole1TopDisk);
  }

  else if (pole1TopDisk > pole2TopDisk)
  {
    push(src, pole1TopDisk);
    push(src, pole2TopDisk);
    moveDisk(d, s, pole2TopDisk);
  }

  else
  {
    push(dest, pole2TopDisk);
    push(dest, pole1TopDisk);
    moveDisk(s, d, pole1TopDisk);
  }
}

void tohIterative(int num_of_disks, struct Stack
      *src, struct Stack *aux,
      struct Stack *dest)
{
  int i, total_num_of_moves;
  char s = 'S', d = 'D', a = 'A';

  if (num_of_disks % 2 == 0)
  {
    char temp = d;
    d = a;
    a = temp;
  }
  total_num_of_moves = pow(2, num_of_disks) - 1;

  for (i = num_of_disks; i >= 1; i--)
    push(src, i);

  for (i = 1; i <= total_num_of_moves; i++)
  {
    if (i % 3 == 1)
    moveDisksBetweenTwoPoles(src, dest, s, d);

    else if (i % 3 == 2)
    moveDisksBetweenTwoPoles(src, aux, s, a);

    else if (i % 3 == 0)
    moveDisksBetweenTwoPoles(aux, dest, a, d);
  }
}

int main()
{
  unsigned num_of_disks = 4;

  struct Stack *src, *dest, *aux;

  src = createStack(num_of_disks);
  aux = createStack(num_of_disks);
  dest = createStack(num_of_disks);

  tohIterative(num_of_disks, src, aux, dest);
  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\ds.exe
Move the disk 1 from 'S' to 'A'
Move the disk 2 from 'S' to 'D'
Move the disk 1 from 'A' to 'D'
Move the disk 3 from 'S' to 'A'
Move the disk 1 from 'D' to 'S'
Move the disk 2 from 'D' to 'A'
Move the disk 1 from 'S' to 'A'
Move the disk 4 from 'S' to 'D'
Move the disk 1 from 'A' to 'D'
Move the disk 2 from 'A' to 'S'
Move the disk 1 from 'D' to 'S'
Move the disk 3 from 'A' to 'D'
Move the disk 1 from 'S' to 'A'
Move the disk 2 from 'S' to 'D'
Move the disk 1 from 'A' to 'D'
PS C:\Users\NISHANT\Documents>
```

6. Applications:-

To solve the famous Tower of Hanoi and similar puzzles for any number of discs iteratively.

Program-13

1. Objective: -

Write a program to implement the Linked List Data structure and insert a new node at the beginning, and at a given position.

2. Theory:-

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. A linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

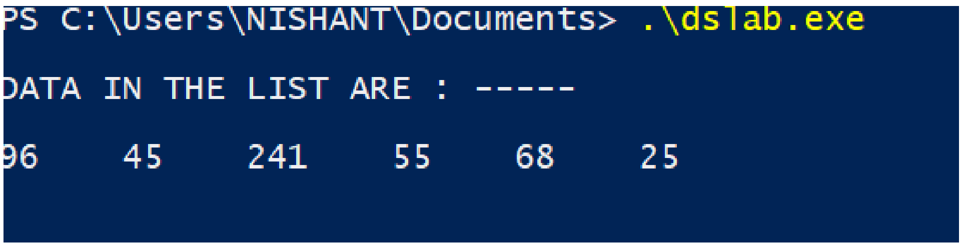
3. Algorithm:-

1. Traverse the Linked list upto position-1 nodes.
2. Once all the position-1 nodes are traversed, allocate memory and the given data to the new node.
3. Point the next pointer of the new node to the next of current node.
4. Point the next pointer of current node to the new node.

4. Code: -

```
#include <stdio.h>

#include<stdlib.h>


struct node{

 int data;

 struct node* next;

};


struct node*start=NULL;


void Insert_at_beginning(int d)

{
 struct node*ptr= (struct node*)malloc(1);
 ptr->data=d;
 ptr->next=start;
 start=ptr;
}

void print_list()
{
 struct node*ptr=start;
 if(start==NULL)
 printf("LIST IS EMPTY\n");
 else{

printf("\nDATA IN THE LIST ARE : -----\n\n");

 while(ptr!=NULL)
 {
 printf("%d    ",ptr->data);
 ptr=ptr->next;
 }
 printf("\n\n\n");
 }


}
void Insert_at_index(int index,int d)
{
 struct node*ptr=start;

 for(int i=0;ptr!=NULL&&i<index-2;i++)
 {
 ptr=ptr->next;
 }

 if(ptr==NULL)
 printf("NOT ENOUGH ELEMENT\n");
 else
 {
 if(index==1)
 Insert_at_beginning(d);
 else{
 struct node*newele = (struct node*)malloc(1);
 newele->data=d;
 struct node*temp= ptr->next;
 ptr->next=newele;
 newele->next=temp;
 }
 }
}
int main()
{



 Insert_at_beginning(25);


 Insert_at_beginning(68);


 Insert_at_beginning(241);
 Insert_at_index(2,55);

 Insert_at_index(1,45);

 Insert_at_beginning(96);

print_list();
 return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\dslab.exe

DATA IN THE LIST ARE : -----

96    45    241    55    68    25
```

6. Applications:-

1. Used in implementation of stacks and queues
2. Used in implementation of graphs: Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Used in dynamic memory allocation: We use linked list of free blocks.

Program-14

1. Objective: -

Write a program to split a given linked list into two sub-list as Front sub-list and Back sub-list, if odd number of element then add last element into front list.

2. Theory:-

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. A linked list consists of nodes where each node contains a data field and a reference (link) to the next node in the list.

3. Algorithm:-

1. Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.
2. Make the second half circular.
3. Make the first half circular.
4. Set head (or start) pointers of the two linked lists.

4. Code: -

```
#include <stdio.h>

#include<stdlib.h>


struct node{

int data;

struct node* next;

};



struct node* Insert_at_beginning(struct node*start,int d)

{
```

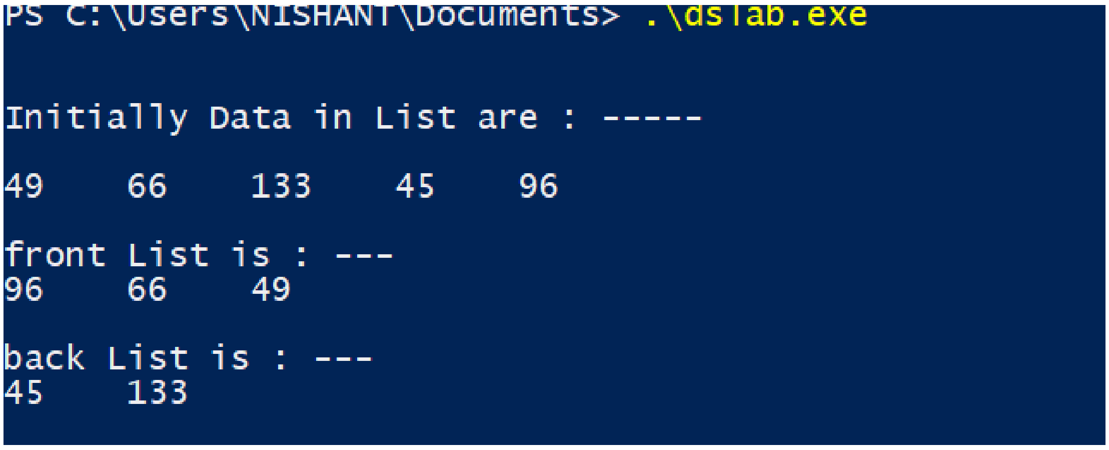
```
struct node*ptr= (struct node*)malloc(1);
ptr->data=d;
ptr->next=start;
start=ptr;
return start;
}


void print_list(struct node* start)
{
struct node*ptr=start;
if(start==NULL)
printf("LIST IS EMPTY\n");
else{
while(ptr!=NULL)
{
printf("%d    ",ptr->data);
ptr=ptr->next;
}
printf("\n\n");
}
}


struct node* Insert_at_index(struct node*start,int index,int d)
{
struct node*ptr=start;
for(int i=0;ptr!=NULL&&i<index-2;i++)
{
```

```
ptr=ptr->next;
}
if(ptr==NULL)
printf("NOT ENOUGH ELEMENT\n");
else
{
if(index==1)
Insert_at_beginning(start,d);
else{
struct node*newele = (struct node*)malloc(1);
newele->data=d;
struct node*temp= ptr->next;
ptr->next=newele;
newele->next=temp;
}
}
return start;
}

int list_size(struct node*start)
{
struct node*ptr=start;
int count=0;
while(ptr!=NULL)
{
count++;
ptr=ptr->next;
}
return count;
```

```
}

void split_list(struct node*start,struct node*&front,struct node*&back)
{
int n=list_size(start);
// printf("%d\n",n);
struct node*ptr=start;
for(int i=1;i<=n/2;i++)
{
front=Insert_at_beginning(front,ptr->data);
ptr=ptr->next;
}
for(int i=1;i<=n/2;i++)
{
back=Insert_at_beginning(back,ptr->data);
ptr=ptr->next;
}
if(n%2==1)
front=Insert_at_beginning(front,ptr->data);


}



int main()
{
struct node*start=NULL;

start=Insert_at_beginning(start,96);
```

```
start=Insert_at_beginning(start,45);
start=Insert_at_beginning(start,133);
start=Insert_at_beginning(start,49);
start=Insert_at_index(start,2,66);

printf("\n\nInitially Data in List are : -----\n\n");

print_list(start);

struct node*front=NULL;

struct node* back=NULL;

split_list(start,front,back);

printf("front List is : ---\n");

print_list(front);

printf("back List is : ---\n");

print_list(back);
return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\dslab.exe

Initially Data in List are : -----

49    66    133    45    96

front List is : ---
96    66    49

back List is : ---
45    133
```

6. Applications:-

1. Used in Maintaining directory of names
2. Used in performing arithmetic operations on long integers
3. Used in manipulation of polynomials by storing constants in the node of linked list
4. Used in representing sparse matrices

Program-15

1. Objective: -

Given a Sorted doubly linked list of positive integers and an integer, then finds all the pairs (sum of two nodes data part) that is equal to the given integer value. Example: Double Linked List 2, 5, 7, 8, 9, 10, 12, 16, 19, 25, and P=35 then pairs will be Pairs will be (10, 25), (16, 19).

2. Theory:-

A Doubly linked list is a linked list that consists of a set of sequentially linked records called nodes. Each node contains two fields that are references to the previous and to the next node in the sequence of node.
A simple approach for this problem is to one by one pick each node and find the second element whose sum is equal to x in the remaining list by traversing in forward direction. Time complexity for this problem will be O(n^2), n is the total number of nodes in the doubly linked list.

3. Algorithm:-

1. Start
2. Initialize two pointer variables to find the candidate elements in the sorted doubly linked list. Initialize first with start of doubly linked list i.e.; first=head and initialize second with last node of doubly linked list i.e.; second=last node.
3. We initialize first and second pointers as first and last nodes. Here we don't have random access, so to find a second pointer, we traverse the list to initialize the second.
4. If the current sum of first and second is less than x, then we move first in forward direction. If the current sum of first and second element is greater than x, then we move second in backward direction.
5. The loop terminates when either of two pointers become NULL, or they cross each other (second->next = first), or they become same (first == second)
6. Stop

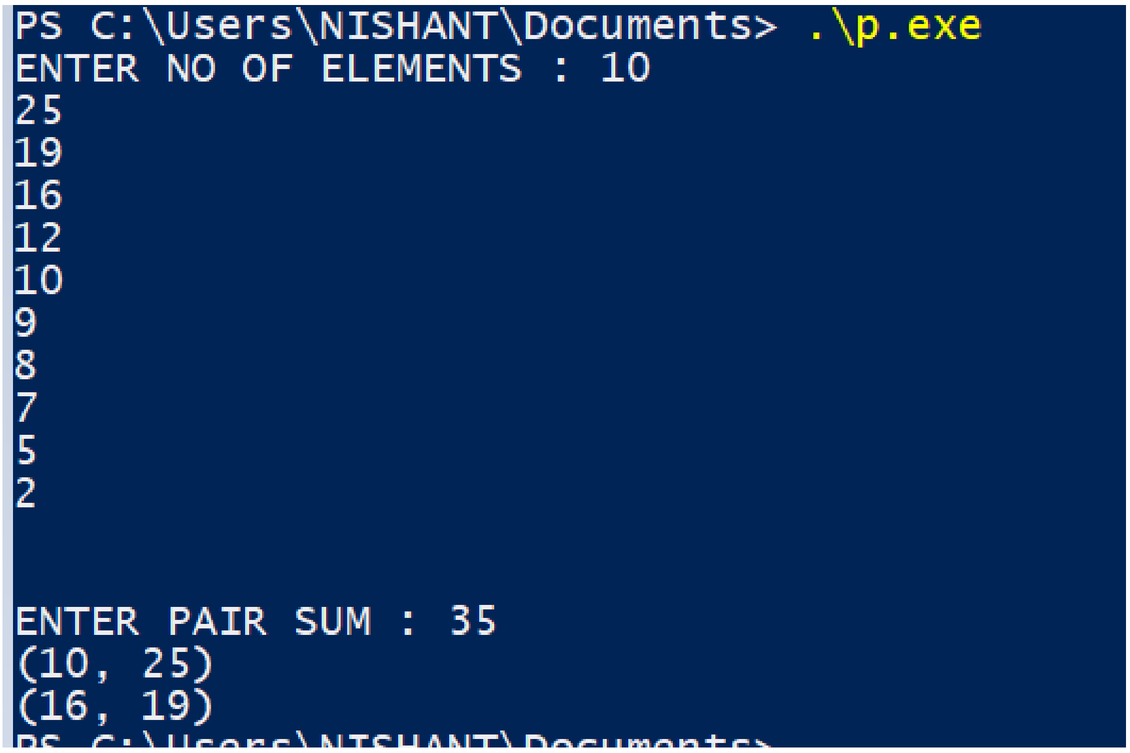
4. Code: -

```
#include<bits/stdc++.h>
using namespace std;
```

```
class ListNode
{    public:
         int val;
         class ListNode *next, *prev;
};

void Pair_Sum(class ListNode *head, int p)
{
         class ListNode *first = head;
         class ListNode *second = head;
         while (second->next != NULL)
                  second = second->next;

         bool found = false;

         while (first != NULL && second != NULL &&
                  first != second && second->next != first)
         {
                  if ((first->val + second->val) == p)
                  {
                           found = true;
                           cout << "(" << first->val<< ", "
                                    << second->val << ")" << endl;

                           first = first->next;

                           second = second->prev;
                  }
                  else
                  {
                           if ((first->val + second->val) < p)
                                    first = first->next;
                           else
                                    second = second->prev;
                  }
         }

         if (found == false)
                  cout << "No pair found";
}

void insert_in_list(class ListNode **head, int val)
{
         class ListNode *temp = new ListNode;
         temp->val = val;
```

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

int main()
{
        class ListNode *head = NULL;

  int n;
cout<<"ENTER NO OF ELEMENTS : "; cin>>n;

for(int i=0;i<n;i++){
int a; cin>>a;
insert_in_list(&head,a);

}
int p;
cout<<"\n\nENTER PAIR SUM : "; cin>>p;


Pair_Sum(head,p );

        return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe
ENTER NO OF ELEMENTS : 10
25
19
16
12
10
9
8
7
5
2


ENTER PAIR SUM : 35
(10, 25)
(16, 19)
```

6. Applications:-

1. Doubly linked lists can be used in navigation systems where both front and back navigation is required.
2. It is used by browsers to implement backward and forward navigation of visited web pages i.e. back and forward buttons.
3. It is also used by various applications to implement Undo and Redo functionality.
4. It can also be used to represent a deck of cards in games.

Program-16

1. Objective: -

Write a program to implement the Binary Tree using linked list and perform In-order traversal.

2. Theory:-

A tree whose elements have at most 2 children is called a binary tree. since each element in a binary tree can have only 2 children, we typically name them the left and right child. In In-order traversal method, the left subtree is visited first, then the root and later the right sub-tree..If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

3. Algorithm:-

```
1. Define Node class
2. node is created, left and right set to null.
3. root node of the tree initialized to null.
4. insert()  adds new node to tree:
    a) It checks if root is null, add new node as root.
    b) Else, it will add root to the queue.
    c) The variable node is current node.
    d) checks if node has left and right child. If yes, adds both nodes to queue.
        i. If left not present, adds left child.
        ii. If left present, adds right child.
5. InorderTraversal()  display nodes in inorder fashion.
```

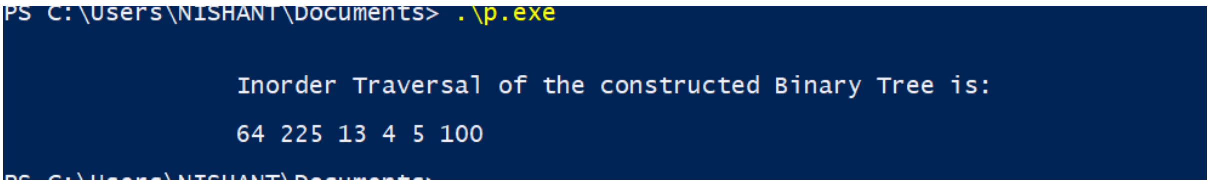
4. Code: -

```
#include <iostream>
#include <string>
#include <queue>
using namespace std;

class ListNode
{ public:
        int data;
```

```
    ListNode* next;
};

class TreeNode
{ public:
    int data;
    TreeNode *left, *right;
};

void insert(class ListNode** head_ref, int new_data)
{
    class ListNode* new_node = new ListNode;
    new_node->data = new_data;

    new_node->next = (*head_ref);

    (*head_ref) = new_node;
}

TreeNode* New_Tree(int data)
{
    TreeNode *temp = new TreeNode;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

void Convert_List_to_binary(ListNode *head, TreeNode* &root)
{
    queue<TreeNode *> q;

    if (head == NULL)
    {
        root = NULL;
        return;
    }

    root = New_Tree(head->data);
    q.push(root);

    head = head->next;

    while (head)
    {
        TreeNode* parent = q.front();
        q.pop();
```

```
                TreeNode *leftChild = NULL, *rightChild = NULL;
                leftChild = New_Tree(head->data);
                q.push(leftChild);
                head = head->next;
                if (head)
                {
                        rightChild = New_Tree(head->data);
                        q.push(rightChild);
                        head = head->next;
                }

                parent->left = leftChild;
                parent->right = rightChild;
        }
}

void inorder_Traversal(TreeNode* root)
{
        if (root)
        {
                inorder_Traversal( root->left );
                cout << root->data << " ";
                inorder_Traversal( root->right );
        }
}

int main()
{
        class ListNode* head = NULL;
        insert(&head, 5);
        insert(&head, 13);
        insert(&head, 64);
        insert(&head, 100);
        insert(&head, 225);
        insert(&head,  4);

        TreeNode *root;
        Convert_List_to_binary(head, root);

        cout << "\n\n\t\tInorder Traversal of the constructed Binary Tree is: \n\n\t\t";
        inorder_Traversal(root);
        cout<<"\n\n";
        return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe

                Inorder Traversal of the constructed Binary Tree is:

                64 225 13 4 5 100

PS C:\Users\NISHANT\Documents>
```

6. Applications:-

Used in many search applications where data is constantly entering/leaving, eg:map and set

Program-17

1. Objective: -

Write a Program to check weather given tree is Binary Search Tree or not.

2. Theory:-

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

3. Algorithm:-

To see if a binary tree is a binary search tree, check:

1) If a node is a left child, then its key and the keys of the nodes in its right subtree are less than its parent’s key.

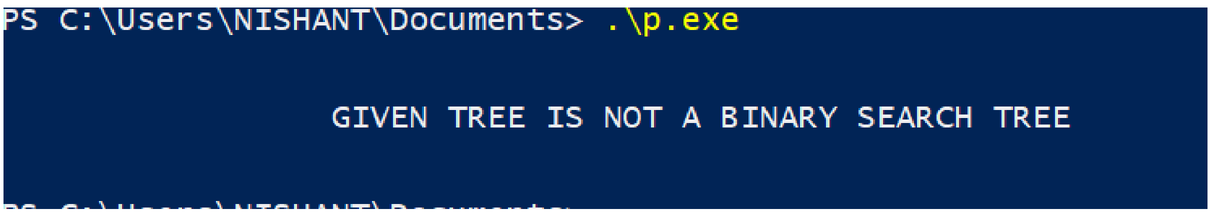
If a node is a right child, then its key and the keys of the nodes in its left subtree are greater than its parent’s key

4. Code: -

```
#include<bits/stdc++.h>


using namespace std;


class TreeNode
{
        public:
        int val;
        TreeNode* left;
        TreeNode* right;
        TreeNode(int val)
        {
                this->val = val;
                this->left = NULL;
                this->right = NULL;
        }
};

bool Check_BST(TreeNode* TreeNode, int min, int max)
{
        if (TreeNode==NULL)
                return 1;

        if (TreeNode->val < min || TreeNode->val > max)
                return 0;

        return
                Check_BST(TreeNode->left, min, TreeNode->val-1);
                Check_BST(TreeNode->right, TreeNode->val+1, max);
}



bool Is_BST(TreeNode* TreeNode)
{
        return(Check_BST(TreeNode, INT_MIN, INT_MAX));
}
```

```
int main()
{
        TreeNode *root = new TreeNode(5);
        root->left = new TreeNode(4);
        root->right = new TreeNode(10);
        root->right->left = new TreeNode(16);
        root->left->left = new TreeNode(6);
        root->left->right = new TreeNode(2);

        if(Is_BST(root))
                cout<<"\n\n\t\t GIVEN TREE IS BINARY SEARCH TREE \n\n\n";
        else
                cout<<"\n\n\t\t GIVEN TREE IS NOT A BINARY SEARCH TREE
\n\n\n";

        return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe

                GIVEN TREE IS NOT A BINARY SEARCH TREE

PS C:\Users\NISHANT\Documents>
```

6. Applications:-

1. It is used to efficiently store data in sorted form in order to access and search stored elements quickly.
2. They can be used to represent arithmetic expressions (Refer here for more info)
3. BST used in Unix kernels for managing a set of virtual memory areas

Program-18

1. Objective: -

Write a program to implement insertion in AVL tree.

2. Theory:-

AVL tree is a self-balancing Binary Search Tree (BST) where difference between heights of left and right subtrees cannot be more than one for all nodes.To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. two basic operations that are performed to re-balance a BST without violating the BST property are Left Rotation and Right Rotation.

3. Algorithm:-

```
1. Insert the node in AVL tree using  same insertion algorithm of BST.
2. The balance factor of each node is updated.
3. Check if any node violates  range of  balance factor
4. If balance factor is violated, then perform rotations
    a)  If BF(node) = +2 and BF(left-child) = +1, perform LL rotation.
    b)  If BF(node) = -2 and BF(right-child) = 1, perform RR rotation.
    c)  If BF(node) = -2 and BF(right-child) = +1, perform RL rotation.
    d)  If BF(node) = +2 and BF(left-child) = -1, perform LR rotation.
```

4. Code: -

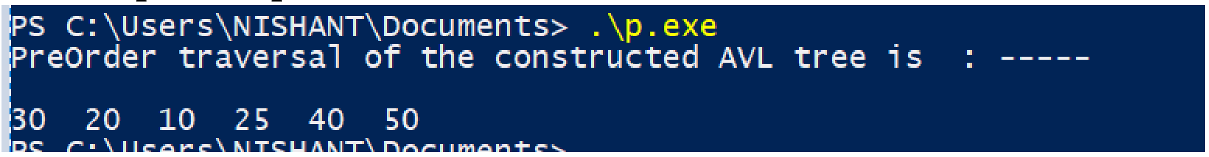
```
    #include<bits/stdc++.h>
using namespace std;

class TreeNode
{
        public:
```

```
{
    int val;
    TreeNode *left;
    TreeNode *right;
    int height;
};



int height(TreeNode *cur)
{
    if (cur == NULL)
        return 0;
    return cur->height;
}




TreeNode* newNode(int val)
{
    TreeNode* root = new TreeNode();
    root->val = val;
    root->left = NULL;
    root->right = NULL;
    root->height = 1;

    return(root);
}

TreeNode *Rotate_right(TreeNode *y)
{
	TreeNode *x = y->left;
	TreeNode *T2 = x->right;

	x->right = y;
	y->left = T2;

	y->height = max(height(y->left),
					height(y->right)) + 1;
	x->height = max(height(x->left),
					height(x->right)) + 1;

	return x;
}

TreeNode *Rotate_left(TreeNode *x)
{
	TreeNode *y = x->right;
	TreeNode *T2 = y->left;

	y->left = x;
	x->right = T2;

	x->height = max(height(x->left),
					height(x->right)) + 1;
	y->height = max(height(y->left),
					height(y->right)) + 1;

	return y;
}
```

```
int Balance_Tree(TreeNode *cur)
{
        if (cur == NULL)
                return 0;
        return height(cur->left) - height(cur->right);
}

TreeNode* Insert_AVL(TreeNode* TreeNode, int val)
{
        if (TreeNode == NULL)
                return(newNode(val));

        if (val < TreeNode->val)
                TreeNode->left = Insert_AVL(TreeNode->left, val);
        else if (val > TreeNode->val)
                TreeNode->right = Insert_AVL(TreeNode->right, val);
        else
                return TreeNode;

        TreeNode->height = 1 + max(height(TreeNode->left),
                                        height(TreeNode->right));

        int balance = Balance_Tree(TreeNode);

        if (balance > 1 && val < TreeNode->left->val)
                return Rotate_right(TreeNode);

        if (balance < -1 && val > TreeNode->right->val)
```

```
                return Rotate_left(TreeNode);

        if (balance > 1 && val > TreeNode->left->val)
        {
                TreeNode->left = Rotate_left(TreeNode->left);
                return Rotate_right(TreeNode);
        }

        if (balance < -1 && val < TreeNode->right->val)
        {
                TreeNode->right = Rotate_right(TreeNode->right);
                return Rotate_left(TreeNode);
        }

        return TreeNode;
}

void PreOrder_AVL(TreeNode *root)
{
        if(root != NULL)
        {
                cout << root->val <<"  ";
                PreOrder_AVL(root->left);
                PreOrder_AVL(root->right);
        }
}

int main()
{
        TreeNode *root = NULL;


        root = Insert_AVL(root, 10);

        root = Insert_AVL(root, 20);

        root = Insert_AVL(root, 30);

        root = Insert_AVL(root, 40);

        root = Insert_AVL(root, 50);

        root = Insert_AVL(root, 25);


        /* The constructed AVL Tree would be

                        30

                       / \

                       20 40

                       / \ \

                  10 25 50

        */

        cout << "PreOrder traversal of the "

                       "constructed AVL tree is  : -----  \n\n";

        PreOrder_AVL(root);


        return 0;

}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe
PreOrder traversal of the constructed AVL tree is  : -----

30  20  10  25  40  50
PS C:\Users\NISHANT\Documents>
```

6. Applications:-

AVL Trees seem to be the best data structure for Database Theory
AVL Trees are used for all sorts of in-memory collections such as sets and dictionaries.

Program-19

1. Objective: -

Write a program to count the number of leaf nodes in an AVL tree.

2. Theory:-

We will be given a AVL Tree and we have to create a C++ program which counts the total number of leaf nodes present in it using recursion. A leaf node is one which has no child. It is also known as internal node.

We can easily find the number of leaf nodes present in any tree using recursion. A leaf node is a node whose left and right child are NULL. We just need to check this single condition to determine whether the node is a leaf node or a non leaf (internal) node.

3. Algorithm:-

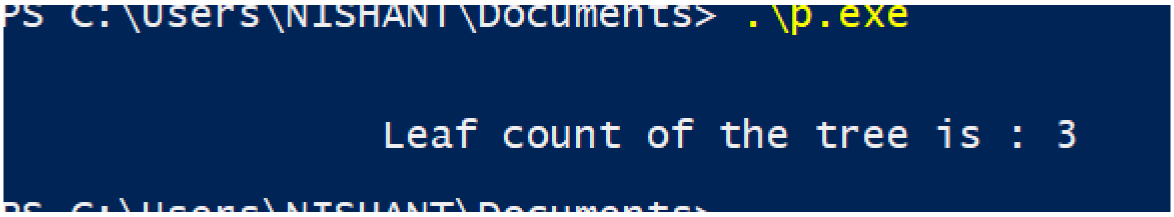
```
1. if the root is null then return zero.
2. start the count with zero
3. push the root into Stack
4. loop until Stack is not empty
5. pop the last node and push left and right children of the last node if they are not null.
Increase the count
```

4. Code: -

```
#include <bits/stdc++.h>

using namespace std;


class TreeNode

{   public:

        int val;

        class TreeNode* left;

        class TreeNode* right;
```

```
};

 int Count_leaves(class TreeNode* TreeNode)
{
        if(TreeNode == NULL)
                return 0;
        if(TreeNode->left == NULL && TreeNode->right == NULL)
                return 1;
        else
                return Count_leaves(TreeNode->left)+
                        Count_leaves(TreeNode->right);
}

class TreeNode* newNode(int val)
{
        class TreeNode* TreeNode = (class TreeNode*)
                                        malloc(sizeof(class TreeNode));
        TreeNode->val = val;
        TreeNode->left = NULL;
        TreeNode->right = NULL;

return(TreeNode);
}

int main()
{
        class TreeNode *root = newNode(5);
        root->left = newNode(3);
        root->right = newNode(6);
        root->left->left = newNode(1);

        root->left->right = newNode(2);


cout << "\n\n\t\tLeaf count of the tree is : "<<

                                        Count_leaves(root) << endl<<endl;

return 0;

}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe


                Leaf count of the tree is : 3

PS C:\Users\NISHANT\Documents>
```

6. Applications:-

1. AVL Tree used is used for indexing large records in database to improve search.
2. AVL trees are beneficial in the cases where you are designing some database where insertions and deletions are not that frequent but you have to frequently look-up for the items present in there.

Program-20

1. **Objective: -**

Write a program to Delete a key from the AVL tree.

2. **Theory:-**

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)).
1) Left Rotation
2) Right Rotation

3. **Algorithm:-**

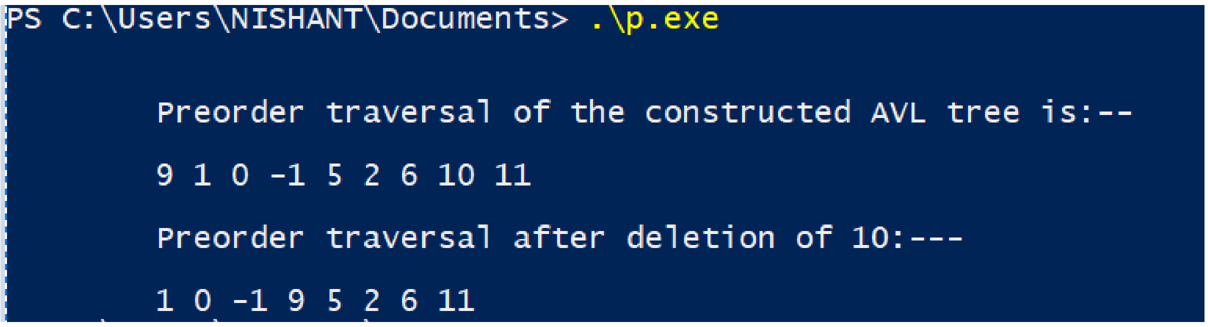
1. Perform the normal BST deletion.
2. The current node must be one of the ancestors of the deleted node. Update the height of the current node.
3. Get the balance factor (left subtree height – right subtree height) of the current node.
4. If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
5. If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

4. **Code: -**

```
#include<bits/stdc++.h>
using namespace std;

class TreeNode
{
        public:
        int val;
        TreeNode *left;
        TreeNode *right;
        int height;
};


int height(TreeNode *cur)
{
        if (cur == NULL)
                return 0;
        return cur->height;
}


TreeNode* New_Node(int val)
{
        TreeNode* cur = new TreeNode();
        cur->val = val;
        cur->left = NULL;
        cur->right = NULL;
        cur->height = 1;
        return(cur);
}

TreeNode *Rotate_Right(TreeNode *y)
{
        TreeNode *x = y->left;
        TreeNode *T2 = x->right;

        x->right = y;
        y->left = T2;

        y->height = max(height(y->left),
                                height(y->right)) + 1;
        x->height = max(height(x->left),
                                height(x->right)) + 1;

        return x;
}

TreeNode *Rotate_left(TreeNode *x)
{
        TreeNode *y = x->right;
        TreeNode *T2 = y->left;

        y->left = x;
        x->right = T2;

        x->height = max(height(x->left),
                                height(x->right)) + 1;
        y->height = max(height(y->left),
                                        height(y->right)) + 1;

        return y;
}

int Balance_get(TreeNode *cur)
{
        if (cur == NULL)
                return 0;
        return height(cur->left) -
                height(cur->right);
}

TreeNode* Insertion(TreeNode* TreeNode, int val)
{
        if (TreeNode == NULL)
                return(New_Node(val));

        if (val < TreeNode->val)
                TreeNode->left = Insertion(TreeNode->left, val);
        else if (val > TreeNode->val)
                TreeNode->right = Insertion(TreeNode->right, val);
        else
                return TreeNode;

        TreeNode->height = 1 + max(height(TreeNode->left),
                                                height(TreeNode->right));
        int balance = Balance_get(TreeNode);


        if (balance > 1 && val < TreeNode->left->val)
                return Rotate_Right(TreeNode);

        if (balance < -1 && val > TreeNode->right->val)
                return Rotate_left(TreeNode);

        if (balance > 1 && val > TreeNode->left->val)
        {
                TreeNode->left = Rotate_left(TreeNode->left);
                return Rotate_Right(TreeNode);
        }

        if (balance < -1 && val < TreeNode->right->val)
        {
                TreeNode->right = Rotate_Right(TreeNode->right);
                return Rotate_left(TreeNode);
        }

        return TreeNode;
}

TreeNode * Min_Val_Node(TreeNode* c)
{
        TreeNode* cur = c;

        while (cur->left != NULL)
```

```
                cur = cur->left;

        return cur;
}

TreeNode* Delete_Node(TreeNode* root, int val)
{

        if (root == NULL)
                return root;

        if ( val < root->val )
                root->left = Delete_Node(root->left, val);

        else if( val > root->val )
                root->right = Delete_Node(root->right, val);

        else
        {
                if( (root->left == NULL) ||
                        (root->right == NULL) )
                {
                        TreeNode *temp = root->left ?
                                                    root->left :
                                                    root->right;

                        if (temp == NULL)
                        {
                                temp = root;
                root = NULL;
            }
            else
            *root = *temp;

            free(temp);
        }
        else
        {
            TreeNode* temp = Min_Val_Node(root->right);

            root->val = temp->val;

            root->right = Delete_Node(root->right,
                                                        temp->val);
        }
    }

    if (root == NULL)
    return root;

    root->height = 1 + max(height(root->left),
                                        height(root->right));

    int balance = Balance_get(root);

    if (balance > 1 &&
        Balance_get(root->left) >= 0)
        return Rotate_Right(root);
        if (balance > 1 &&
                Balance_get(root->left) < 0)
        {
                root->left = Rotate_left(root->left);
                return Rotate_Right(root);
        }

        if (balance < -1 &&
                Balance_get(root->right) <= 0)
                return Rotate_left(root);

        if (balance < -1 &&
                Balance_get(root->right) > 0)
        {
                root->right = Rotate_Right(root->right);
                return Rotate_left(root);
        }

        return root;
}

void preOrder(TreeNode *root)
{
        if(root != NULL)
        {
                cout << root->val << " ";
                preOrder(root->left);
                preOrder(root->right);
        }
}

int main()
{
TreeNode *root = NULL;

        root = Insertion(root, 9);
        root = Insertion(root, 5);
        root = Insertion(root, 10);
        root = Insertion(root, 0);
        root = Insertion(root, 6);
        root = Insertion(root, 11);
        root = Insertion(root, -1);
        root = Insertion(root, 1);
        root = Insertion(root, 2);

        /* The constructed AVL Tree would be
      9
                / \
                1 10
                / \ \
          0   5 11
        / / \
  -1 2  6
        */

        cout << "\n\n\tPreorder traversal of the "
                        "constructed AVL tree is:-- \n\n\t";
        preOrder(root);

        root = Delete_Node(root, 10);

        /* The AVL Tree after deletion of 10
                1
                / \
                0 9
                / / \
          -1  5   11
                / \
                2 6
        */

        cout << "\n\n\tPreorder traversal after"
                << " deletion of 10:--- \n\n\t";
        preOrder(root);

        return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe

        Preorder traversal of the constructed AVL tree is:--

        9 1 0 -1 5 2 6 10 11

        Preorder traversal after deletion of 10:---

        1 0 -1 9 5 2 6 11
```

6. Applications:-

AVL Tree used is used for indexing large records in database to improve search. AVL trees are beneficial in the cases where you are designing some database where insertions and deletions are not that frequent but you have to frequently look-up for the items present in there.

Program-21

1. Objective: -

Write a program to implement Stack Data Structure using Queue

2. Theory:-

A stack can be implemented using two queues. Let stack to be implemented be ‘s’ and queues used to implement be ‘q1’ and ‘q2’. Stack ‘s’ can be implemented in two ways:

Method 1 (By making push operation costly)
This method makes sure that newly entered element is always at the front of ‘q1’, so that pop operation just dequeues from ‘q1’. ‘q2’ is used to put every new element at front of ‘q1’.
push(s, x) operation’s step are described below:
Enqueue x to q2
One by one dequeue everything from q1 and enqueue to q2.
Swap the names of q1 and q2
pop(s) operation’s function are described below:
Dequeue an item from q1 and return it.

Method 2 (By making pop operation costly)
In push operation, the new element is always enqueued to q1. In pop() operation, if q2 is empty then all the elements except the last, are moved to q2. Finally the last element is dequeued from q1 and returned.
push(s, x) operation:
Enqueue x to q1 (assuming size of q1 is unlimited).
pop(s) operation:
One by one dequeue everything except the last element from q1 and enqueue to q2.
Dequeue the last item of q1, the dequeued item is result, store it.
Swap the names of q1 and q2
Return the item stored in step 2.

3. Algorithm:-

Push:

1. The new element is always added to the rear of queue q1 and it is kept as top stack element.

Pop:

1. The last inserted element in q2 is kept as top.

2. Then the algorithm removes the last element in q1.

3. We swap q1 with q2 to avoid copying all elements from q2 to q1.

4. Code: -

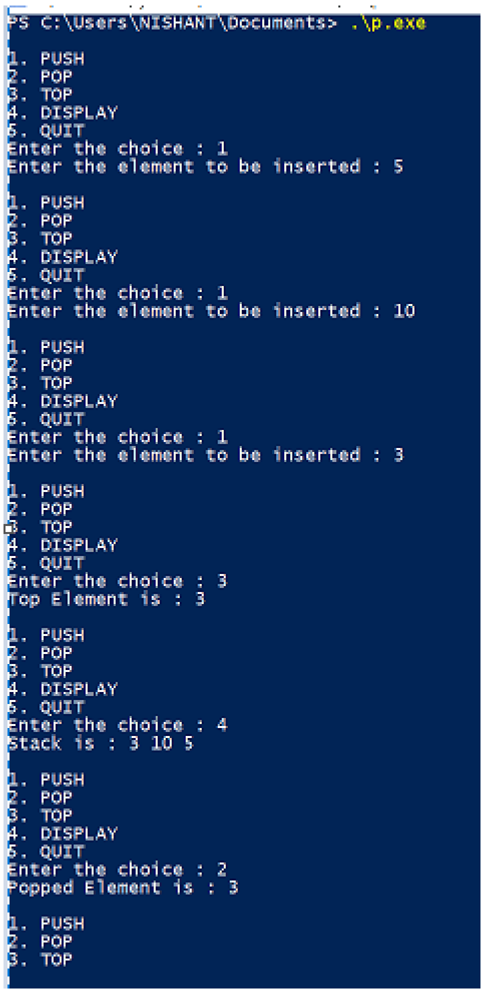
```
#include <iostream>
#include <conio.h>
#include <stdlib.h>

using namespace std;

struct Node
{
   int info;
   Node *link;
};

class Queue
{
   private:
      Node *front = NULL;
      Node *rear = NULL;
   public:
      void enqueue(int item);
      int dequeue();
      int peek();
      int isEmpty();
      void display();

};

void Queue::enqueue(int item)
{
   Node *p;
   p = (Node *)malloc(sizeof(Node));
   if(p == NULL)
   {
      cout<<"Press a key"<<endl;
   }
```

```
    else
    {
        p -> info = item;
        p -> link = NULL;
        if(front == NULL)
        {
            front = p;
            rear = p;
        }
        else
        {
            rear -> link = p;
            rear = p;
        }
    }
}

int Queue::dequeue()
{
    int item;
    if(front == NULL)
    {
        cout<<"Press a key"<<endl;
        getch();
        exit(1);
    }
    else
    {
        item = front -> info;
        front = front -> link;
        return item;
    }
}

int Queue::peek()
{
    int item;
    if(front ==  NULL)
    {
        cout<<"Press a key"<<endl;
        getch();
        exit(1);
    }
    else
    {
        item = front -> info;
        return item;
    }
}

int Queue::isEmpty()
{
    if(front == NULL)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

void Queue::display()
{
    Node *p;
    if(front == NULL)
    {
        cout<<"Stack is Empty"<<endl;
        getch();
        exit(1);
    }
    else
    {
        p = (Node *)malloc(sizeof(Node));
        p = front;
        cout<<"Stack is : ";
        while(p!=NULL)
        {
            cout<<p -> info<<" ";
            p = p -> link;
        }
        cout<<endl;
    }
}


class Stack
{
    private:
        Queue q1,q2;
    public:
        void push(int x);
        int pop();
        int top();
        void display();
};


void Stack::push(int x)
{
    q2.enqueue(x);
    while(!q1.isEmpty())
    {
        q2.enqueue(q1.peek());
        q1.dequeue();
    }
    Queue q;
    q = q1;
    q1 = q2;
    q2 = q;
}

int Stack::pop()
{
    if(q1.isEmpty())
    {
        cout<<"Stack is Empty"<<endl;
        getch();
        exit(1);
    }
    else
    {
        return q1.dequeue();
    }
}

int Stack::top()
{
    if(q1.isEmpty())
    {
        cout<<"Stack is Empty"<<endl;
        getch();
        exit(1);
    }
    else
    {
        return q1.peek();
    }
```

```
}

void Stack::display()
{
   q1.display();
}

int main()
{
   int choice;
   Stack s1;
   while(1)
   {
      cout<<endl;
      cout<<"1. PUSH"<<endl;
      cout<<"2. POP"<<endl;
      cout<<"3. TOP"<<endl;
      cout<<"4. DISPLAY"<<endl;
      cout<<"5. QUIT"<<endl;
      cout<<"Enter the choice : ";
      cin>>choice;
      switch(choice)
      {
         case 1:
            int d1;
            cout<<"Enter the element to be inserted : ";
            cin>>d1;
            s1.push(d1);
            break;
         case 2:
            int d2;
            d2= s1.pop();
            cout<<"Popped Element is : ";
            cout<<d2<<endl;
            break;
         case 3:
            int d3;
            d3 = s1.top();
            cout<<"Top Element is : ";
            cout<<d3<<endl;
            break;
         case 4:
            s1.display();
            break;
         case 5:
            cout<<"Thanks!!!!!"<<endl;
```

```
                getch();
                exit(0);
            default:
                cout<<"Wrong Choice!!!!!"<<endl;
                cout<<"Try Again....."<<endl;
                break;
        }
    }
    return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe

1. PUSH
2. POP
3. TOP
4. DISPLAY
5. QUIT
Enter the choice : 1
Enter the element to be inserted : 5

1. PUSH
2. POP
3. TOP
4. DISPLAY
5. QUIT
Enter the choice : 1
Enter the element to be inserted : 10

1. PUSH
2. POP
3. TOP
4. DISPLAY
5. QUIT
Enter the choice : 1
Enter the element to be inserted : 3

1. PUSH
2. POP
3. TOP
4. DISPLAY
5. QUIT
Enter the choice : 3
Top Element is : 3

1. PUSH
2. POP
3. TOP
4. DISPLAY
5. QUIT
Enter the choice : 4
Stack is : 3 10 5

1. PUSH
2. POP
3. TOP
4. DISPLAY
5. QUIT
Enter the choice : 2
Popped Element is : 3

1. PUSH
2. POP
3. TOP
```

6. Applications:-

1. Stacks can be used for expression evaluation.
2. Stacks can be used to check parenthesis matching in an expression.
3. Stacks can be used for Conversion from one form of expression to another.

Program-22

1. Objective: -

Write a program to implement Queue Data Structure using Stack

2. Theory:-

Method 1 (By making enqueue operation costly)

This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

Method 2 (By making dequeue operation costly)

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

3. Algorithm:-

Method 1:

Enqueue()

1. While stack1 is not empty, push everything from stack1 to stack2.
2. Push x to stack1 (assuming size of stacks is unlimited).
3. Push everything back to stack1.

Dequeue(q):

1. If stack1 is empty then error
2. Pop an item from stack1 and return it

Method 2:

Enqueue(q, x)

```
  1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

Dequeue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty
       While stack1 is not empty, push everything from stack1 to stack2.
  3) Pop the element from stack2 and return it.
```

4. Code: -

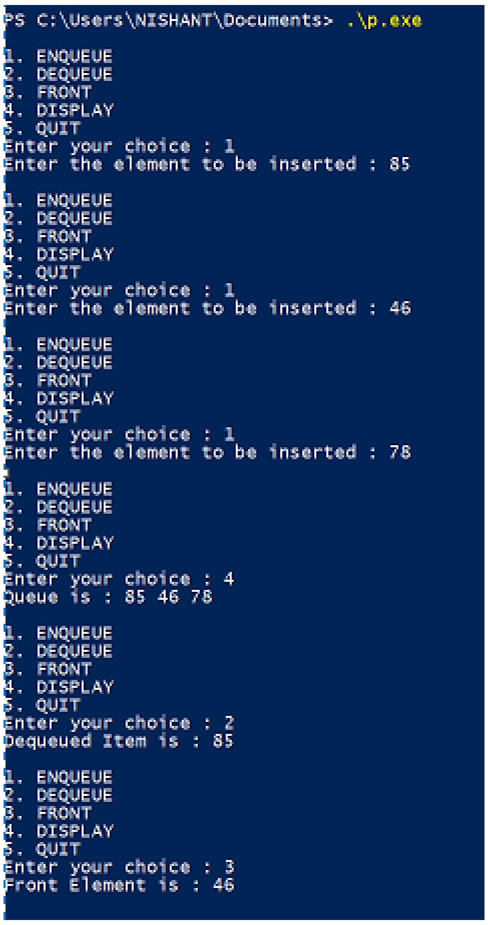
```
#include <iostream>
#include <conio.h>
#include <stdlib.h>

using namespace std;



struct Node
{
    int info;
    Node *link;
};

class Stack
{
    private:
        Node *top = NULL;
    public:
        void push(int item);
        int pop();
        int peek();
        void display();
        int isEmpty();
};

void Stack::push(int item)
{
    Node *p;
    p = (Node*)malloc(sizeof(Node));
    if(p == NULL)
    {
        cout<<"Press a Key"<<endl;
    }
    else
    {
        p -> info = item;
        p -> link = top;
        top = p;
    }
}

int Stack::pop()
{
    int item;
    Node *p;
    p = (Node *)malloc(sizeof(Node));
    if(top==NULL)
    {
        cout<<"Press a Key"<<endl;
        getch();
        exit(1);
    }
    else
    {
        p = top;
        item = p -> info;
        top = p -> link;
        return item;
    }
}

int Stack::peek()
{
    int item;
    if(top == NULL)
    {
        cout<<"Press a Key";
        getch();
        exit(1);
    }
    else
    {
        item = top -> info;
        return item;
    }
```

```
}

void Stack::display()
{
    Node *p;
    if(top == NULL)
    {
        cout<<"Queue is Empty"<<endl;
        getch();
        exit(1);
    }
    else
    {
        p = (Node *)malloc(sizeof(Node));
        p = top;
        cout<<"Queue is : ";
        while(p != NULL)
        {
            cout<< p -> info<<" ";
            p = p -> link;
        }
        cout<<endl;
    }
}

int Stack::isEmpty()
{
    if(top==NULL)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

class Queue
{
    private:
        Stack s1,s2;
    public:
        void enqueue(int item);
        int dequeue();
        int front();
        void display();
};

void Queue::enqueue(int item)
{
    while(!s1.isEmpty())
    {
        s2.push(s1.peek());
        s1.pop();
    }
    s1.push(item);
    while(!s2.isEmpty())
    {
        s1.push(s2.peek());
        s2.pop();
    }
}

int Queue::dequeue()
{
    int x;
    if(s1.isEmpty())
    {
        cout<<"Queue is Empty"<<endl;
        getch();
        exit(1);
    }
    else
    {
        x = s1.pop();
    }
    return x;
}

int Queue::front()
{
    int x;
    if(s1.isEmpty())
    {
        cout<<"Queue is Empty"<<endl;
        getch();
        exit(1);
    }
    else
    {
        x = s1.peek();
    }
    return x;
}

void Queue::display()
{
    s1.display();
}

int main()
{
    Queue q1;
    int choice;
    while(1)
    {
        cout<<endl;
        cout<<"1. ENQUEUE"<<endl;
        cout<<"2. DEQUEUE"<<endl;
        cout<<"3. FRONT"<<endl;
        cout<<"4. DISPLAY"<<endl;
        cout<<"5. QUIT"<<endl;

        cout<<"Enter your choice : ";
        cin>>choice;
```

```
switch(choice)
{
    case 1:
        int d1;
        cout<<"Enter the element to be inserted : ";
        cin>>d1;
        q1.enqueue(d1);
        break;
    case 2:
        int d2;
        d2 = q1.dequeue();
        cout<<"Dequeued Item is : ";
        cout<<d2<<endl;
        break;
    case 3:
        int d3;
        d3 = q1.front();
        cout<<"Front Element is : ";
        cout<<d3<<endl;
        break;
    case 4:
        q1.display();
        break;
    case 5:
        cout<<"Thanks!!!!!"<<endl;
        getch();
        exit(0);
    default:
        cout<<"Wrong Choice!!!!!"<<endl;
```

```
                cout<<"Try Again....."<<endl;
                break;
            }
        }
        return 0;
    }
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\p.exe

1. ENQUEUE
2. DEQUEUE
3. FRONT
4. DISPLAY
5. QUIT
Enter your choice : 1
Enter the element to be inserted : 85

1. ENQUEUE
2. DEQUEUE
3. FRONT
4. DISPLAY
5. QUIT
Enter your choice : 1
Enter the element to be inserted : 46

1. ENQUEUE
2. DEQUEUE
3. FRONT
4. DISPLAY
5. QUIT
Enter your choice : 1
Enter the element to be inserted : 78

1. ENQUEUE
2. DEQUEUE
3. FRONT
4. DISPLAY
5. QUIT
Enter your choice : 4
Queue is : 85 46 78

1. ENQUEUE
2. DEQUEUE
3. FRONT
4. DISPLAY
5. QUIT
Enter your choice : 2
Dequeued Item is : 85

1. ENQUEUE
2. DEQUEUE
3. FRONT
4. DISPLAY
5. QUIT
Enter your choice : 3
Front Element is : 46
```

6. Applications:-

1. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
2. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Program-23

1. Objective: -

Write a program to implement Graph Data Structure and Its traversal BFS and DFS

2. Theory:-

A graph data structure consists of a finite (and possibly mutable) set of vertices (also called nodes or points), together with a set of unordered pairs of these vertices for an undirected graph or a set of ordered pairs for a directed graph. These pairs are known as edges (also called links or lines), and for a directed graph are also known as arrows. The vertices may be part of the graph structure, or may be external entities represented by integer indices or references.

Traversal means visiting all the nodes of a graph.

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a search key), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

3. Algorithm:-

1. Start by putting any one of the graph's vertices at the back of a queue
2. Take the front item of the queue and add it to the visited list
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue
4. Keep repeating steps 2 and 3 until the queue is empty.
   //dfs
5. Start by putting any one of the graphs vertices on top of a stack
6. Take the top item of the stack and add it to the visited list
7. Create a list of the vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack

8. Keep repeating steps 6 and 7 until the stack is empty.

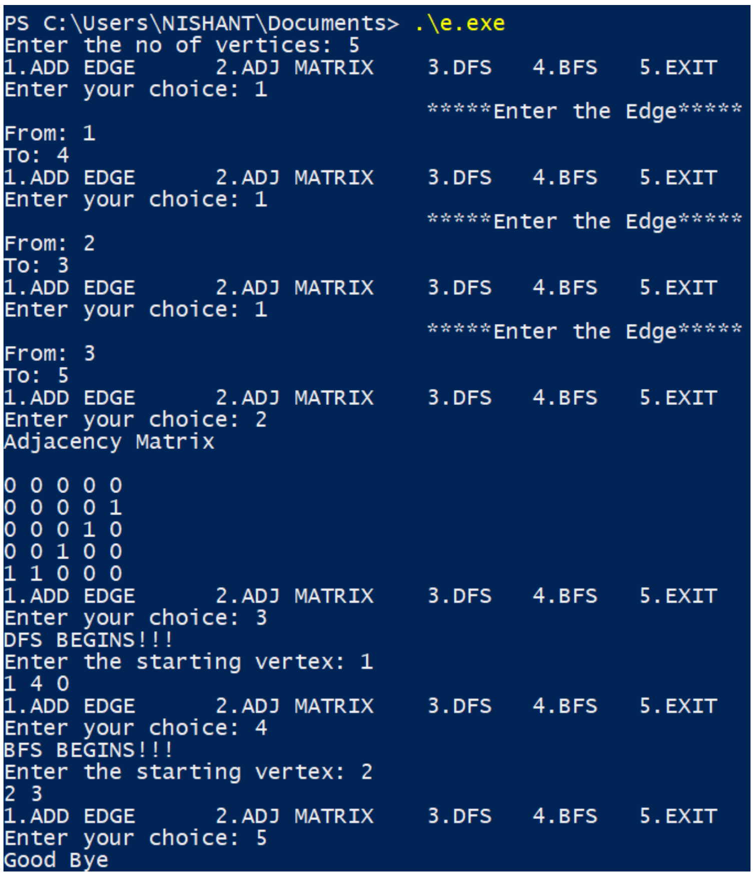
4. Code: -

```
#include <iostream>
using namespace std;
struct node
{
  int data;
  node* next;
};
node* head=0;
node* last=0;
void enqueue(int x)
{
  if(head==0)
  {
    node* newnode= new node;
    newnode->data=x;
    newnode->next=0;
    head=last=newnode;
  }
  else
  {
    node* newnode=new node;
    newnode->data=x;
    newnode->next=0;
    last->next=newnode;
    last=newnode;
  }
```

```
}

void dequeue()
{
  node* temp=head;
  head=temp->next;
  delete temp;
}

int front()
{
  return head->data;
}

int empty()
{
  int empty;
  if(head==0)
   empty=0;
  else
    empty=1;

  return empty;
}
void add_edge(int* a,int n)
{
  int u,v;
  cout<<"\t\t\t\t*****Enter the Edge*****\n";
  cout<<"From: ";
  cin>>u;
  cout<<"To: ";
  cin>>v;
  *((a+u*n)+v)=1;
  *((a+v*n)+u)=1;
}

void adj_matrix (int * a,int n)
{
  cout<<"Adjacency Matrix\n\n";
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<n;j++)
    {
      cout<<*((a+i*n)+j)<<" ";
    }
    cout<<endl;
  }
}

void dfs(int* a,int n,int start,int visited[])
{

  if(visited[start]==0)
  {
    cout<<start<<" ";
    visited[start]=1;
    for(int j=0;j<n;j++)
    {
      if(*((a+start*n)+j)==1&&visited[j]==0)
       dfs((int*) a,n,j,visited);
    }
  }
}

void bfs(int *a,int n,int start,int visited[] )
{
  enqueue(start);
  visited[start]=1;
  while(empty()!=0)
  {
    cout<<front()<<" ";
    int j=front();
    dequeue();
    for(int i=0;i<n;i++)
    {
      if(*((a+j*n)+i)==1&&visited[i]==0)
      {
        enqueue(i);
        visited[i]=1;
      }
    }
  }
}

int main() {
  int n,x;
  cout<<"Enter the no of vertices: ";
```

```
cin>>n;
int a[n][n];
for(int i=0;i<n;i++)
{
  for(int j=0;j<n;j++)
    a[i][j]=0;
}
int visited [n];

do
{
  cout<<"1.ADD EDGE\t2.ADJ MATRIX\t3.DFS\t4.BFS\t5.EXIT\n";
  cout<<"Enter your choice: ";
  cin>>x;
  switch(x)
  {
    case 1:
    {
      add_edge(&a[0][0],n);
      break;
    }
    case 2:
    {
      adj_matrix((int *)a,n);
      break;
    }
    case 3:
    {
      int start;
  for(int i=0;i<n;i++)
    visited[i]=0;
  cout<<"DFS BEGINS!!!\n";
  cout<<"Enter the starting vertex: ";
  cin>>start;
  dfs((int*)a,n,start,visited);
  cout<<endl;
  break;
}
case 4:
{

  int start;
  for(int i=0;i<n;i++)
    visited[i]=0;
  cout<<"BFS BEGINS!!!\n";
  cout<<"Enter the starting vertex: ";
  cin>>start;
  bfs((int *)a,n,start,visited);
  cout<<endl;
  break;
}
case 5:
{
  cout<<"Good Bye\n";
  break;
}
default:
{
        cout<<"Invalid Choice!!\n";
      }
    }
  }
  while(x!=5);


  return 0;
}
```

5. Input/Output:-



```
PS C:\Users\NISHANT\Documents> .\e.exe
Enter the no of vertices: 5
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 1
                                *****Enter the Edge*****
From: 1
To: 4
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 1
                                *****Enter the Edge*****
From: 2
To: 3
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 1
                                *****Enter the Edge*****
From: 3
To: 5
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 2
Adjacency Matrix

0 0 0 0 0
0 0 0 0 1
0 0 0 1 0
0 0 1 0 0
1 1 0 0 0
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 3
DFS BEGINS!!!
Enter the starting vertex: 1
1 4 0
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 4
BFS BEGINS!!!
Enter the starting vertex: 2
2 3
1.ADD EDGE      2.ADJ MATRIX    3.DFS   4.BFS   5.EXIT
Enter your choice: 5
Good Bye
```

6. Applications:-

Graph

1. In Computer science graphs are used to represent the flow of computation.
2. Google maps uses graphs for building transportation systems

3. In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.
4. In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices.

<u>Bfs</u>

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In minimum spanning tree

<u>Dfs</u>

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph