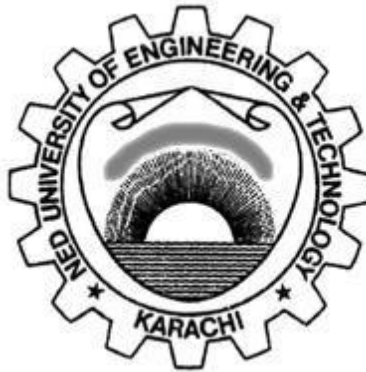


Workbook

Data Structure Algorithm & Implementation (CT – 157)



Name: Group ID - 02 (Ali Zia Khan, M. Anas Bin Ateeq, Maheen Bukhtiar, Mesum Sultan, Tooba Izzat)

Roll No. : SE-052, SE-055, SE-069, SE-096, SE-100

Batch: 2019

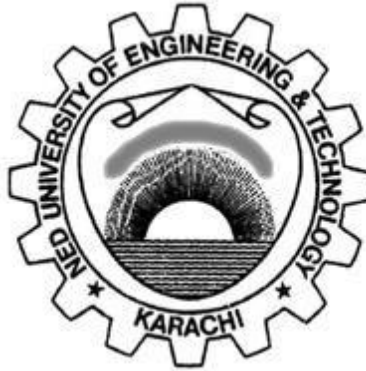
Year: Second

Department: Software Engineering

Workbook

Data Structure Algorithm & Implementation

(CT – 157)



Prepared by
Engr. Sana Fatima
Lecturer- SE

Approved by
Chairman

Department of Software Engineering

Table of Contents

S. No.	Object	Page No.	Signature
01	To learn the basic concepts of Data Structure & Algorithms	4	
02	Array data structures & Operations i. Insertion ii. Deletion iii. Traversing	6	
03	Searching Algorithms i. Linear Search ii. Binary Search	15	
04	Sorting Algorithms i. Bubble Sort Algorithm ii. Quick Sort Algorithm	20	
05	Stack data structure & Operations i. Push ii. Pop	24	
06	Expression Evaluation through Stack Data Structure i. Infix ii. Postfix iii. Prefix	28	
07	Queue data structure & Operations i. Enqueue ii. Dequeue	36	
08	Recursive Algorithms (Recursion) Tower of Hanoi Problem	42	
09	Tree data structure & Operations i. Insertion ii. Deletion	48	
10	Tree Traversal Algorithms i. Inorder ii. Preorder iii. Postorder	52	
11	Graph data structure & Operations i. Adjacency List ii. Adjacency Matrix	59	
12	Graph Traversal Algorithms i. DFS ii. BFS	65	
13	Rat in a Maze Path finding problem	76	
14	N-Queen Problem	79	

LAB NO 1

To learn the basic concepts of Data Structure & Algorithms

(Done by Tooba Izat, SE-100)

QUESTION NO 1:

Following are some characteristics of an algorithm

Unambiguous:

The instructions or steps of the algorithm should be clear and precise. Moreover, the inputs and outputs should also be unambiguous and lead to one meaning only.

Input:

The input is the raw data which is processed and converted into desired result. The input should be well defined. There can be 0 or more inputs but should be well defined

Output:

An algorithm may have one or more well defines output. Output of an algorithm is the desired result of it.

Finiteness:

An algorithm should have finite no of steps like it should not have loops in which the condition is never false hence it continues forever.

Feasibility:

The algorithm should be easily implemented in different programming languages. It should not require high no of sources to be implemented.

Independent:

The step-by-step instructions of an algorithm should be able to be implemented in any of the programming language. We should be able to convert an algorithm into a no of languages.

QUESTION NO 2:

Linear data structures:

A linear data structures stores data elements in a sequential manner or in an ordered way. These data elements are also stored sequentially in the memory.

Example:

Array,list,stack,queue

Non linear data structure:

Data elements are stored and accessed in a random manner. Any data element can be linked to any other data element.

Example:

Graph,has,heap,trees

QUESTION NO 3:

When data structures are merged with operation then they are called abstract data types. Some commonly used data types are: array, list, map, queue,stack,table,tree etc.

QUESTION NO 4:

By algorithm analysis we mean that we determine or analyses the efficiency and accuracy of an algorithm. We determine efficiency on the basis of time it takes to run an algorithm and the space it requires in memory. Time analysis falls in three categories. The best case which the least time required by the algorithm. The average case which is the average time required for time execution and lastly the worst case the maximum time required for program execution. We represent it by big O notation.

LAB NO. 2

Array data structures & Operations

i. Insertion ii. Deletion iii. Traversing

(Done by Ali Zia Khan, SE-052)

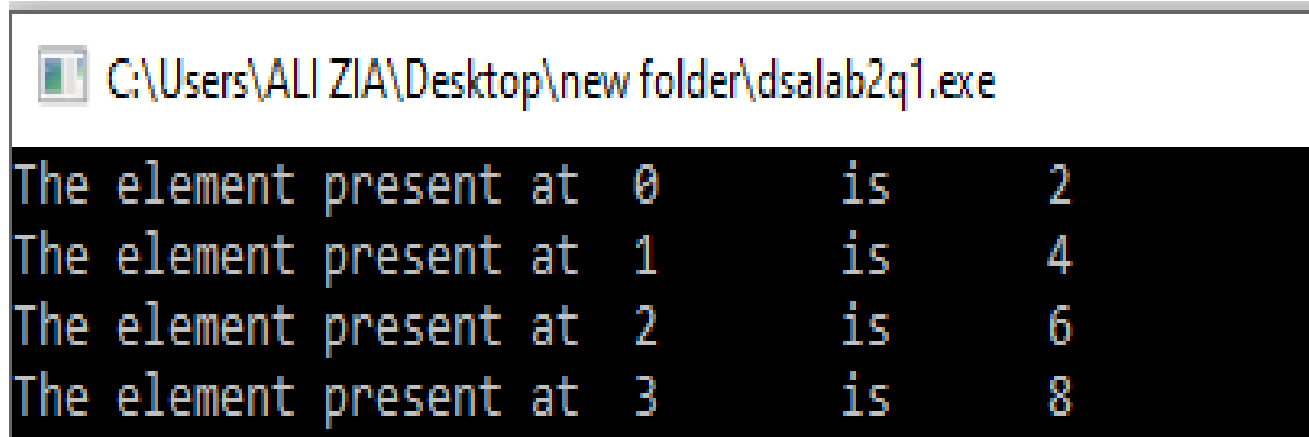
1. Let an array named LA consist of 4 elements 2,4,6 & 8 . write down the code to traverse(or print) all elements in the array.

```
int main(){

    int LA[4]={2,4,6,8};
    int n=(sizeof(LA)/sizeof(LA[1]));
    for(int i=0;i<n;i++){
        cout<<"The element present at\t"<<i<<"\tis"<<"\t"<<LA[i]<<endl;

    }
}
```

OUTPUT:



```
C:\Users\ALI ZIA\Desktop\new folder\dsalab2q1.exe
The element present at 0 is 2
The element present at 1 is 4
The element present at 2 is 6
The element present at 3 is 8
```

2. Use insertion algorithm to add an element (Give implementation)

i. 1 at index 0 ii. 5 at index 2 iii. 3 at index 4

```
#include<iostream>
#include<conio.h>
using namespace std;

int main(){
    int arr1[5]={9,48,7,14,85};
    int s=sizeof(arr1)/sizeof(arr1[1]);
    cout<<"Initially the array was"<<endl;
    for(int i=0;i<s;i++){
        cout<<"The element present at\t"<<i<<"\tis"<<"\t"<<arr1[i]<<endl;
    }
    int elm1,elm2,elm3;
    cout<<"Element to enter at index 0"<<endl;
    cin>>elm1;
    cout<<"Element to enter at index 2"<<endl;
    cin>>elm2;
    cout<<"Element to enter at index 4"<<endl;
    cin>>elm3;

    arr1[0]=elm1;
    arr1[2]=elm2;
    arr1[4]=elm3;
    int z=sizeof(arr1)/sizeof(arr1[1]);
    for(int i=0;i<z;i++){
        cout<<"The element present at\t"<<i<<"\tis"<<"\t"<<arr1[i]<<endl;
    }
}
```

 C:\Users\ALI ZIA\Desktop\new folder\dsaforrunning.exe

Initially the array was

The element present at	0	is	9
The element present at	1	is	48
The element present at	2	is	7
The element present at	3	is	14
The element present at	4	is	85

Element to enter at index 0

1

Element to enter at index 2

5

Element to enter at index 4

3

The element present at	0	is	1
The element present at	1	is	48
The element present at	2	is	5
The element present at	3	is	14
The element present at	4	is	3

Process exited after 10.88 seconds with return value 0

Press any key to continue . . .

3. If the size of given array is 4 then calculate the Time complexity of insertion algorithm

when ,

i. Insert element at the beginning of array

i. Insert element at the end of array

ii. Insert element in the middle of array

ALGORITHM

1. Set $J = N-1$

2. Repeat steps 3 and 4 while $J \geq K$

3. Set $LA[J+1] = LA[J]$

4. Set $J = J-1$

5. Set $LA[K] = \text{ITEM}$

6. Set $N=N+1$

7. Exit

Statement	Operation	Iteration	Cost
1	2	1	2
2	2	$n+1$	$2(n+1)$
3	2	N	$2(n)$
4	2	N	$2(n)$
5	1	1	1
6	2	1	2

$$T(n)=6(n)+7$$

Considering the insertion algorithm we can say that the time complexity of insertion of element in array algorithm is linear. i.e $C(N)+C'$, where N is the no of elements.

The difference between them is only due to no iterations required for insertion .

If No of elements =4:

LOCATION FOR INSERTION	AT START OF ARRAY	IN MIDDLE OF ARRAY	IN END OF ARRAY
NO OF ITERATIONS REQUIRED	4	2	1

Thus the worst case of this algorithm will be when we have to insert the element at start , where no of iterations will be highest .The best case when the no of iterations are least which is when we have to insert at end consequently the insertion at middle will be the average case for this algorithm.

BIG O NOTATION WILL ALSO BE $O(N)$.

4. Consider the array in question

,Use deletion algorithm(Give implementation) to remove an element

- i. 1 at index 0
- ii. 5 at index 2
- iii. 3 at index 4

```

#include<iostream>
#include<conio.h>
using namespace std;

int main(){
    int item,ind;

    int arr1[5]={1,2,5,4,3};
    for(int i=0;i<5;i++){
        cout<<"The element present at\t"<<i<<"\tis"<<"\t"<<arr1[i]<<endl;
    }
    cout<<endl;
    cout<<"Enter position where to delete"<<endl;
    cin>>ind;

    int n=5;
    int j=ind;
    while(j<n){
        arr1[j]=arr1[j+1];
        j++;
    }
    n=n-1;

    for(int i=0;i<n;i++){
        cout<<"The element present at\t"<<i<<"\tis"<<"\t"<<arr1[i]<<endl;
    }
}

```

```

C:\Users\ALI ZIA\Desktop\new folder\dsalab2q4.exe
The element present at 0 is 1
The element present at 1 is 2
The element present at 2 is 5
The element present at 3 is 4
The element present at 4 is 3

Enter position where to delete
0
The element present at 0 is 2
The element present at 1 is 5
The element present at 2 is 4
The element present at 3 is 3

-----
Process exited after 1.796 seconds with return value 0
Press any key to continue . . .

```

```

Select C:\Users\ALI ZIA\Desktop\new folder\dsalab2q4.exe
The element present at 0 is 1
The element present at 1 is 2
The element present at 2 is 5
The element present at 3 is 4
The element present at 4 is 3

Enter position where to delete
2
The element present at 0 is 1
The element present at 1 is 2
The element present at 2 is 4
The element present at 3 is 3

-----
Process exited after 16.91 seconds with return value 0
Press any key to continue . . .

```

```

C:\Users\ALI ZIA\Desktop\new folder\dsalab2q4.exe
The element present at 0 is 1
The element present at 1 is 2
The element present at 2 is 5
The element present at 3 is 4
The element present at 4 is 3

Enter position where to delete
4
The element present at 0 is 1
The element present at 1 is 2
The element present at 2 is 5
The element present at 3 is 4

-----
Process exited after 1.772 seconds with return value 0
Press any key to continue . . .

```

5. If the size of given array is 4 then calculate the Time complexity of deletion algorithm when ,

- i. Delete element at the beginning of array
- iii. Delete element at the end of array
- iv. Delete element in the middle of array

ALGORITHM

1. Set $J = K$
2. Repeat steps 3 and 4 while $J < N-1$
3. Set $LA[J] = LA[J + 1]$
4. Set $J = J+1$
5. Set $N = N-1$
6. Stop

Statement	Operation	Iteration	Cost
1	1	1	1
2	2	N	2N
3	2	N-1	2(N-1)
4	2	N-1	2(N-1)
5	2	1	2

$$T(n)=6N-1$$

Considering the deletion algorithm we can say that the time complexity of deletion of element in array algorithm is linear. i.e $C(N)-C'$, where N is the no of elements as well as no of iterations .

The difference between them is only due to no iterations required for deletion .

If No of elements =4:

LOCATION FOR deletion	AT START OF ARRAY	IN MIDDLE OF ARRAY	IN END OF ARRAY
NO OF ITERARTIONS REQUIRED	3	2	0

Thus the worst case of this algorithm will be when we have to delete the element at start , where no of iterations will be highest.The best case when the no of iterations are least which is when we have to insert at end having no iteration involved consequently the insertion at middle will be the average case for this algorithm.

BIG O NOTATION will also be $O(N)$

LAB NO 3

Searching Algorithms

i. Linear Search ii. Binary Search

(Done by Maheen Bukhtiar, SE-069)

Q1: assume array[] = {2,4,6,8,}. Give implementation of Linear search algorithm to ;

i. find element 8 in array

ii. find element 3 in array

ANSWER:

CODE:

```
#include <iostream>

using namespace std;

int main()
{
    const int MAX = 4;
    bool found = false;
    int i = 0, number, LA[MAX] = { 2, 4, 6, 8 };
    cout << "Enter number to search: ";
    cin >> number;
    while (i < MAX)
    {
        if (number == LA[i])
        {
            cout << "Found at index " << i << ".\n";
            found = true;
            break;
        }
        else if (i == MAX - 1 && found == false)
            cout << "Element not found.\n";
    }
}
```

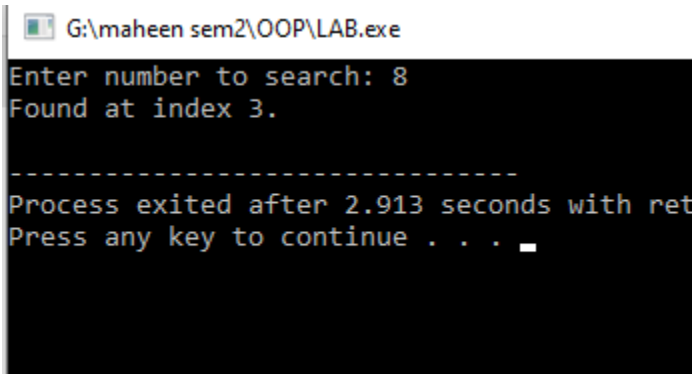
```

        i++;
    }
}

```

OUTPUT:

(i):

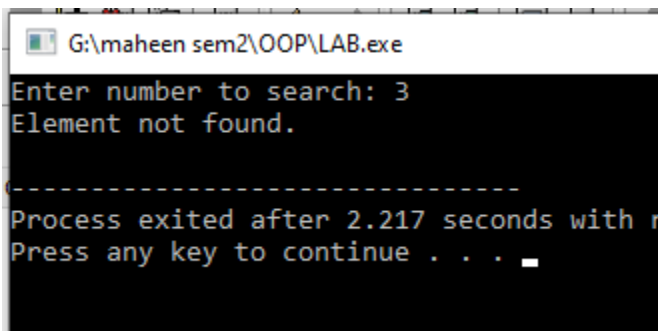


```

G:\maheen sem2\OOP\LAB.exe
Enter number to search: 8
Found at index 3.
-----
Process exited after 2.913 seconds with return code 0
Press any key to continue . . .

```

(ii):



```

G:\maheen sem2\OOP\LAB.exe
Enter number to search: 3
Element not found.
-----
Process exited after 2.217 seconds with return code 0
Press any key to continue . . .

```

Q2. let assume $A[] = \{2, 5, 5, 5, 6, 6, 8, 9, 9, 9\}$ sorted array of integers containing duplicates, apply binary search algorithm to Count occurrences of a number provided, if the number is not found in the array report that as well. (Give implementation)

For example :

Input: 5

Output:

Element 5 occurs 3 times

ANSWER:

CODE:

```
#include <iostream>
```



```

using namespace std;

int main()
{
    const int MAX = 10;
    bool found = false;
    int start = 0, end = MAX - 1, i = (start + end) / 2, number, occurrences = 0;
    int LA[MAX] = { 2, 5, 5, 5, 6, 6, 8, 9, 9, 9 };
    cout << "Enter the number to search: ";
    cin >> number;
    while (true)
    {
        if (LA[i] == number)
        {
            occurrences++;
            found = true;
            int j = 1;
            if (LA[i - 1] == number)
            {
                while (LA[i - j] == number)
                {
                    occurrences++;
                    j--;
                }
            }
            j = 1;
            if (LA[i + 1] == number)
            {
                while (LA[i + j] == number)

```

```

        {
            occurrences++;

            j++;
        }
    }
    break;
}

else if (start == end)
{
    found = false;
    break;
}

else if (number > LA[i])
{
    start = i + 1;
    i = (start + end) / 2;
    continue;
}

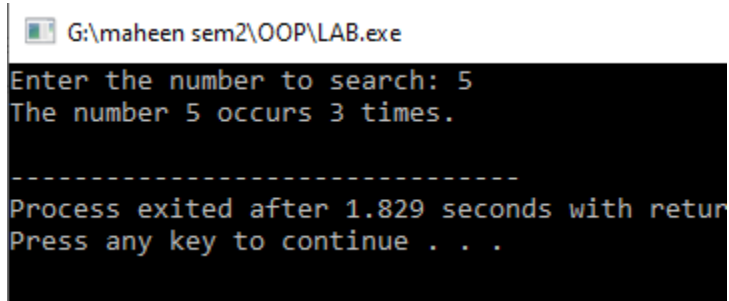
else
{
    end = i - 1;
    i = (start + end) / 2;
    continue;
}
}

if (found == true)
    cout << "The number " << number << " occurs " << occurrences << " times.\n";
else

```

```
    cout << "Not found.\n";  
}
```

OUTPUT:



```
G:\maheen sem2\OOP\LAB.exe  
Enter the number to search: 5  
The number 5 occurs 3 times.  
-----  
Process exited after 1.829 seconds with return code 0  
Press any key to continue . . .
```

Q3: Compare linear & binary Search algorithms on the basis of time complexity, which one is better in your opinion? Discuss.

ANSWER:

Input data needs to be sorted in Binary Search and not in Linear Search. Linear search does the sequential access whereas Binary search access data randomly. Time complexity of linear search - $O(n)$, Binary search has time complexity $O(\log n)$. Linear search performs equality comparisons and Binary search performs ordering comparisons. In case of a sorted array binary search would require no sorting and it would get the job done much faster than linear search and so binary search would be better in this case.

LAB NO. 04
Sorting Algorithms
i.Bubble Sort Algorithm ii. Quick Sort Algorithm
(Done by Mesum Sultan, SE-096)

Q.1:

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

int main()
{
    const int MAX = 6;
    int A[MAX] = { 6, 5, 4, 3, 2, 1 }, i = 0, ptr;
    while (i < MAX - 1)
    {
        ptr = 0;
        while (ptr < MAX - i - 1)
        {
            if (A[ptr] > A[ptr + 1])
            {
                A[ptr] = A[ptr] + A[ptr + 1];
                A[ptr + 1] = A[ptr] - A[ptr + 1];
                A[ptr] = A[ptr] - A[ptr + 1];
            }
            ptr++;
        }
        i++;
    }
    i = 0;
    while (i < MAX)
    {
        cout << A[i] << " ";
        i++;
    }
}
```

Q.2:

```
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

void bubblesort(int[], int, int);
int main()
```

```

{
    const int MAX = 13;
    int A[MAX] = { 6, 5, 4, 3, 2, 1, 12, 98, 4, 6, 13, 5, 0 }, i = 0;
    bubblesort(A, i, MAX);
    while (i < MAX)
    {
        cout << A[i] << " ";
        i++;
    }
}

```

Q.3:

```

#include <stdio.h>
int partition(int a[], int beg, int end);
void quickSort(int a[], int beg, int end);
void main()
{
    int i;
    int arr[10] = { 9,7,5,11,12,2,14,3,10,6 };
    quickSort(arr, 0, 9);
    printf("\n The sorted array is: \n");
    for (i = 0; i < 10; i++)
        printf(" %d\t", arr[i]);
}
int partition(int a[], int beg, int end)
{
    int left, right, temp, loc, flag;
    loc = left = beg;
    right = end;
    flag = 0;
    while (flag != 1)
    {
        while ((a[loc] <= a[right]) && (loc != right))
            right--;
        if (loc == right)
            flag = 1;
        else if (a[loc] > a[right])
        {
            temp = a[loc];
            a[loc] = a[right];
            a[right] = temp;
            loc = right;
        }
        if (flag != 1)
        {

```

```

    while ((a[loc] >= a[left]) && (loc != left))
        left++;
    if (loc == left)
        flag = 1;
    else if (a[loc] < a[left])
    {
        temp = a[loc];
        a[loc] = a[left];
        a[left] = temp;
        loc = left;
    }
}
}
return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if (beg < end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc - 1);
        quickSort(a, loc + 1, end);
    }
}

```

OUTPUT:

The sorted array is:

2 3 5 6 7 9 10 11 12 14 |

Q.4:*Bubble Sort:*

The simplest sorting algorithm. It involves the sorting the list in a repetitive fashion. It compares two adjacent elements in the list, and swaps them if they are not in the designated order. It continues until there are no swaps needed. This is the signal for the list that is sorted. It is also called as comparison sort as it uses comparisons.

Advantages:

The primary advantage of the bubble sort is that it is popular and easy to implement. In the bubble sort, elements are swapped in place without using additional temporary storage. The space requirement is at a minimum

Disadvantages:

The main disadvantage of the bubble sort is the fact that it does not deal well with a list containing a huge number of items.

The bubble sort requires n^2 processing steps for every n number of elements to be sorted.

The bubble sort is mostly suitable for academic teaching but not for real-life applications.

Quick Sort:

The best sorting algorithm which implements the 'divide and conquer' concept. It first divides the list into two parts by picking an element a 'pivot'. It then arranges the elements those are smaller than pivot into one sub list and the elements those are greater than pivot into one sub list by keeping the pivot in its original place.

Advantages:

The quick sort is regarded as the best sorting algorithm.

It is able to deal well with a huge list of items.

Because it sorts in place, no additional storage is required as well

Disadvantages:

The slight disadvantage of quick sort is that its worst-case performance is similar to average performances of the bubble, insertion or selections sorts.

If the list is already sorted than bubble sort is much more efficient than quick sort

If the sorting element is integers than radix sort is more efficient than quick sort.

LAB NO.5

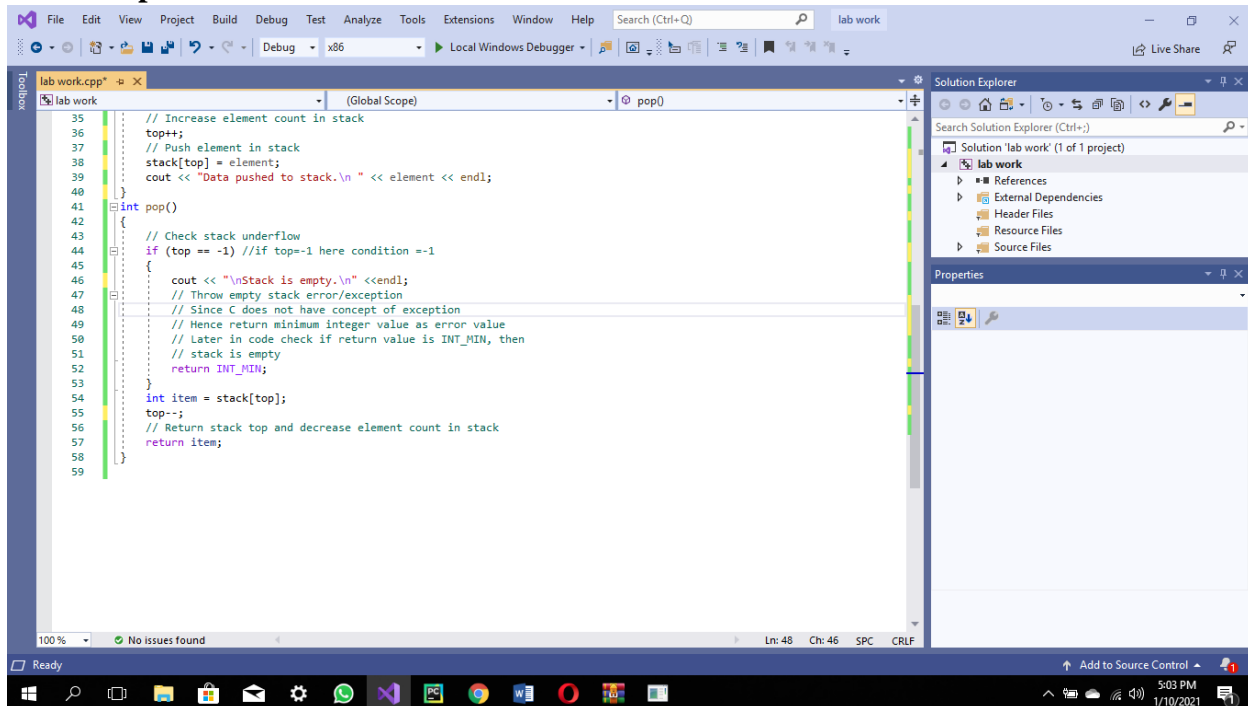
Stack data structure & Operations

i. Push ii. Pop

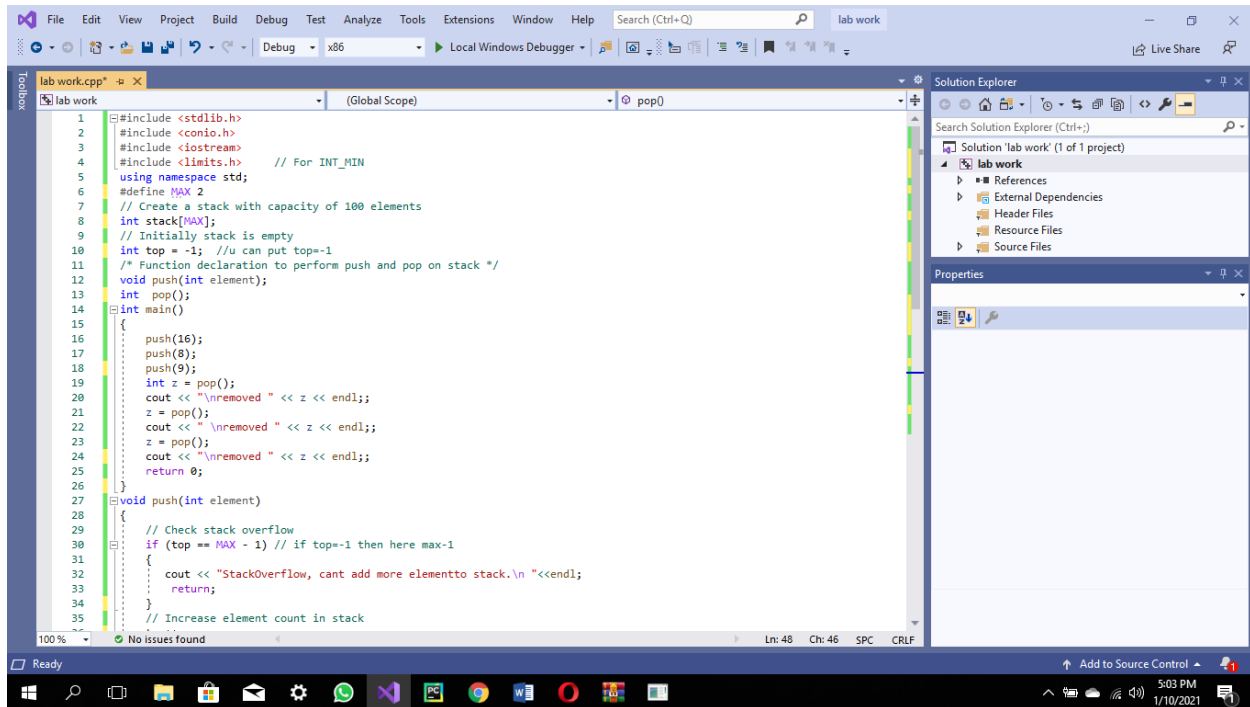
(Done by Mesum Sultan, SE-096)

Q.1:

Code Snap:



```
35 // Increase element count in stack
36 top++;
37 // Push element in stack
38 stack[top] = element;
39 cout << "Data pushed to stack.\n " << element << endl;
40
41 int pop()
42 {
43     // Check stack underflow
44     if (top == -1) //if top=-1 here condition ==-1
45     {
46         cout << "\nStack is empty.\n" << endl;
47         // Throw empty stack error/exception
48         // Since C does not have concept of exception
49         // Hence return minimum integer value as error value
50         // Later in code check if return value is INT_MIN, then
51         // stack is empty
52         return INT_MIN;
53     }
54     int item = stack[top];
55     top--;
56     // Return stack top and decrease element count in stack
57     return item;
58 }
59
```

OUTPUT SNAP:

```

Data pushed to stack.
16
Data pushed to stack.
8
StackOverflow, cant add more elementto stack.

removed 8
removed 16
Stack is empty.

removed -2147483648

C:\Users\AK\source\repos\lab work\Debug\lab work.exe (process 8648) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Q.2:

Time Complexity of an algorithm is the representation of the amount of time required by the algorithm to execute to completion. Time requirements can be denoted or defined as a

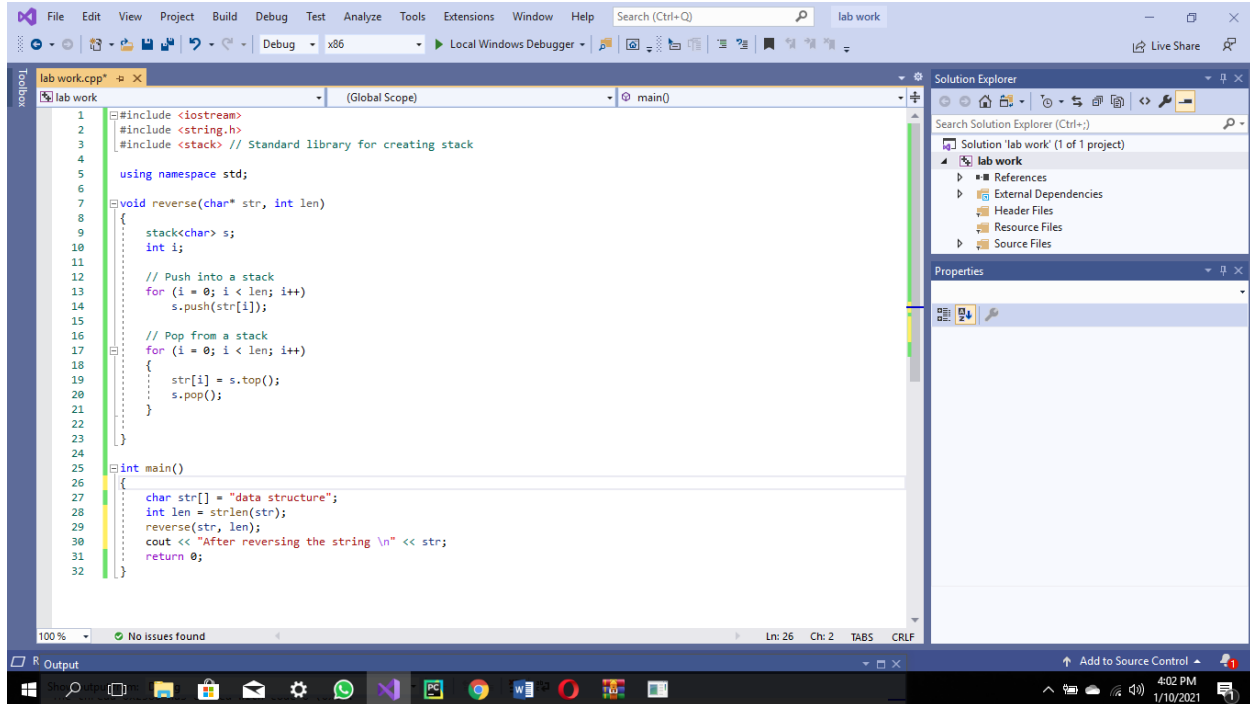
numerical function $t(N)$, where $t(N)$ can be measured as the number of steps, provided each step takes constant time.

Q.3:

Algorithm:

- 1- START
- 2- Create an empty stack
- 3- Push each character of the string into the stack one by one
- 4- Now, pop each character from stack and add to string
- 5- END

CODE SNAP:



OUTPUT SNAP:

```

After reversing the string
erutcurts atad
C:\Users\AK\source\repos\lab work\Debug\lab work.exe (process 9190) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Q.4:

Algorithm:

- 1- START
- 2- Initialize two stacks ST1 and ST2.
- 3- Initialize their respective end elements TOP1 = -1 (beginning of array) and TOP2 = n (end of array)
- 4- Push array element into ST1 and increase TOP1 by 1 each time. Push array element from opposite end into ST2 and decrease TOP2 by 1 each time.
- 5- Overflow occurs when both ST1 and ST2 collide with each other in the middle of the array, this signals the end of the program.

END

LAB NO 6

Expression Evaluation through Stack Data Structure

i. Infix ii. Postfix iii. Prefix

(Done by Tooba Izat, SE-100)

QUESTION NO 1:

i. P: 5,6,2,+,*, 12,4,/,-

SYMBOL SCANNED	STACK
(1) 5	5
(2) 6	5,6
(3) 2	5,6,2
(4) +	5,8
(5) *	40
(6) 12	40,12
(7) 4	40,12,4
(8) /	40,3
(9) -	37
(10))	

ii. 2,3,^,5,2,2,^,*,12,6,/,-,+

SYMBOL SCANNED	STACK
(1) 2	2
(2) 3	2,3
(3) ^	8
(4) 5	8,5
(5) 2	8,5,2
(6) 2	8,5,2,2
(7) ^	8,5,4
(8) *	8,20
(9) 12	8,20,12
(10) 6	8,20,12,6
(11) /	8,20,2
(12) -	8,18
(13) +	26
(14))	

QUESTION NO 2:

Implementation:

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

```
using namespace std;
```

```
int precedence(char a)
{
    if(a == '^')
        return 3;
    else if(a == '*' || a == '/')
        return 2;
    else if(a == '+' || a == '-')
        return 1;
    else
        return -1;
}
```

```
void infixToPostfix(string s)
{
    std::stack<char> st;
    st.push('N');
    int l = s.length();
    string ns;
    for(int i = 0; i < l; i++)
    {
```

```

if((s[i] >= 'a' && s[i] <= 'z') ||
   (s[i] >= 'A' && s[i] <= 'Z'))
    ns+=s[i];

```

```

else if(s[i] == '(')

```

```

    st.push('(');

```

```

else if(s[i] == ')')

```

```

{
    while(st.top() != 'N' && st.top() != '(')
    {
        char c = st.top();
        st.pop();
        ns += c;
    }
    if(st.top() == '(')
    {
        char c = st.top();
        st.pop();
    }
}

```

```

else{

```

```
while(st.top() != 'N' && precedence(s[i]) <=
      precedence(st.top()))
{
    char c = st.top();
    st.pop();
    ns += c;
}
st.push(s[i]);
}

}

while(st.top() != 'N')
{
    char c = st.top();
    st.pop();
    ns += c;
}

cout << ns << endl;

}

int main()
{
```

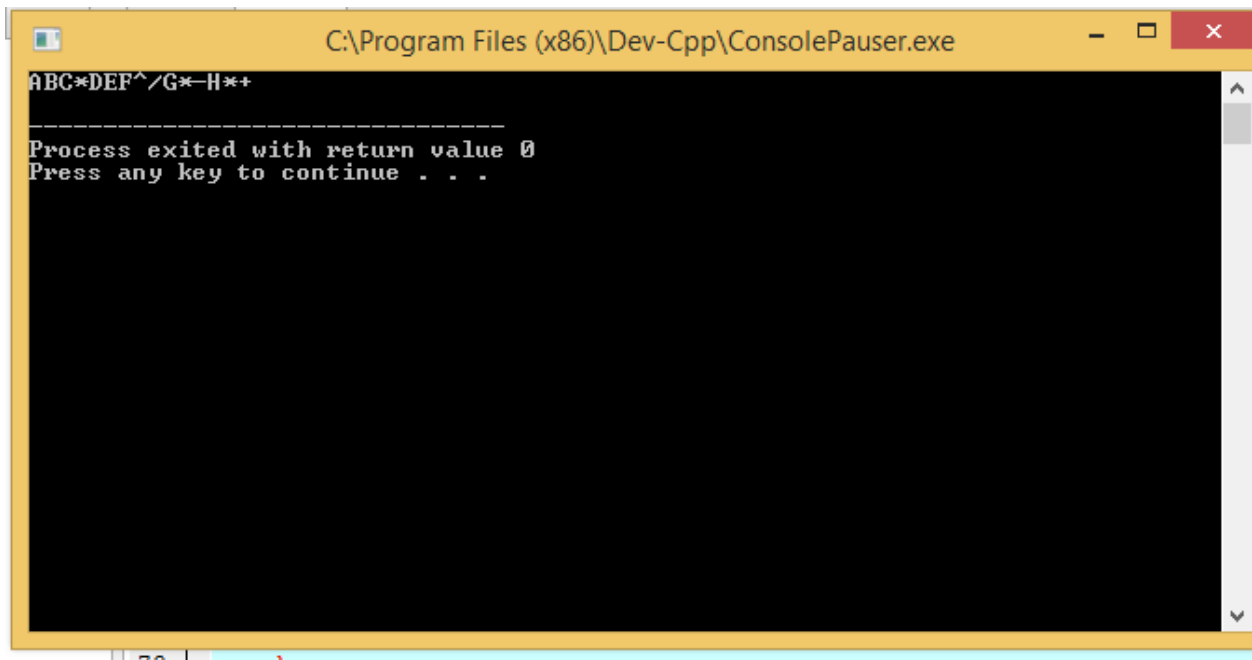
```

string exp = "(A+(B*C-(D/E^F)*G)*H) ";
infixToPostfix(exp);

return 0;
}

```

OUTPUT:



QUESTION NO 3:

Algorithm:

This algorithm finds the VALUE of an arithmetic expression P written in prefix notation.

1. START
2. Scan P from right to left to and repeats step 3 and 4 for each element until the index is greater and equal to zero.
3. If an operand is encountered put in on stack
4. If an operator op is encountered then:
 - a. Remove the top two element of stack where x is the top element and y is the next top element.

b. Evaluate $x \text{ op } y$

c. Place the result back on stack

[End of If structure]

[End of step 2 loop]

5. Set VALUE equal to the top element on STACK.

6. Exit

Implementation:

Input:

```
#include <iostream>
```

```
#include <stack>
```

```
using namespace std;
```

```
//function to evaluate given prefix expression
```

```
int prefix(string exp)
```

```
{
```

```
    //create an empty stack
```

```
    stack<int> stack;
```

```
    //traverse the given expression
```

```
    for (int i = exp.size() - 1; i >= 0; i--)
```

```
    {
```

```
        //if current char is an operand, push it to the stack
```

```
        if (exp[i] >= '0' && exp[i] <= '9') {
```

```
            stack.push(exp[i] - '0');
```

```
        }
```

```
        //if current char is an operator
```

```
    else
```

```
    {
```

```
        //pop top two elements from the stack
```

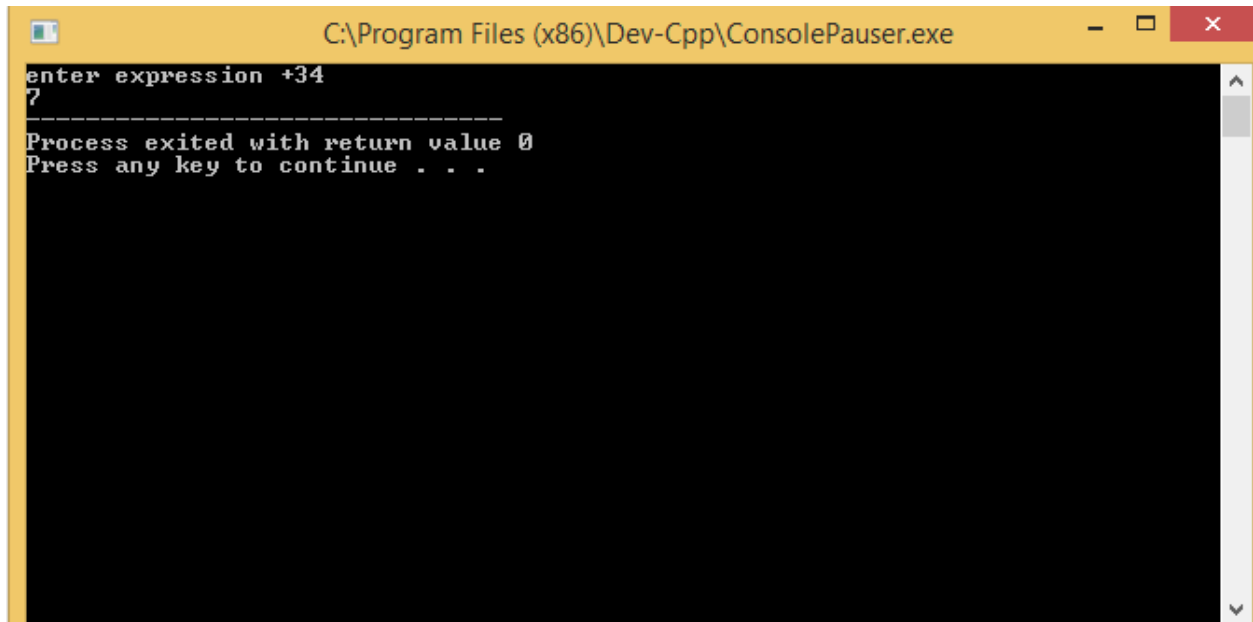
```

int x = stack.top();
stack.pop();
int y = stack.top();
stack.pop();
    //evaluate the expression x op y, and push the result back to the stack
if (exp[i] == '+')
stack.push(x + y);
else if (exp[i] == '-')
stack.push(x - y);
else if (exp[i] == '*')
stack.push(x * y);
else if (exp[i] == '/')
stack.push(x / y);
    }
}

//at this point, the stack is left with only one element i.e. expression result.
return stack.top();
}
int main()
{
string exp;
cout<<"enter expression ";cin>>exp;
int VALUE=prefix(exp);
cout<< VALUE;
return 0;
}

```

Output:



```
enter expression +34
?
-----
Process exited with return value 0
Press any key to continue . . .
```

LAB NO 7

Queue data structure & Operations

i. Enqueue ii. Dequeue

(Done by Maheen Bukhtiar, SE-069)

Q1. Write Algorithms for Enqueue and Dequeue operations in a circular queue.

ENQUEUE:

START

QINSERT(Queue, N, FRONT, REAR, ITEM)

STEP 1: [Queue already filled?]

If $FRONT = 1$ and $REAR = N$, or if $FRONT = REAR + 1$, then:

Write: OVERFLOW, and Return.

STEP 2: [Find new value of REAR.]

If $FRONT := NULL$, then: [Queue initially empty.]

Set $FRONT := 1$ and $REAR := 1$

Else if $REAR = N$, then:

Set $REAR := 1$

Else:

Set $REAR := REAR + 1$

[End of If structure]

STEP 3: Set $QUEUE[REAR] := ITEM$. [This inserts new element]

STEP 4: Return

STOP

DEQUEUE:

START

QDELETE(Queue, N, FRONT, REAR, ITEM)

STEP 1: [Queue already filled?]

If FRONT := NULL, then: Write: UNDERFLOW, and Return.

STEP 2: Set ITEM := QUEUE[FRONT].

STEP 3: [Find new value of FRONT]

If FRONT = REAR, then: [Queue has only one element to start.]

Set FRONT := NULL and REAR := NULL.

Else if FRONT = N, then:

Set FRONT := 1.

Else:

Set FRONT := FRONT + 1.

[End of If structure]

STEP 4: Return

STOP

Q2. Give implementation of queue data structure using two stacks (let S1 & S2 be the two stacks

For push and pop operations respectively).

IMPLEMENTATION:

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

```
using namespace std;
```

```
struct Queue {
```

```
    stack<int> s1, s2;
```

```
    void enQueue(int x)
```

```
    { // Move all elements from s1 to s2
```

```
while (!s1.empty()) {
```

```
    s2.push(s1.top());
```

```
    s1.pop();
```

```
}
```

```
// Push item into s1
```

```
s1.push(x);
```

```
// Push everything back to s1
```

```
while (!s2.empty()) {
```

```
    s1.push(s2.top());
```

```
    s2.pop();
```

```
}
```

```
}
```

```
// Dequeue an item from the queue
```

```
int deQueue()
```

```
{
```

```
    // if first stack is empty
```

```
    if (s1.empty()) {
```

```
        cout << "Q is Empty";
```

```
        exit(0);
```

```
    }
```

```
    // Return top of s1
```

```
    int x = s1.top();
```

```
s1.pop();

return x;

}

};
```

```
// Driver code
```

```
int main()
{

    Queue q;

    q.enqueue(1);

    q.enqueue(2);

    q.enqueue(3);


    cout << q.dequeue() << '\n';
```



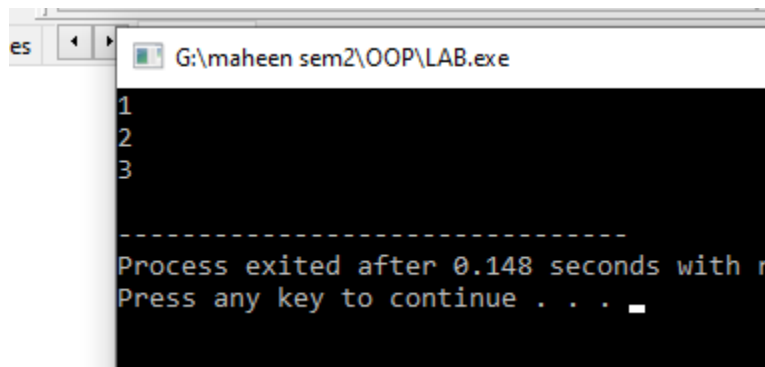
```
cout << q.dequeue() << '\n';
```

```
cout << q.dequeue() << '\n';
```

```
return 0;
```

```
}
```

OUTPUT:



LAB NO 8

Recursive Algorithms (Recursion)

Tower of Hanoi Problem

(Done by Ali Zia Khan, SE-052)

EXERCISE:

1. Calculate the Time complexity of Fibonacci and factorial recursive algorithms, Also give the upper bound of both algorithms.

TIME COMPLEXITY OF RECURSIVE FACTORIAL ALGORITHM:

$$T(n) = T(n-1) + 3 \quad \text{---eq(1)} \quad (T(0) = 1 \text{ if } n=0)$$

Similarly:

$$T(n-1) = 3 + T(n-2) \quad \text{---eq(2)}$$

$$T(n-2) = 3 + T(n-3) \quad \text{---eq(3)}$$

Now putting eq 2 in eq 1:

We get:

$$T(n) = 6 + T(n-2) \quad \text{---eq(4)}$$

Now putting eq 3 in eq 4

We get:

$$T(n) = 9 + T(n-3) \quad \text{---eq 5}$$

So,

$$T(n) = T(n-k) + 3k$$

Since $n-k=0$ or $n=k$

$$T(n) = T(n-n) + 3n$$

$$T(n) = T(0) + 3n \quad (T(0) = 1)$$

$$T(n) = 1 + 3n$$

Therefore Upper bound of algorithm i.e Big O Notation will be

$$O(n)$$

TIME COMPLEXITY OF RECURSIVE FIBONACCI ALGORITHM:

If $n \leq 1$:

$$T(0)=1$$

If $n > 1$:

$$T(n)=T(n-1)+T(n-2) +4$$

Although we know that $T(n-1)$ takes more time but here we will assume that $T(n-1)$ and $T(n-2)$ will take approximately same time. [$T(n-1) \sim T(n-2)$]

So equation will be like:

$$T(n)=T(n-1)+T(n-1)+4$$

$$T(n)=2T(n-1)+c \text{ <-eq 1}$$

NOW BY SUBSTITUTING:

$$T(n-1)=2T(n-2)+C \text{ <-eq 2}$$

$$T(n-2)=2T(n-3)+c \text{ <-eq 3}$$

$$T(n-3)=2T(n-4)+c \text{ <-eq 4}$$

Now putting eq 2 in eq 1:

We get:

$$T(n)=4T(n-2)+3c \text{ <-eq 5}$$

Now putting eq 3 in eq 5

We get:

$$T(n)=8T(n-3)+7c \text{ <- eq 6}$$

Now putting eq 4 in eq 6

We get:

We get:

$$T(n)=16T(n-4)+15c$$

So,

$$T(n)=2^k T(n-k)+(2^k-1) c$$

Now

For $n-k=0$ or $n=k$

$$T(n)=2^n T(0)+(2^n-1).c \quad [T(0)=1]$$

$$T(n)=2^n \cdot 1 +(2^n-1).c$$

$$T(n)=2^n(1+c)-c$$

SO THE UPPER BOUND i.e BIG O NOTATION OF ALGORITHM WILL BE:

$O(2^n)$

2. Write iterative factorial and Fibonacci algorithms ,also analyze the time complexity.

Iterative Fibonacci series algorithm:

N=total no of elements

Start

1. Declaring variable a,b,e,n,i
 2. a=1 b=1,i=3
 3. Take n as input
 4. Print a and b
 5. Repeat steps until $i \leq n$;
 - a) $C=a+b$
 - b) Print c
 - c) $a=b, b=c$
 - d) $i=i+1$
- Stop

Time complexity:

Statement	Operation	Iteration	Subtotal
2	3	1	3
3	1	1	1
4	1	1	1
5	1	n-1	n-1
5a	2	n-2	$2(n-2)$
5b	1	n-2	$(n-2)$
5c	2	n-2	$2(n-2)$
5d	2	n-2	$2(n-2)$
			$4+n+7(n-2)$

Time Complexity will be linear that is:

$$T(n)=8n-10$$

Where the most dominant factor is term with n and keeping away the constant, the big O notation or what we can say upper bound is $O(N)$

ITERATIVE FACTORIAL ALGORITHM:

N:NUMBER TO FIND FACTORIAL OF

START.

FACTORIAL(FACT , N)

1. IF N=0,

- a. SET FACT=1
- b. RETURN
- 2.SET FACT=1

2. REPEAT for K=1 TO N

- a) SET FACT=K*FACT
- 3.RETURN

STOP

Time complexity:

Statement	Operation	Iteration	Subtotal
1	1	1	1
1a	1	1	1
1b	1	1	1
2	1	n+1	n+1
2a	2	n	2n
3	1	1	1
			3n+5

Time Complexity will be linear that is:

$$T(n)=3n+5$$

Where the most dominant factor is term with n and keeping away the constant, the big O notation or what we can say upper bound is $O(N)$

3. Write recursive algorithm for Tower of Hanoi Puzzle also calculate its upper bound (also give implementation)

RECURSIVE ALGORITHM FOR TOWER OF HANOI PUZZLE:

TOWER OF HANOI ALGORITHM:

N=no of disks

Tower(n,beg,aux,end)

1.If n=1,then:

Print:beg->end

Return

2.[move n-1 disks from beg peg to peg aux]

Call tower(n-1,beg,end,aux)

3.Print: beg->end
 4.[move n-1 disks from peg aux to peg end]
 Call tower(n-1,aux,beg,end)
 6.Return

Time Complexity:

From Algorithm:

For $N > 1$

$$T(n) = 2T(n-1) + 1 \leq 1 \quad \text{Also } T(0) = 1 \text{ (FOR } N \leq 1)$$

Now for $N-1$:

$$T(n-1) = 2T(n-2) + 1 \leq 2$$

$$T(n-2) = 2T(n-3) + 1 \leq 3$$

Put value of $T(n-2)$ in equation-2 with help of equation-3

$$T(n-1) = 2(2T(n-3) + 1) + 1$$

PUT VALUE OF $T(N-1)$ IN EQUATION-4

$$T(N) = 2(2(2T(n-3) + 1) + 1) + 1$$

$$T(n) = 2^3 T(n-3) + 2^2 + 2^1 + 1$$

Hence it can be asserted:

$$T(n) = 2^k T(n-k) + 2^{(k-1)} + 2^{(k-2)} + \dots + 2^2 + 2^1 + 1$$

Since

$$T(0) = 1$$

$$n-k=0 \text{ OR } n=k$$

put $k=n$

$$T(n) = 2^n T(0) + 2^{(n-1)} + 2^{(n-2)} + \dots + 2^2 + 2^1 + 1$$

Since $T(0) = 1$

So

$$T(n) = 2^n(1) + 2^{(n-1)} + 2^{(n-2)} + \dots + 2^2 + 2^1 + 1$$

It is GP Series, sum is $2^{(n+1)} - 1$

$$T(n) = O(2^{(n+1)} - 1)$$

UPPER BOUND:

$$O(2^n)$$

```

2  #include<iostream>
3  using namespace std;
4
5
6  void towerofhanoi(int n, string beg,string aux,string end){
7      if(n<=1){
8          cout<<"Move disk "<<n<<" from " <<beg<<" to "<<end<<endl;
9          return;
10     }
11
12     towerofhanoi(n-1,beg,end,aux);
13
14     cout<<"Move disk "<<n<<" from " <<beg<<" to "<<end<<endl;
15     towerofhanoi(n-1,aux,beg,end);
16 }
17
18 int main()
19 {
20     int n=3;
21     string beg="A",aux="B",end="C";
22     towerofhanoi(n,beg,aux,end);
23 }

```

Output:

```

C:\Users\ALI ZIA\Desktop\new folder\dsaforrunning2.exe
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C

-----
Process exited after 0.2515 seconds with return value 0
Press any key to continue . . .

```

LAB NO. 9

Tree data structure & Operations

i. Insertion ii. Deletion

(Done by Mesum Sultan, SE-096)

Q.1:

Insertion in Binary Search Tree:

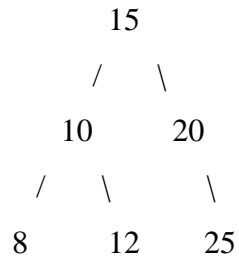
```

struct BSTnode
{
    int data;
    struct BSTnode *left;
    struct BSTnode *right;
};
BSTnode *rootptr;
struct BSTnode *GetNewNode(int data)
{
    struct BSTnode *temp = (struct BSTnode *)malloc(sizeof(struct BSTnode));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}
struct node *insert(struct BSTnode *root, int data)
{
    if (root == NULL)
        return GetNewNode(data);
    else if (data < root->data)
        root->left = insert(root->left, data);
    else
        root->right = insert(root->right, data);
    return root;
}
int main()
{
    struct node *root = NULL;
    root = insert(root, 15);
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 25);
    root = insert(root, 8);
    root = insert(root, 12);
}

```


Output:

The resulting tree will be:



Q.2:

Deleting the leaf left and right nodes of 10 through the following function code:

```

struct BSTnode *deleteNode(struct BSTnode *root, int data)
{
    if (root == NULL)
    {
        return root;
    }
    else if (data < root->data)
    {
        root->left = deleteNode(root->left, data);
    }
    else if (data > root->key)
        root->right = deleteNode(root->right, data);
    else {
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        root->key = temp->data;
        root->right = deleteNode(root->right, temp->data);
        return root;
    }
}
  
```

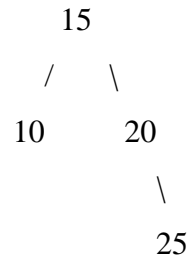
```

}
int main()
{
    root = deleteNode(root, 8);
    root = deleteNode(root, 12);
}

```

Output:

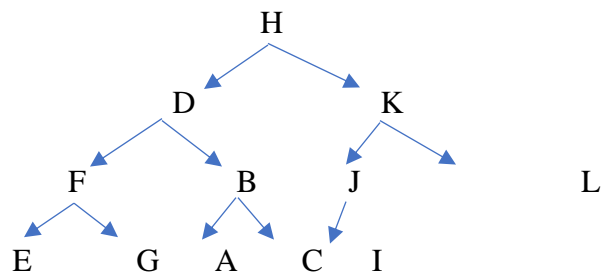
The deletion of the nodes will lead to a binary search tree that looks like:



Q.3:

- i- Root of tree: A
- ii- Height: 3
- iii- Degree: 2
- iv- Size: 9
- v- Leaf nodes: 5 (D, E, F, G, I)

Q.4:



Q.5:

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer
2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). [Self-balancing search trees](#) like [AVL](#) and [Red-Black trees](#) guarantee an upper bound of $O(\log n)$ for search.
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). [Self-balancing search trees](#) like [AVL](#) and [Red-Black trees](#) guarantee an upper bound of $O(\log n)$ for insertion/deletion.
4. Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

LAB NO 10

Tree Traversal Algorithms

i. Inorder ii. Preorder iii. Postorder

(Done by Ali Zia Khan, SE-052)

EXERCISE:

1. Traverse the following binary trees using the pre, in, and post order traversal methods.

FOR PART 1:

1) **PREORDER:** (ROOT,LEFT,RIGHT)

+,*,3,5,-,2,/,8,4

2) **INORDER:** (LEFT,ROOT,RIGHT)

3,*,5,+,2,-,8,/,4

3) **POSTORDER :** (LEFT,RIGHT,ROOT)

3,5,*,2,8,4,/,-,+

FOR PART 2:

1) **PREORDER:** (ROOT,LEFT,RIGHT)

2,7,2,6,5,11,5,9,4

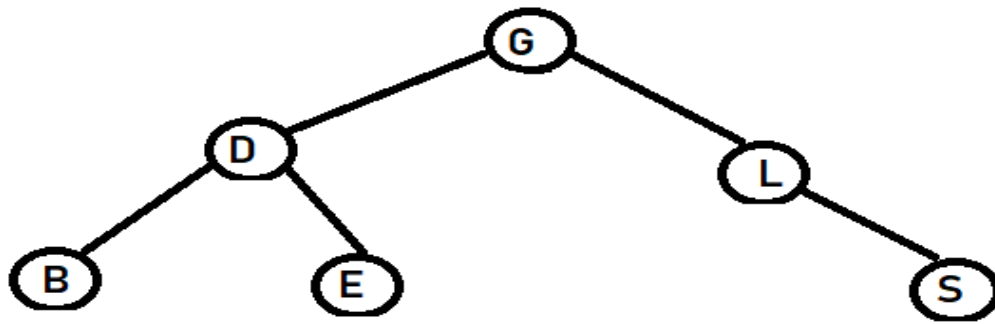
2) **INORDER:** (LEFT,ROOT,RIGHT)

2,7,5,6,11,2,5,4,9

3) **POSTORDER:** (LEFT,RIGHT,ROOT)

2,5,11,6,7,4,9,5,2

2. Implement all three tree traversal algorithms , also determine the worst time complexity,



```

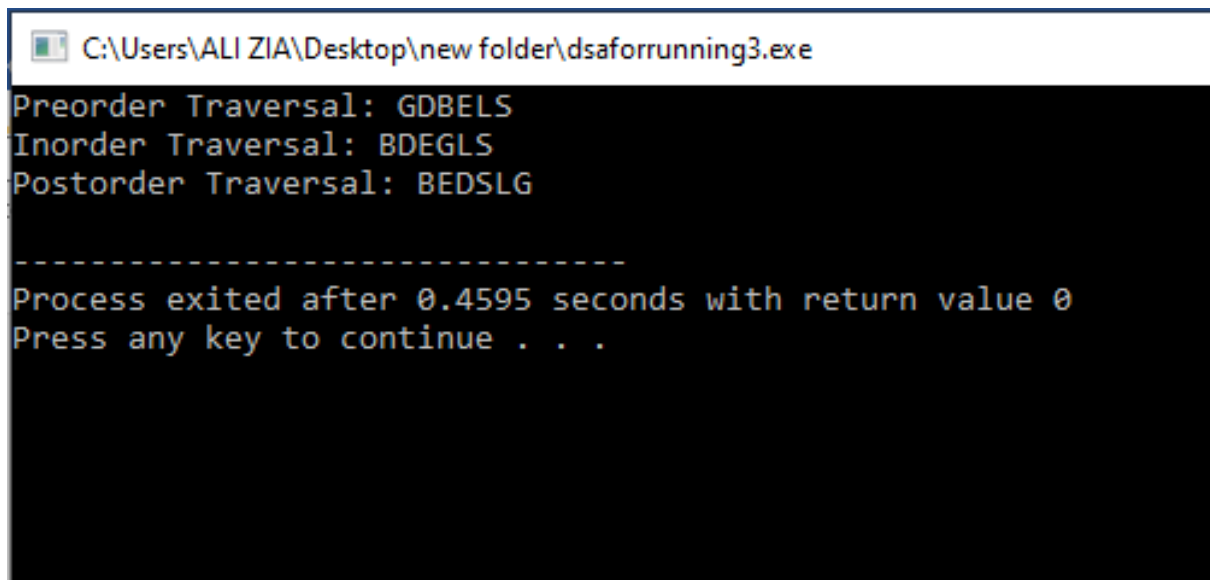
1
2 #include<iostream>
3 #include<conio.h>
4 using namespace std;
5
6 struct Node {
7     char data;
8     struct Node *left;
9     struct Node *right;
10 };
11
12 //Function for Preorder traversal
13 void for_preorder(struct Node *root) {
14     if(root == NULL)
15         return;
16     cout<<root->data; // Print data
17     for_preorder(root->left); // Visit left subtree
18     for_preorder(root->right); // Visit right subtree
19 }
20
21 //Function to visit nodes in Inorder
22 void for_inorder(Node *root) {
23     if(root == NULL)
24         return;
25     for_inorder(root->left); //Visit left subtree
26
27     cout<<root->data; //Print data
28
29     for_inorder(root->right); // Visit right subtree
30 }
31
32
33

```

```

33
34 //Function to visit nodes in Postorder
35 void for_postorder(Node *root) {
36     if(root == NULL)
37         return;
38     for_postorder(root->left);    // Visit Left subtree
39
40     for_postorder(root->right);    // Visit right subtree
41
42     cout<<root->data; // Print data
43 }
44
45 // Function to Insert Node in a Binary Search Tree
46 Node* Insert_Node(Node *root,char data) {
47     if(root == NULL) {
48
49         root = new Node();
50         root->data = data;
51         root->left = root->right = NULL;
52
53     }
54     else if(data <= root->data)
55
56         root->left = Insert_Node(root->left,data);
57     else
58         root->right = Insert_Node(root->right,data);
59     return root;
60 }
61
62 int main() {
63
64     Node* root = NULL;
65     root = Insert_Node(root,'G');
66     root = Insert_Node(root,'D');
67     root = Insert_Node(root,'L');
68     root = Insert_Node(root,'S');
69     root = Insert_Node(root,'B');
70     root = Insert_Node(root,'E');
71
72     cout<<"Preorder Traversal: ";
73     for_preorder(root);
74     cout<<"\n";
75
76     cout<<"Inorder Traversal: ";
77     for_inorder(root);
78     cout<<"\n";
79
80     cout<<"Postorder Traversal: ";
81     for_postorder(root);
82     cout<<"\n";
83
84     return 0;
85 }
86

```

OUTPUT:


```

C:\Users\ALI ZIA\Desktop\new folder\dsaforrunning3.exe
Preorder Traversal: GDBELS
Inorder Traversal: BDEGLS
Postorder Traversal: BEDSLG
-----
Process exited after 0.4595 seconds with return value 0
Press any key to continue . . .

```

The Recurrence

$$T(n) = 2 * T(n/2) + 1$$

where $T(n)$ is the number of operations executed in your traversal algorithm (in-order, pre-order, or post-order makes no difference).

Explanation of the Recurrence

There are two $T(n)$ because inorder, preorder, and postorder traversals all call themselves on the left and right child node. So, think of each recursive call as a $T(n)$. In other words, `**left T(n/2) + right T(n/2) = 2 T(n/2) **`.

The "1" comes from any other constant time operations within the function, like printing the node value, etc

It could be a 1 or any constant number

Analysis

$$T(n) = 2T(n/2) + c$$

$$T(n/2) = 2T(n/4) + c \Rightarrow T(n) = 4T(n/4) + 2c + c$$

Similarly, $T(n) = 8T(n/8) + 4c + 2c + c$

and as it goes on:

$T(n) = nT(1) + c(\text{sum of powers of 2 from 0 to } h(\text{height of tree}))$

so Complexity is $O(2^{(h+1)} - 1)$

but $h = \log(n)$

so, $O(2n - 1) = O(n)$

Therefore,

the worst time complexity or big o notation of the tree-traversal methods is $O(n)$.

3. Gives the implementation of Print all nodes of a perfect binary tree in Top-to-Down order

TOP-TO-DOWN:

```

1
2  #include <iostream>
3  #include <conio.h>
4  #include <queue>
5  using namespace std;
6
7
8  struct Node
9  {
10     int data;
11     Node *left;
12     Node *right;
13 };
14
15
16 Node *newNode(int data)
17 {
18     Node *node = new Node;
19     node->data = data;
20     node->right = node->left = NULL;
21     return node;
22 }
23
24
25 void SpecificLevelOrder(Node *root)
26 {
27     if (root == NULL)
28         return;
29
30     cout << root->data;
31
32

```



```

32
33
34     if (root->left != NULL)
35         cout << " " << root->left->data;
36         cout << " " << root->right->data;
37
38
39     if (root->left->left == NULL)
40         return;
41
42     // Using queue
43     queue <Node *> q_u;
44     q_u.push(root->left);
45     q_u.push(root->right);
46
47
48     Node *first = NULL, *second = NULL;
49
50     |
51     while (!q_u.empty())
52     {
53
54         first = q_u.front();
55         q_u.pop();
56         second = q_u.front();
57         q_u.pop();
58
59         cout << " " << first->left->data;
60         cout << " " << second->right->data;
61         cout << " " << first->right->data;
62         cout << " " << second->left->data;
63

```

```

64
65     if (first->left->left != NULL)
66     {
67         q_u.push(first->left);
68         q_u.push(second->right);
69         q_u.push(first->right);
70         q_u.push(second->left);
71     }
72 }
73
74
75
76 int main()
77 {
78     Node *root = newNode(1);
79
80     root->left = newNode(2);
81     root->right = newNode(3);
82
83     root->left->left = newNode(4);
84     root->left->right = newNode(5);
85     root->right->left = newNode(6);
86     root->right->right = newNode(7);
87


```

```

87
88     root->left->left->left = newNode(8);
89     root->left->left->right = newNode(9);
90     root->left->right->left = newNode(10);
91     root->left->right->right = newNode(11);
92     root->right->left->left = newNode(12);
93     root->right->left->right = newNode(13);
94     root->right->right->left = newNode(14);
95     root->right->right->right = newNode(15);
96
97
98
99     cout << "Specific Level Order traversal of binary tree is this\n";
100    SpecificLevelOrder(root);
101
102    return 0;
103 }
104

```

Output:

 C:\Users\ALI ZIA\Desktop\new folder\dsaforrunning4.exe

```

Specific Level Order traversal of binary tree is this
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12
-----

```

```

Process exited after 0.4178 seconds with return value 0
Press any key to continue . . .

```

LAB NO 11

Graph data structure & Operations

i. Adjacency List ii. Adjacency Matrix

(Done by M. Anas Bin Ateeq, SE-100)

EXERCISE:

1. Gives the algorithms for adjacency list and adjacency matrix methods. also determine their time complexity.

2. Assume an undirected graph having 4 vertices 0,1,2,3. the vertices are connected in following

manner 0-1, 0-2, 1-0, 1-2, 2-3, 3-2. Implement graph DS by using Adjacency list and Adjacency Matrix method.

2. Differentiate between graph and Tree data structure.

Question No. 01:

Give the algorithms for adjacency list and adjacency matrix methods and also determine their time complexity.

Adjacency List:

- Algorithm:

Step 1) Make class graph

Step 1.1) Declare and Initialize List

Step 1.2) Create function addedge(x,y) which creates an edge between vertex x and y

Step 1.2.1) List[x] -> y

Step 1.2.2) List[y] -> x

Step 2) Print List

- **Time Complexity:** $O(m)$ where m is the number of edges in the graph

Adjacency Matrix:

- **Algorithm:**

Step 1) Declare and Initialize the matrix

Step 2) Create function `add_edge(x,y)` where x signifies a row and y signifies a column.

Step 2.1) `matrix[x][y] = 1`

Step 2.2) `matrix[y][x] = 1`

Step 3) Add corresponding edges into the matrix using above function

Step 4) Print matrix

- **Time Complexity:** $O(n^2)$ where n is the number of vertices

Question No. 02:

Assume an undirected graph having 4 vertices 0,1,2,3. the vertices are connected in following manner $0 \square 1, 0 \square 2, 1 \square 0, 1 \square 2, 2 \square 3, 3 \square 2$. Implement graph DS by using Adjacency list and Adjacency Matrix method.

Implement graph DS by using Adjacency list

```
#include<iostream>
```

```
#include<list>
```

```
using namespace std;
```

```
class Graph
```

```
{
```

```
protected:
```

```
int V;
```

```

//array of list
list<int> *l;

public:

    Graph(int V)
    {
        this->V=V;

        l=new list<int>[V];
    }

    void AddEdge(int x, int y)
    {
        l[x].push_back(y);
        l[y].push_back(x);
    }

    void printlist()
    {
        for(int i=0;i<V;i++)
        {
            cout<<"Vertex"<<i<<"->";

            for(int nbr:l[i])
            {
                cout<<nbr<<" ";
            }

            cout<<endl;
        }
    }
}

```

```
};

int main()
{
    Graph g(4);

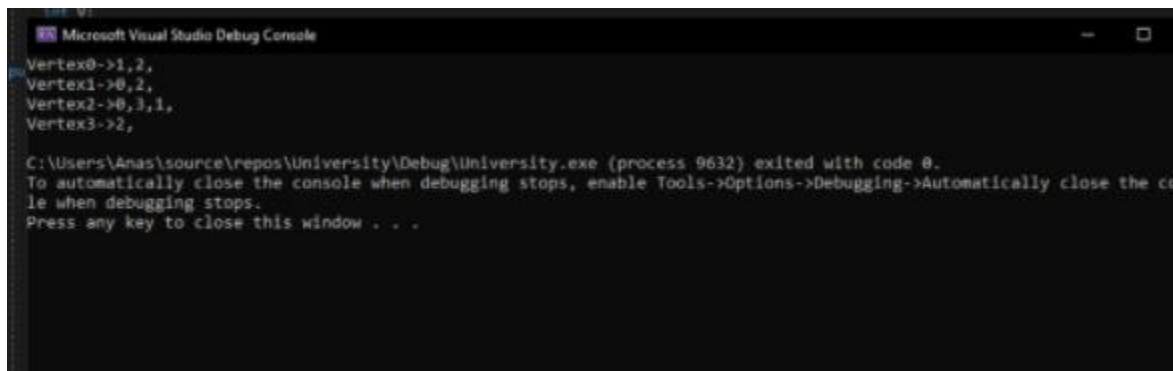
    g.AddEdge(0,1);

    g.AddEdge(0,2);

    g.AddEdge(2,3);

    g.AddEdge(1,2);

    g.printlist();
}
```



Implement graph DS by using Adjacency Matrix:

```
#include<iostream>

using namespace std;

int vertArr[20][20]; //the adjacency matrix initially 0

int count = 0;

void displayMatrix(int v) {

    int i, j;

    cout<<"adjacency matrix representation is : "<<endl;
```

```

for(i = 0; i < v; i++) {
    for(j = 0; j < v; j++) {
        cout << vertArr[i][j] << " ";
    }

    cout << endl;
}

}

void add_edge(int u, int v) {    //function to add edge into the matrix
    vertArr[u][v] = 1;
    vertArr[v][u] = 1;
}

main(int argc, char* argv[]) {
    int v = 4;    //there are 4 vertices in the graph

    add_edge(0,2);
    add_edge(0,3);
    add_edge(2,0);
    add_edge(1,1);

    displayMatrix(v);
}

```

```

adjacency matrix representation is :
0 0 0 1
0 0 1 0
0 1 0 0
1 0 0 0

-----
Process exited after 0.1358 seconds with return value 0
Press any key to continue . . .

```

Question No. 03:

Differentiate between graph and Tree data structure.

Graph	Tree
Graph is non linear	Tree is linear
It may be a collection of vertices/nodes and edges.	It may be a collection of hubs and edges.
A node can have multiple number of edges	Trees can also have any number of edges but in the case of binary tree a try can only have at max two edges
There is no root node	A root node is necessary
Cycles can be formed	There is no cycle
Applications: For finding a most brief way in organizing chart is used.	Applications: For amusement trees, choice trees, the tree are utilized.

LAB NO 12

Graph Traversal Algorithms i. DFS ii. BFS

(Done by M. Anas Bin Ateeq, SE-055)

1. Give Implementation of depth & breadth first search algorithms on graph (lab 11 Q2)

IMPLEMENTATION OF BREADTH FIRST SEARCH (BFS)

CODING:

```
#include <iostream>

#include <list>

using namespace std;

class Graph {

    int numVertices;

    list<int>* adjLists;

    bool* visited;


    public:

    Graph(int vertices);

    void addEdge(int src, int dest);

    void BFS(int startVertex);

};


// Create a graph with given vertices,
// and maintain an adjacency list

Graph::Graph(int vertices) {

    numVertices = vertices;

    adjLists = new list<int>[vertices];
```

```
}
```

```
// Add edges to the graph
```

```
void Graph::addEdge(int src, int dest) {
```

```
    adjLists[src].push_back(dest);
```

```
    adjLists[dest].push_back(src);
```

```
}
```

```
// BFS algorithm
```

```
void Graph::BFS(int startVertex) {
```

```
    visited = new bool[numVertices];
```

```
    for (int i = 0; i < numVertices; i++)
```

```
        visited[i] = false;
```

```
    list<int> queue;
```

```
    visited[startVertex] = true;
```

```
    queue.push_back(startVertex);
```

```
    list<int>::iterator i;
```

```
    cout<<"taking 2 as a root node"<<endl;
```

```
    cout<<endl;
```

```
    cout<<"THE BFS IMPLEMENTATION IS :"<<endl;
```

```
    cout<<endl;
```

```
    while (!queue.empty()) {
```

```
        int currVertex = queue.front();
```

```
        cout << "Visited " << currVertex << " ";
```

```

queue.pop_front();

for (i = adjLists[currVertex].begin(); i != adjLists[currVertex].end(); ++i) {

    int adjVertex = *i;

    if (!visited[adjVertex]) {

        visited[adjVertex] = true;

        queue.push_back(adjVertex);

    }

}

}

int main() {

    Graph g(6);

    g.addEdge(0, 5);

    g.addEdge(0, 2);

    g.addEdge(1, 3);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(3, 5);

    g.BFS(1);

    return 0;

}

```

```

E:\CPP FILES\LAB 12 Q 2.exe
taking 2 as a root node

THE BFS IMPLEMENTATION IS :

Visited 1 Visited 3 Visited 0 Visited 2 Visited 5 Visited 4
-----
Process exited after 0.151 seconds with return value 0
Press any key to continue . . .

```

IMPLEMENTATION DEPTH FIRST SEARCH(DFS)

```

#include <iostream>

#include <list>

using namespace std;

class Graph {

int numofvertices;

list<int> *Lists;

bool *visited;

public:

Graph(int V);

void addEdge(int source, int destination);

void dfs(int vertex);

};

Graph::Graph(int vertices) {

numofvertices = vertices;

```

```

Lists = new list<int>[vertices];
visited = new bool[vertices];
}

void Graph::addEdge(int src, int dest) {
Lists[src].push_front(dest);
}

void Graph::dfs(int vertex) {
visited[vertex] = true;
list<int>adjList = Lists[vertex];

cout<< vertex << " ";

list<int>::iterator i;
for (i = adjList.begin(); i != adjList.end(); ++i)
if (!visited[*i])
dfs(*i);
}

int main() {
    Graph graph(6);
graph.addEdge(0, 3);
graph.addEdge(0, 5);
graph.addEdge(1, 8);
graph.addEdge(1, 2);

```

```

graph.addEdge(2, 9);
graph.addEdge(3, 2);
graph.dfs(0);
return 0;
}

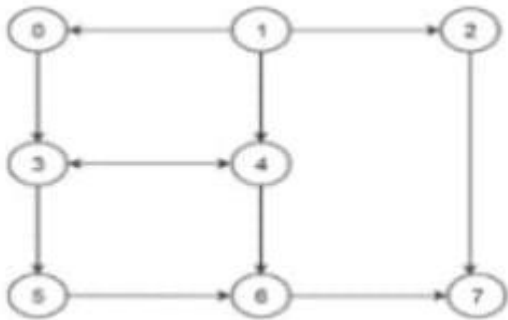
```

```

0 5 3 2
-----
Process exited after 0.1317 seconds with return value 0
Press any key to continue . . .

```

2. Use BFS (diagram below) to Find the path between given vertices in a directed graph



THE BFS PATH OF GIVEN DIAGRAM IS : **0,3,4,6,7.**

```
#include<iostream>
```

```
#include <list>
```

```
using namespace std;
```

```
// This class represents a directed graph using
```

```
// adjacency list representation
```

```
class Graph
```

```
{
```

```

    int V;

    // Pointer to an array containing adjacency
    // lists
    list<int> *adj;

public:

    Graph(int V);

    // function to add an edge to graph
    void addEdge(int v, int w);

    // prints BFS traversal from a given source s
    void BFS(int s);

};

Graph::Graph(int V)
{
    this->V = V;

    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

void Graph::BFS(int s)
{
    // Mark all the vertices as not visited
    bool *visited = new bool[V];

```

```

for (int i = 0; i < V; i++)
    visited[i] = false;

// Create a queue for BFS
list<int> queue;

// Mark the current node as visited and enqueue it
visited[s] = true;
queue.push_back(s);

// 'i' will be used to get all adjacent
// vertices of a vertex
list<int>::iterator i;
while (!queue.empty())
{
    // Dequeue a vertex from queue and print it
    s = queue.front();
    cout << s << " ";

    queue.pop_front();

    // Get all adjacent vertices of the dequeued
    // vertex s. If a adjacent has not been visited,
    // then mark it visited and enqueue it

    for (i = adj[s].begin(); i != adj[s].end(); ++i)

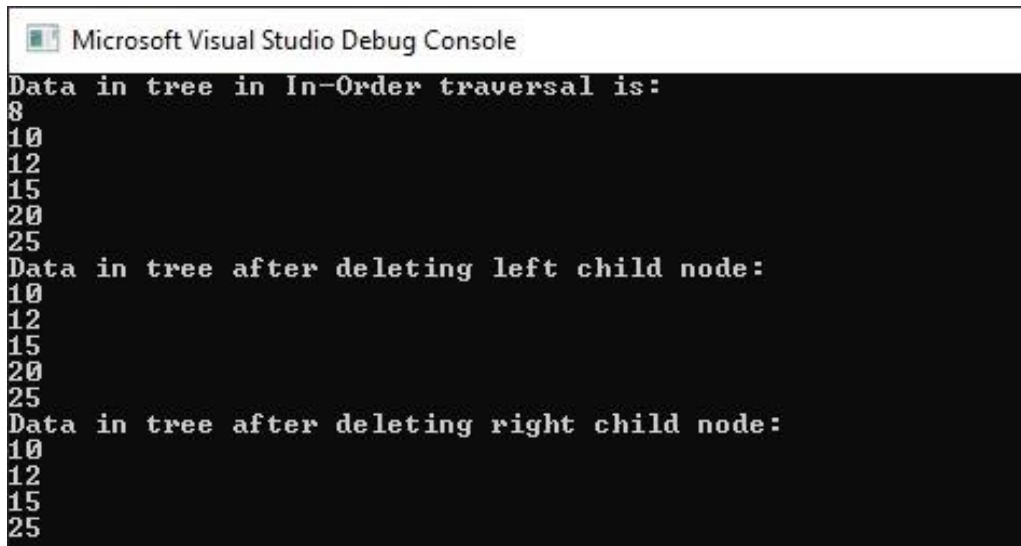
```



```

        {
            if (!visited[*i])
            {
                visited[*i] = true;
                queue.push_back(*i);
            }
        }
    }
}

```



```

Microsoft Visual Studio Debug Console
Data in tree in In-Order traversal is:
8
10
12
15
20
25
Data in tree after deleting left child node:
10
12
15
20
25
Data in tree after deleting right child node:
10
12
15
25

```

3. Write algorithm to find minimum no. of throws required to win snake & Ladder game by using BFS approach , also analyze its time complexity .(Give implementation as well).

ALGORITHM:

- Consider each as a vertex in directed graph.
- From cell 1 you can go to cells 2, 3, 4, 5, 6, 7 so vertex 1 will have directed edge towards vertex 2, vertex 3....vertex 7. Similarly consider this for rest of the cells.
- For snake- connect directed edge from head of snake vertex to tail of snake vertex. (See example image above- snake from 12 to 2. So directed edge from vertex 12 to vertex 2)
- For ladder- connect directed edge from bottom of ladder vertex to top of the ladder vertex.
- Now problem is reduced to Shorted path problem. So by Breadth-First Search (using queue) we can solve the problem.

IMPLEMENTATION

```
#include<iostream>
using namespace std;
int main() {
    int ldr[10] = { 9,12,10,25,30,41,45,88,89,94 };
    int loc=1;
    int rolls=0;
    cout << endl << "Rolls  " << rolls << " ---- " << "location " << loc << endl;
    while (true) {
        if (loc == 100) {
            break;
        }
        else if (loc >= 100) {
            loc += 1;
            break;
        }

        rolls += 1;
        int max = 0;
        for (int i = 1; i < 7; i++) {
            for (int j = 0; j < 9; j += 2) {
                if (loc + i == ldr[j]) {
                    if (ldr[j + 1] > max) {
                        max = ldr[j + 1];
                    }
                }
            }
        }
        if (max == 0) {
            loc += 6;
        }
        else {
            loc = max;
        }
        cout << endl << "Rolls  " << rolls << " ---- " << "location " << loc << endl;
    }
    cout << endl << "TOTAL ROLLS ---> "<< " "<< rolls << endl;

};
```

OUTPUT:

```
E:\CPP FILES\LAB 12 Q 2.exe

Rolls  0 ---- location 1
Rolls  1 ---- location 7
Rolls  2 ---- location 25
Rolls  3 ---- location 41
Rolls  4 ---- location 88
Rolls  5 ---- location 94
Rolls  6 ---- location 100

TOTAL ROLLS --->  6

-----
Process exited after 0.1376 seconds with return value 0
Press any key to continue . . .
```

LAB NO 13

Rat in a Maze Path finding problem

(Done by Tooba Izat, SE-100)

QUESTION NO 1:

I use recursion here to solve the problem. Recursion makes it possible to go through all the possible routes in the maze without making the code too bloated. Assuming that the rat can only move down or forward I use recursion to check that moving which way will get the rat to the final spot. Recursion makes it possible to move through all the possible routes until the correct route is found.

QUESTION NO 2:

The following algorithms is for the function which return true if it finds a route from start to finish. It takes six parameters. An int pointer representing the rat (rat), two 2D arrays representing the maze and the correct route through it respectively (maze[][] & sol[][]), the maximum number of elements(MAX), and the two indexes of the array(optr & iptr).

1. If rat is equal to maze[MAX - 1][MAX - 1]
 - 1.1. Set sol[MAX - 1][MAX - 1] equal to 1 and return true.
2. Else.
 - 2.1. If maze[optr][iptr + 1] is equal to 1 and (iptr + 1) is less than MAX.
 - 2.1.1. Set sol[optr][iptr + 1] equal to 1.
 - 2.1.2. Set rat equal to Address of maze[optr][iptr + 1].
 - 2.1.3. if (found(rat, maze, sol, MAX, optr, iptr + 1)).
 - 2.1.3.1. return true.
 - 2.1.3.2. End of if.
 - 2.1.4. End of if.
 - 2.2. If maze[optr + 1][iptr] is equal to 1 and (optr + 1) is less than MAX.
 - 2.2.1. Set sol[optr + 1][iptr] equal to 1.
 - 2.2.2. Set rat equal to Address of maze[optr + 1][iptr].
 - 2.2.3. if (found(rat, maze, sol, MAX, optr + 1, iptr)).
 - 2.2.3.1. return true.
 - 2.2.3.2. End of if.
 - 2.2.4. End of if.
 - 2.3. End of else.
3. Set sol[optr][iptr] equal to 0.
4. Return false.

Complexity Analysis:

- **Time Complexity:** $O(2^{(n^2)})$.
The recursion can run upper-bound $2^{(n^2)}$ times.
- **Space Complexity:** $O(n^2)$.
Output matrix is required so an extra space of size $n*n$ is needed.

QUESTION NO 3:

```

#include<iostream>
using std::cout;
using std::cin;
using std::endl;
bool found(int*, int[][4], int[][4], int, int, int);
int main()
{
    constint MAX = 4;
    int maze[MAX][MAX] = {
        {1, 0, 0, 0},
        {1, 1, 0, 1},
        {0, 1, 0, 0},
        {1, 1, 1, 1}
    };

    int sol[MAX][MAX] = {
        {1, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };

    int* rat = &maze[0][0];
    if (found(rat, maze, sol, MAX, 0, 0))
    {
        for (int i = 0; i < MAX; i++)
        {
            for (int j = 0; j < MAX; j++)
            {
                cout << sol[i][j] <<" ";
            }
            cout << endl;
        }
    }
}

bool found(int* rat, intmaze[][4], intsol[][4], intMAX, intoptr, intiptr)
{
    if (rat == &maze[MAX - 1][MAX - 1])
    {
        sol[MAX - 1][MAX - 1] = 1;
        return true;
    }
    else
    {
        if (maze[optr][iptr + 1] == 1 && (iptr + 1) < MAX)
        {

```

```
sol[optr][iptr + 1] = 1;
rat = &maze[optr][iptr + 1];
if (found(rat, maze, sol, MAX, optr, iptr + 1))
    return true;
    }
if (maze[optr + 1][iptr] == 1 && (optr + 1) < MAX)
    {
    sol[optr + 1][iptr] = 1;
    rat = &maze[optr + 1][iptr];
    if (found(rat, maze, sol, MAX, optr + 1, iptr))
        return true;
    }
    }
sol[optr][iptr] = 0;
return false;
}
```

LAB NO. 14

N-queen Problem

(Done by Maheen Bukhtiar, SE-069)

Q1. What approach do you used to solve problem and why?

ANSWER:

A queen can only be attacked if it lies on the same row, or same column, or the same diagonal of any other queen. To solve this problem, we will make use of the Backtracking algorithm.

Here, backtracking through recursion is the applied process. A queen is placed only it's not attacked by the opposite queen. to do this we must ensure that we place our queen to the upper left diagonal, the upper right diagonal and within the above columns. This must be done for every row and when a point comes where both the queens are attacking one another, we backtrack our solution to the point which should be changed and begin from there. Also, if there's no space left a queen then we'll backtrack until and redo all the steps by placing the various queen during a different position. This must be done until all the N queens are placed in a suitable position.

Q2. Give the algorithm of above problem, Also analyze the time & space complexity of your

ALGORITHM:

1. isValid(board, row, column)
 - 1.1. if there is a queen at left then
 - 1.1.1. Return false.
 - 1.2. If there is a queen at the left upper diagonal then
 - 1.2.1. Return false.
 - 1.3. If there is a queen at left lower diagonal then
 - 1.3.1. Return false.
2. solveNQueen(board, col)
 - 2.1. if all columns are filled, then
 - 2.1.1. Return true.
 - 2.2. For each row, do
 - 2.2.1. If isValid(board, I, column), then
 - 2.2.1.1. Set queen at place (I, col) in the board
 - 2.2.1.2. If solveNQueen(board, col + 1) is true, then
 - 2.2.1.2.1. Return true.
 - 2.2.1.3. Otherwise remove queen from place (I, col) from board.
 - 2.2.2. End of loop.
 - 2.3. Return false.
3. Time Complexity is $O(2^{(n^2)})$ as the recursion can run upper-bound $2^{(n^2)}$ times.
4. Space Complexity is $O(1)$ as no extra elements are required besides i and j.

Q3. Give implementation of 8 Queen Problem.

IMPLEMENTATION:

```
#include<iostream>
using namespace std;
#define N 8
void printBoard(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            cout << board[i][j] << " ";
        cout << endl;
    }
}
bool isValid(int board[N][N], int row, int col)
{
    for (int i = 0; i < col; i++)
        if (board[row][i])
            return false;
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
    for (int i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;
    return true;
}
bool solveNQueen(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if (isValid(board, i, col))
        {
            board[i][col] = 1;
            if (solveNQueen(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
}
```



```

    }
    return false;
}
bool checkSolution()
{
    int board[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            board[i][j] = 0;
    if (solveNQueen(board, 0) == false)
    {
        cout << "Solution does not exist";
        return false;
    }
    printBoard(board);
    return true;
}
int main()
{
    checkSolution();
}

```

```

3
3 G:\maheen sem2\OOP\LAB.exe
3
3 1 0 0 0 0 0 0
3 0 0 0 0 0 0 1
3 0 0 0 0 1 0 0
3 0 0 0 0 0 0 0
3 0 0 0 0 0 0 1
4 0 1 0 0 0 0 0
4 0 0 0 1 0 0 0
4 0 0 0 0 0 1 0
4 0 0 1 0 0 0 0
4
4 -----
4 Process exited after 0.1536 seconds with return value 0
4 Press any key to continue . . .
4
5
5

```