Lab 01

To learn the basic concepts of Data Structure & Algorithms

Data Definition : Data Definition defines a particular data with the following characteristics.

- **Atomic** − Definition should define a single concept.
- **Traceable** − Definition should be able to be mapped to some data element.
- **Accurate** − Definition should be unambiguous.
- **Clear and Concise** − Definition should be understandable.

Data Object : Data Object represents an object having a data**.**

Data Type: Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Basic Operations: The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Data Structure

Data structures are fundamental concepts of computer science which helps is writing efficient programs in any language. Python is a high-level, interpreted, interactive and object-oriented scripting language using which we can study the fundamentals of data structure in a simpler way as compared to other programming languages.

There are also some data structures specific to python which is listed as another category.

General Data Structures

The various data structures in computer science are divided broadly into two categories shown below. We will discuss about each of the below data structures in detail in subsequent chapters.

Liner Data Structures

These are the data structures which store the data elements in a sequential manner.

- **Array:** It is a sequential arrangement of data elements paired with the index of the data element.
- **Linked List:** Each data element contains a link to another element along with the data present in it.
- **Stack:** It is a data structure which follows only to specific order of operation. LIFO(last in First Out) or FILO(First in Last Out).
- **Queue:** It is similar to Stack but the order of operation is only FIFO(First In First Out).
- **Matrix:** It is two dimensional data structure in which the data element is referred by a pair of indices.

Non-Liner Data Structures

These are the data structures in which there is no sequential linking of data elements. Any pair or group of data elements can be linked to each other and can be accessed without a strict sequence.

- **Binary Tree:** It is a data structure where each data element can be connected to maximum two other data elements and it starts with a root node.
- **Heap:** It is a special case of Tree data structure where the data in the parent node is either strictly greater than/ equal to the child nodes or strictly less than it's child nodes.

- **Hash Table:** It is a data structure which is made of arrays associated with each other using a hash function. It retrieves values using keys rather than index from a data element.
- **Graph: .**It is an arrangement of vertices and nodes where some of the nodes are connected to each other through links.

Python Specific Data Structures

These data structures are specific to python language and they give greater flexibility in storing different types of data and faster processing in python environment.

- **List:** It is similar to array with the exception that the data elements can be of different data types. You can have both numeric and string data in a python list.
- **Tuple:** Tuples are similar to lists but they are immutable which means the values in a tuple cannot be modified they can only be read.
- **Dictionary:** The dictionary contains Key-value pairs as its data elements.

Algorithms Basics

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.

- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

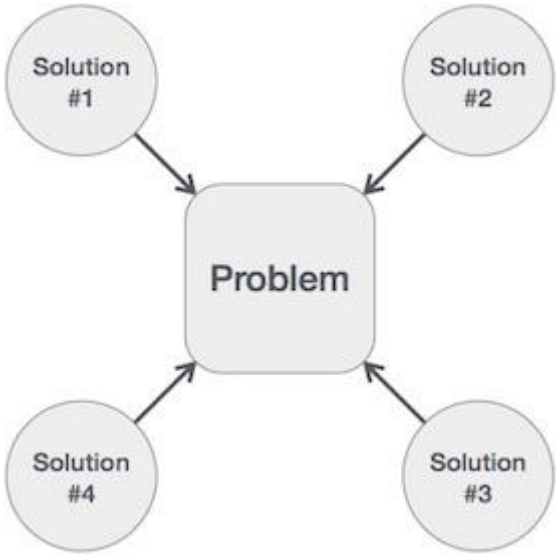
How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

step 1 − START
step 2 − declare three integers **a**, **b** & **c**
step 3 − define values of **a** & **b**
step 4 − add values of **a** & **b**
step 5 − store output of step 4 to **c**
step 6 − print **c**
step 7 − STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as

step 1 − START ADD
step 2 − get values of **a** & **b**
step 3 − c ← a + b
step 4 − display c
step 5 – STOP

Asymptotic Analysis of algorithm

The efficiency and accuracy of algorithms have to be analysed to compare them and choose a specific algorithm for certain scenarios. The process of making this analysis is called Asymptotic analysis. It refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be nearly the same if n is significantly small.

Usually, the time required by an algorithm falls under three types −

- Best Case − Minimum time required for program execution.
- Average Case − Average time required for program execution.
- Worst Case − Maximum time required for program execution.

EXERCISE :

1. Discuss the characteristics of Algorithms.

2. How do you differentiate linear and non- linear data structures , Also give some examples of both data structures.

3. name some commonly used abstract data types .

4. what do you understand by the term Algorithm analysis. Discuss.

Lab 02

Array data structures

i. Insertion ii. Deletion iii. Traversing

Arrays

Arrays are referred to as structured data types. An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations. The elements of array are accessed through index set containing 'n' consecutive numbers.

finite means data range must be defined.

ordered means data must be stored in continuous memory addresses.

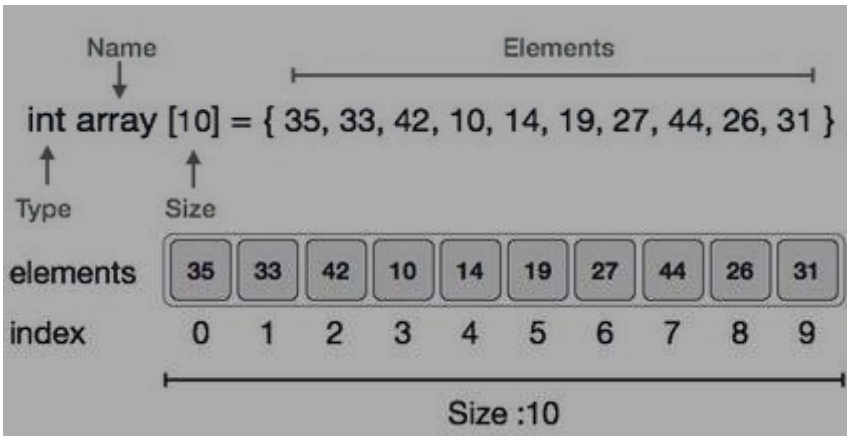
homogenous means data must be of similar data type. OR

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

Element − Each item stored in an array is called an element.
Index − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elements in following order.

Data Type	Default Value
Bool	False
Char	0
Int	0

Insertion Operation in Array

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Types of Insertions:

1. Insertion at the beginning of array
2. Insertion at the end of array
3. Insertion in the middle of array

ALGORITHM:

Variables:

LA: Liner Array
N: Number of elements in LA
K: Position where insertion should be done
ITEM: An element that needs to be inserted

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] = ITEM
6. Set N=N+1
7. Exit

Traversal Operation in an Array

This operation is to traverse through the elements of an array.

Variables:

START: Initialized with starting index of array.
N: Number of elements in array
A: Variable for array

Algorithm:

1. START = 0
2. Repeat Step3 while (START<N)
3. Read A [START]
4. START = START + 1

Deletion Operation in Array

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Types of Deletions:

1. Deletion at the beginning of array
2. Deletion at the end of array
3. Deletion in the middle of array

ALGORITHM:

Variables:

LA: Liner Array
N: Number of elements in LA
K: Position at which deletion should be done

Algorithm:

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

Insertion and deletion are costly operation if the selected index is the first or near to first index of array.

EXERCISE:

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.

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

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

4. Consider the array in question 2 ,Use deletion algorithm(Give implementation) to remove an element
   i. 1 at index 0 ii. 5 at index 2 iii. 3 at index 4

5. If the size of given array is 4 then calculate the Time complexity of deletion algorithm when ,
   i. Insert element at the beginning of array
   iii. Insert element at the end of array
   iv. Insert element in the middle of array

Lab 03

Searching Algorithms

i. Linear Search ii. Binary Search

Searching refers to the process / operation of finding the location of the ITEM/Value specified by the user in a data structure. The location is basically the index where that item/value is stored.

Three outcomes are possible:

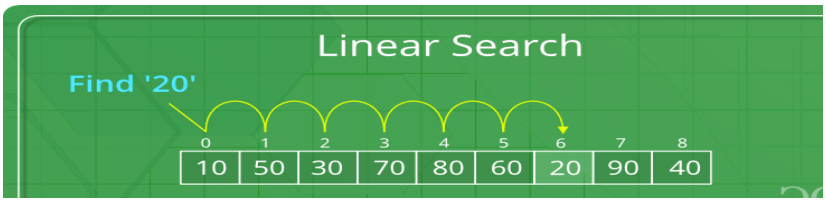
- The item does not exist
- The item exists once in a data structure
- The item exist more than once in a data structure.

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search.
2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures. These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half. For Example: Binary Search.

LINEAR SEARCHING:

Linear search operates by checking every element of a list one at a time in sequence until a match is found. It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1. Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.



A simple approach is to do a **linear search**, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.

Linear Search Algorithm

Variables:

DATA: Liner Array

N: Number of elements in DATA

ITEM: The value which is to be searched

LOC: Location of ITEM in DATA

1. Set DATA[N] = ITEM
2. Set LOC = 0
3. Repeat while DATA[LOC] ≠ ITEM
4. Set LOC = LOC+1
5. If LOC = N
6. Set LOC= -1
7. Return LOC

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

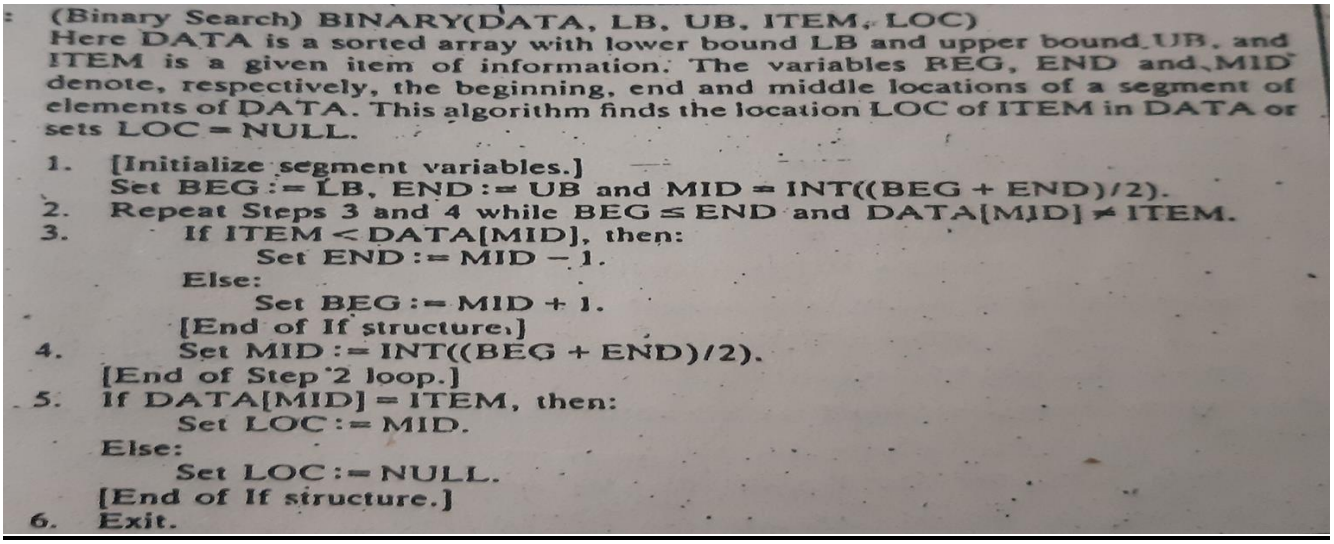
BINARY SEARCHING:

This algorithm works on the principle of divide and conquer .But for this algorithm to work properly, the data collection should be in the sorted form. It works by repeatedly dividing in half , the portion of the list that could contain the value, until the possible location is narrowed down to just one.One of the most common ways to use binary search is to find an item in an array.
In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.

3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Algorithm



(Binary Search) BINARY(DATA, LB, UB, ITEM, LOC)
Here DATA is a sorted array with lower bound LB and upper bound UB, and ITEM is a given item of information. The variables BEG, END and MID denote, respectively, the beginning, end and middle locations of a segment of elements of DATA. This algorithm finds the location LOC of ITEM in DATA or sets LOC = NULL.

1. [Initialize segment variables.]
   Set BEG := LB, END := UB and MID = INT((BEG + END)/2).
2. Repeat Steps 3 and 4 while BEG ≤ END and DATA[MID] ≠ ITEM.
3. If ITEM < DATA[MID], then:
   Set END := MID − 1.
   Else:
   Set BEG := MID + 1.
   [End of If structure.]
4. Set MID := INT((BEG + END)/2).
   [End of Step 2 loop.]
5. If DATA[MID] = ITEM, then:
   Set LOC := MID.
   Else:
   Set LOC := NULL.
   [End of If structure.]
6. Exit.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)**

EXERCISE:

1. 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

2. 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

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

Lab # 04

Sorting Operation in an Array

i. Bubble Sort Algorithm ii. Quick Sort Algorithm

Data sorting is the process that involves arranging the data into some meaningful order to make it easier to understand, analyze or visualize. Sorting refers to ordering data in an increasing or decreasing fashion according to some linear relationship among the data items. Sorting can be done on names, numbers and records.

Bubble Sort

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on. If we have total n elements, then we need to repeat this process for n-1 times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Steps:

- Bubble sort algorithms cycle through a list, analyzing pairs of elements from left to right, or beginning to end.
- If the leftmost element in the pair is less than the rightmost element, the pair will remain in that order.
- If the rightmost element is less than the leftmost element, then the two elements will be switched.
- This cycle repeats from beginning to end until a pass in which no switch occurs.

Algorithm

Variable:

- DATA: Linear Array
- N: Number of elements in array

- Algorithm:

1. Repeat Step 2 and 3 for K= 0 to N-1
2. Set PTR =0
3. Repeat while PTR < N-K-1
    1. If DATA[PTR]>DATA[PTR+1] then Interchange DATA[PTR] and DATA[PTR+1]
    2. Set PTR=PTR+1
4. Exit

Example: Assume Array A[]={ 5,1,4,2,8}
First Pass:
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:
(**1 2** 4 5 8) –> (**1 2** 4 5 8)
(1 **2 4** 5 8) –> (1 **2 4** 5 8)
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Features of Bubble Sort Algorithm:

Worst Case Time Complexity: In Bubble Sort, `n-1` comparisons will be done in the 1st pass, `n-2` in 2nd pass, `n-3` in 3rd pass and so on. So the total number of comparisons will be, O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Quick Sort :

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing. It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

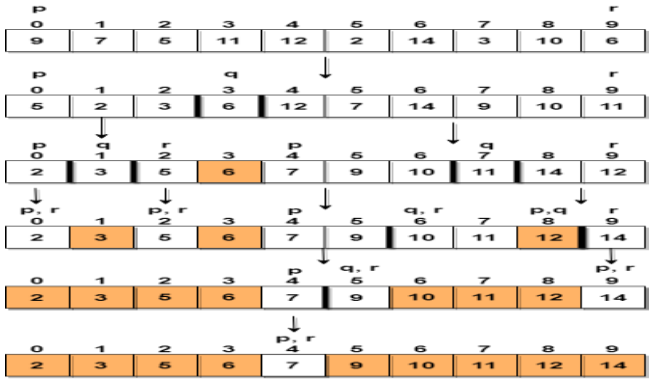
1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element.

Steps:

- After selecting an element as **pivot**,
- In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the **pivot** will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
- And the **pivot** element will be at its final **sorted** position.
- The elements to the left and right, may not be sorted.
- Then we pick subarrays, elements on the left of **pivot** and elements on the right of **pivot**, and we perform **partitioning** on them by choosing a **pivot** in the subarrays.

For example:



Quick Sort Algorithm

Partition(A,Beg,End) ,A= Array ,Beg=Starting index of Array,End= last index of Array, It also uses Local variables left,right and Loc. Left & Right will contain boundry values of the list, loc keeps track of position of the first element.

```
1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
2. [Scan from right to left.]
     (a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
           RIGHT := RIGHT − 1.
         [End of loop.]
     (b) If LOC = RIGHT, then: Return.
     (c) If A[LOC] > A[RIGHT], then:
           (i) [Interchange A[LOC] and A[RIGHT].]
                 TEMP := A[LOC], A[LOC] := A[RIGHT],
                 A[RIGHT] := TEMP.
          (ii) Set LOC := RIGHT.
         (iii) Go to Step 3.
         [End of If structure.]
3. [Scan from left to right.]
     (a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
           LEFT := LEFT + 1.
         [End of loop.]
     (b) If LOC = LEFT, then: Return.
     (c) If A[LEFT] > A[LOC], then
           (i) [Interchange A[LEFT] and A[LOC].]
                 TEMP := A[LOC], A[LOC] := A[LEFT],
                 A[LEFT] := TEMP.
          (ii) Set LOC := LEFT.
         (iii) Go to Step 2.
         [End of If structure.]
```

Features of Quick Sort Algorithm:

Worst Case Time Complexity: For an array, in which partitioning leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.And if keep on getting unbalanced subarrays, then the running time is the worst case, which is **$O(n^2)$.**

Best Case Time Complexity: Where as if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O(n*log n)**.

EXERCISE:

1. Give implementation of bubble sort algorithm let's assume Array "A" is consisting of 6 elements which are stored in descending order.

2. Re-write bubble sort algorithm for recursive function call, Also analyze the time complexity of recursive bubble sort algorithm.

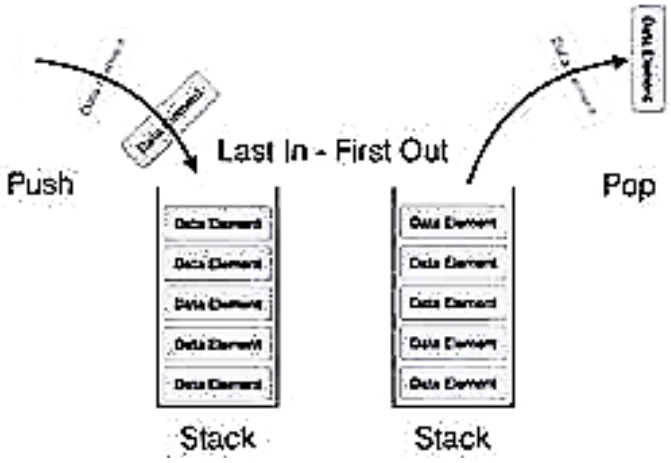
3. Implement the Quick sort Algorithm on Array A[]={ 9,7,5,11,12,2,14,3,10,6}

4. Compare the two sorting algorithms (discuss in this session) and discuss their advantages and disadvantages.

Lab 05

Stack Data Structure

A stack is a container of elements that are inserted and removed according to the last-in first-out **(LIFO)** principle**.** OR A **stack** is an ordered list in which all insertions and deletions are made at one end, called the top.The element at the top is called the top element. The operations of inserting and deleting elements are called push() and pop() respectively. Given a stack S=(a[1],a[2],…a[n])then we can say that a1 is the bottom most element and element a[n] is on the top of element.



Insertion or Push Operation:

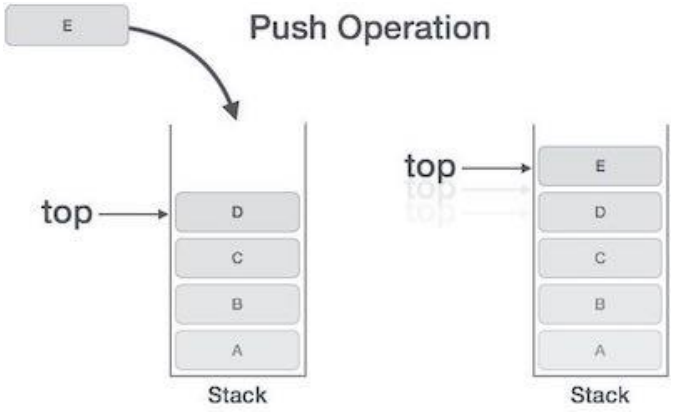
- Insertion or push operation requires following steps in general:
    - If stack is already filled then insertion can not happen
    - Increase the size of stack and then insert the desired elements

PUSH Algorithm

PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.

Example for Insertion or Push Operation Algorithm



Deletion or Pop Operation

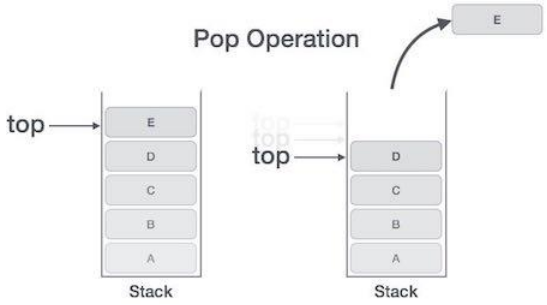
- Deletion or pop operation requires following steps:
  - If the stack has no element then deletion can not happen
  - Remove the item from top of stack
  - Decrease the size of stack.

POP Algorithm

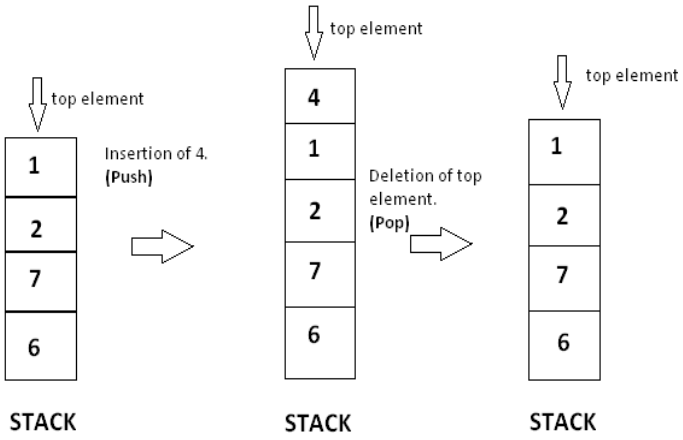
POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP – 1. [Decreases TOP by 1.]
4. Return.

Example for Deletion or Pop Operation Algorithm



Push & Pop Operations



Applications of Stack

There are many situations where a stack can be useful, a few implementations of it's use could be found in:

- Backtracking features - This could be an undo feature in a text editing application or to a previous choice point in a game. The stack simply allows us to pop the previous item from it's data structure.
- Recursive algorithms - During recursion, we sometimes need to push temporary data onto a stack, popping the data as we back track through the stages of our algorithm.

EXERCISE:

1. Give implementations of Push & Pop Algorithms with underflow & Overflow exceptions

2. What should be the time complexity of algorithms (discuss in this session) in your opinion? Discuss.

3. Write algorithm to Reverse a string using stack data structure also gives implementation.

4. Write algorithm to implement two stacks in a single array

A simple solution would be to divide the array in two halves and allocate each half for implementing two stacks. In other words, if given an array A of size n, the solution would allocate `A[0,n/2]` memory for first stack and `A[n/2+1,n-1]` memory for the second stack. The problem with this approach is that it doesn't efficiently utilize the available space in the array. For instance if one half of the array is full, any subsequent push operations would lead to stack overflow exception even if other half has space available.

To handle this we can grow stack from two extreme corners of the array. In other words, the first stack grows from the `0'th` index and the second stack grows from the `(n-1)'th` index where n is the size of the array. Both stacks can grow towards each other with no fixed capacity. Now overflow will only happen if both stacks are full (i.e. top elements of both stacks are adjacent) and there is no space left in the array to accommodate a new element.

Lab 06

Expression Evaluation through Stack Data Structure

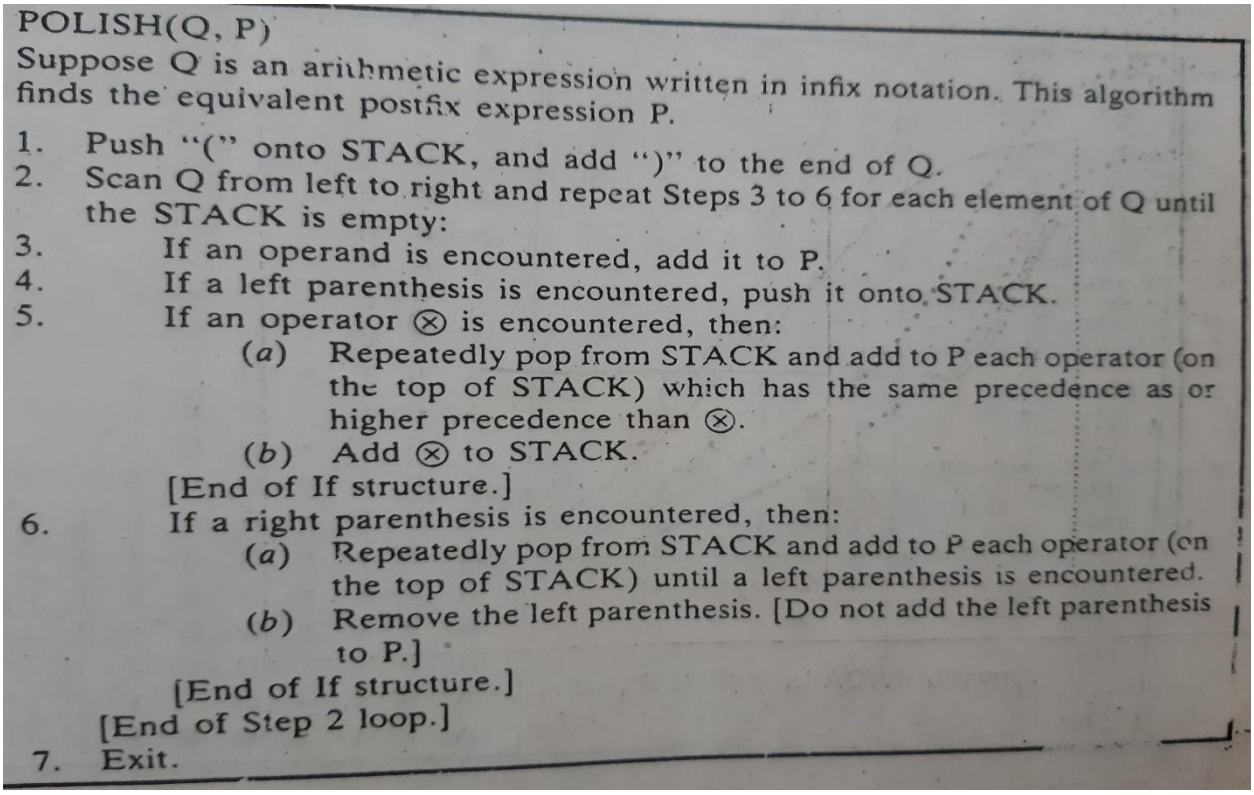
i. Infix ii. Postfix iii. Prefix

Stack can be use to evaluate expressions. The expression can contain parentheses and operators. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

i. Infix Notation: Operators are written between the operands they operate on, e.g. 3 + 4
ii. Prefix OR Polish notation: Operators are written before the operands, e.g + 3 4
iii. Postfix OR Reverse Polish Notation: Operators are written after operands, e.g. 3 4 +

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms.

Algorithm for Converting an Infix expression into Postfix Expression



POLISH(Q, P)
Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
   (*b*) Add ⊗ to STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
7. Exit.

Algorithm for Converting an Infix expression into Prefix Expression

1. Step 1. Push ")" onto STACK, and add "(" to end of the A
2. Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
3. Step 3. If an operand is encountered add it to B
4. Step 4. If a right parenthesis is encountered push it onto STACK
5. Step 5. If an operator is encountered then:
   a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
   b. Add operator to STACK
6. Step 6. If left parenthesis is encontered then
   a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
   b. Remove the left parenthesis
7. Step 7. Exit

Algorithm for solving Postfix Expression

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

1. Add a right parenthesis "")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel "")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator $\otimes$ is encountered, then:
   (*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (*b*) Evaluate B $\otimes$ A.
   (*c*) Place the result of (*b*) back on STACK.
   [End of If structure.]
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

EXERCISE:

1. Solve the following postfix expressions via algorithm
   i. P: 5 ,6,2,+,*, 12,4,/,- ii. 2,3,^,5,2,2,^,*,12,6,/,-,+
2. Implement the algorithm to convert Q: (A + (B * C – (D / E ↑ F) * G) * H) infix expression to equivalent postfix form.
3. Write algorithm to solve prefix expression also give implementation.

Lab 07

Queue Data Structure

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing operation. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Queue Representation

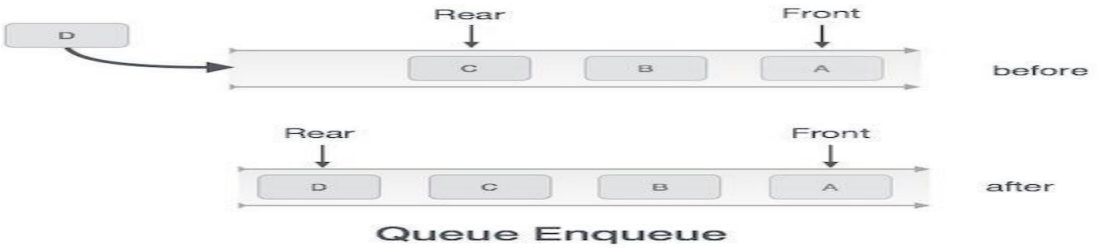
Basic Operations

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue. Enqueue Operation

enqueue Operation:

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.The following steps should be taken to enqueue (insert) data into a queue :

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Algorithm for enqueue operation

procedure enqueue(data)

```
  if queue is full
    return overflow
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true

end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation :

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.

Algorithm for dequeue operation

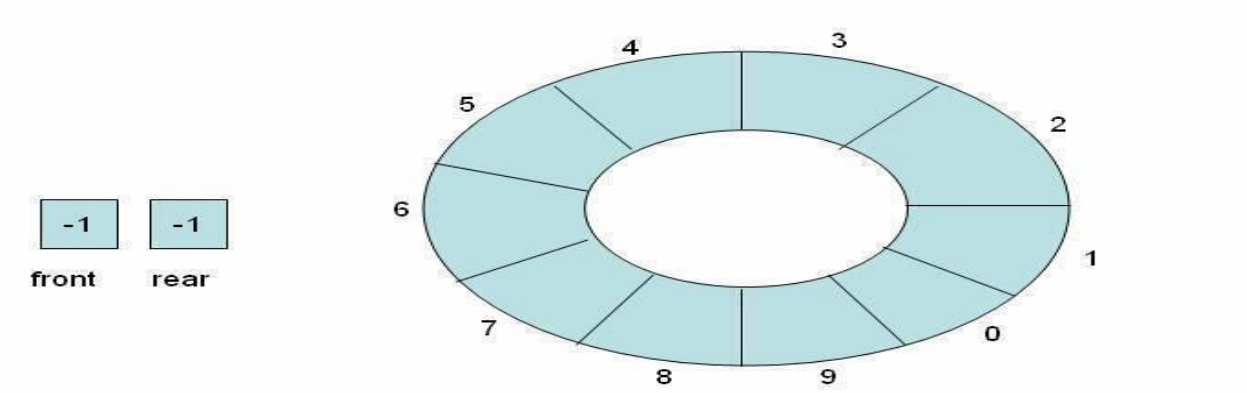
```
procedure dequeue
  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```

Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'.



Applications of Queue Data Structure

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.

EXERCISE:

1. Write Algorithms for Enqueue and dequeue operations in a circular queue.

2. Give implementation of queue data structure using two stacks (let S1 & S2 be the two stacks for push and pop operations respectively).

Lab 08

Recursive Algorithms (Recursion)

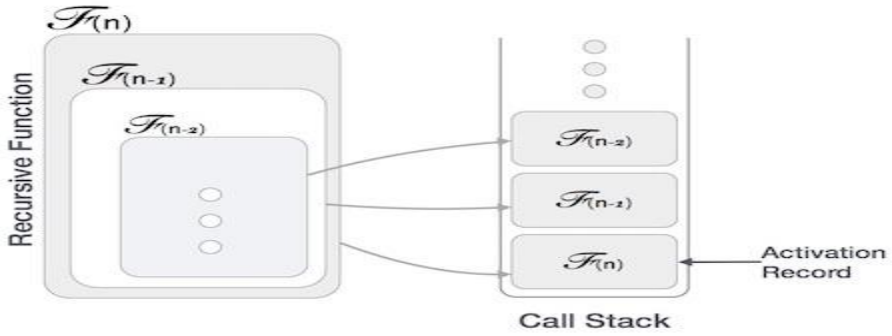
Computer programming languages allow a module or function to call itself. This technique is known as recursion. In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.

Properties:

A recursive function can go infinite like a loop. To avoid infinite running of recursive function, there are two properties that a recursive function must have :

- **Base criteria** − There must be at least one base criteria or condition, such that, when this condition is met the function stops calling itself recursively.
- **Progressive approach** − The recursive calls should progress in such a way that each time a recursive call is made it comes closer to the base criteria.

Many programming languages implement recursion by means of **stacks**. Generally, whenever a function (**caller**) calls another function (**callee**) or itself as callee, the caller function transfers execution control to the callee. This transfer process may also involve some data to be passed from the caller to the callee. This implies, the caller function has to suspend its execution temporarily and resume later when the execution control returns from the callee function. Here, the caller function needs to start exactly from the point of execution where it puts itself on hold. It also needs the exact same data values it was working on. For this purpose, an activation record (or stack frame) is created for the caller function.



Recursive Fibonacci Algorithm

Fibonacci series generates the subsequent number by adding two previous numbers. Fibonacci series starts from two numbers − **F_0 & F_1**. The initial values of F_0 & F_1 can be taken 0, 1 or 1, 1 respectively. Fibonacci series satisfies the following conditions:

$F_n = F_{n-1} + F_{n-2}$

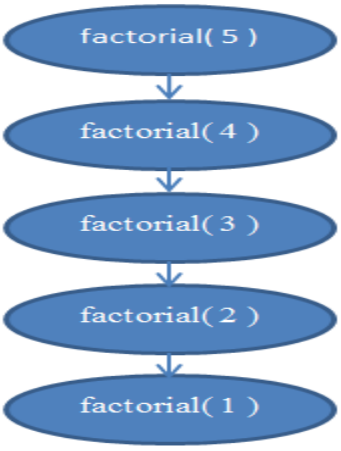
Hence, a Fibonacci series can look like this :
F_8 = 0 1 1 2 3 5 8 13
or, this :
F_8 = 1 1 2 3 5 8 13 21

Fib(int n)
1. If nis less than OR equals to 1
2. return 1
3. Else
return fib(n-1)+fib(n-2)

Factorial Recursive Algorithm

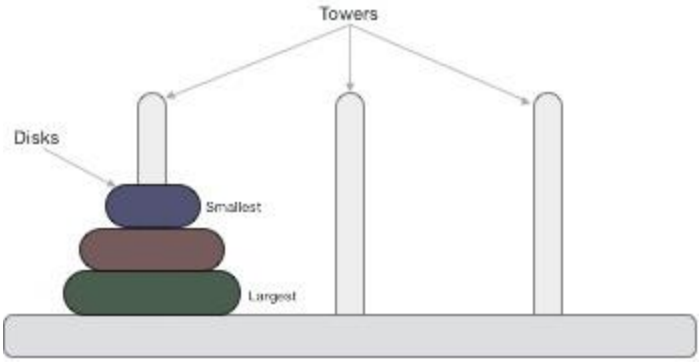
1. func factorial(n)
2. if (n == 1)
3. return 1
4. return n * factorial(n -1)

For factorial(5) We get the following Number of iterations
1) factorial(5) = 5 * factorial(4)
2) factorial(4) = 4 * factorial(3)
3) factorial(3) = 3 * factorial(2)
4) factorial(2) = 2 * factorial(1)
5) factorial(1) = 1



Tower of Hanoi:

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted:



Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

The above puzzle can be easily solve by using recursive approach.

Analysis of Recursion

One may argue why to use recursion, as the same task can be done with iteration. The first reason is, recursion makes a program more readable and because of latest enhanced CPU systems, recursion is more efficient than iterations.

Time Complexity

In case of iterations, we take number of iterations to count the time complexity. Likewise, in case of recursion, assuming everything is constant, we try to figure out the number of times a recursive call is being made. A call made to a function is O(1), hence the (n) number of times a recursive call is made makes the recursive function O(n).

Space Complexity

Space complexity is counted as what amount of extra space is required for a module to execute. In case of iterations, the compiler hardly requires any extra space. The compiler keeps updating the values of variables used in the iterations. But in case of recursion, the system needs to store activation record each time a recursive call is made. Hence, it is considered that space complexity of recursive function may go higher than that of a function with iteration.

EXERCISE:

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

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

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

Lab 09

Tree data Structure

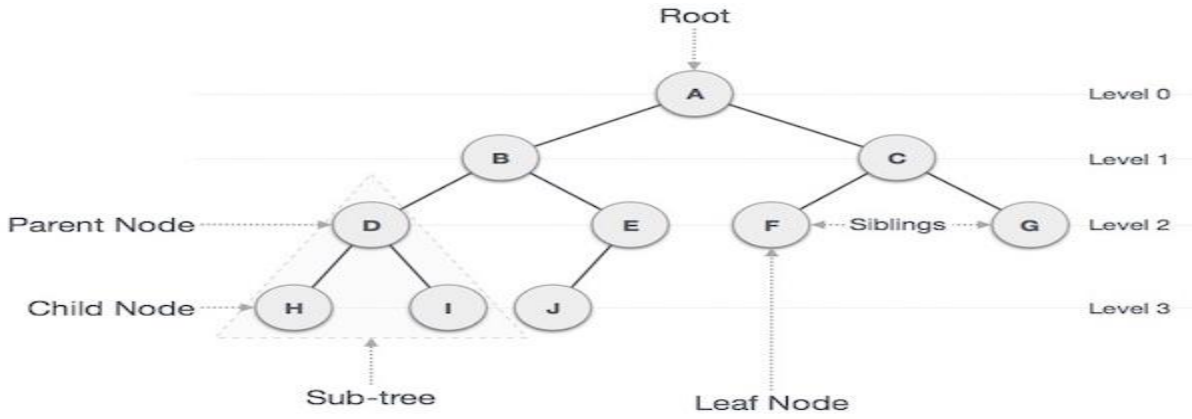
A ***tree*** is a ***nonlinear*** data structure used to represent entities that are in some ***hierarchical*** relationship. Generally, tree represents tree represents the nodes connected by edges

Examples in real life:

- Family tree
- Table of contents of a book
- Computer file system (folders and subfolders)
- Decision trees
- Top-down design

Binary Tree:

Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Difference between A Tree & Binary Tree

In General Tree, each node can have infinite number of children. Binary tree is the type of tree in which each parent can have at most two children. The children are referred to as left child or right child.(no node in a binary tree may have a degree more than 2).

Tree Terminologies:

Following are the important terms with respect to tree.

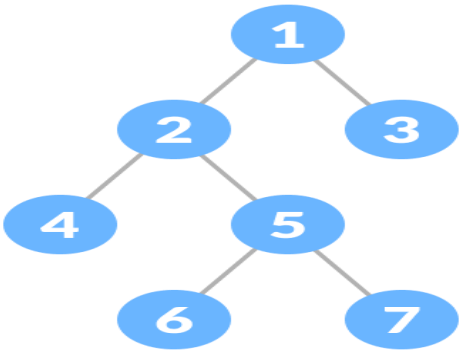
- Path − Path refers to the sequence of nodes along the edges of a tree.
- Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- Parent − Any node except the root node has one edge upward to a node called parent. the node directly above in the hierarchy
- Child − The node below a given node connected by its edge downward is called its child node.
- Leaf − The node which does not have any child node is called the leaf node.
- Subtree − Subtree represents the descendants of a node.
- Visiting − Visiting refers to checking the value of a node when control is on the node.
- Traversing − Traversing means passing through nodes in a specific order.
- Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- keys − Key represents a value of a node based on which a search operation is to be carried out for a node.
- Nodes- the elements in the tree
- Edges-connections between nodes
- Interior node-a node that is not a leaf node
- Empty tree- has no nodes and no edges
- Siblings:-nodes that have the same parent
- Ancestors of a node: -its parent, the parent of its parent, etc.
- Descendants of a node- its children, the children of its children, etc.
- Degree of a node- the number of children it has
- Degree of a tree- the maximum of the degrees of the tree's nodes
- Height of a Node-The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

- Height of a Tree-The height of a Tree is the height of the root node or the depth of the deepest node.
- Size of tree- total number of nodes in a tree

Types of Binary Tree:

Full/proper Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. An improper Binary tree can be converted into proper binary tree by viewing missing nodes as Null nodes.

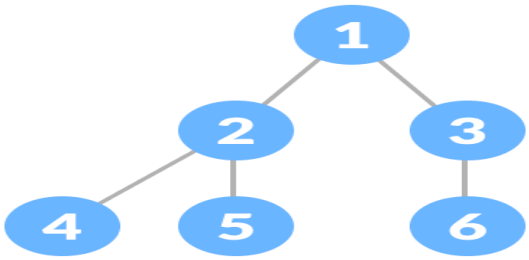


Complete Binary Tree

A complete binary tree is just like a full binary tree, but with some major differences

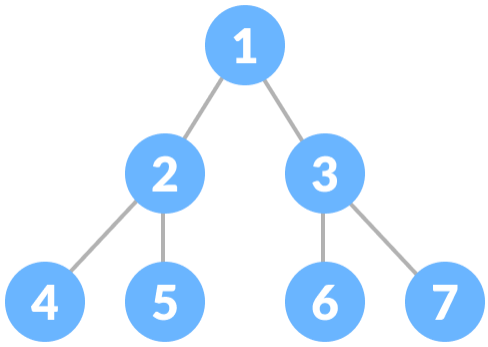
1. Every level must be completely filled
2. All the leaf elements must lean towards the left.

The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level. A perfect binary tree has the maximum no. of nodes for a given height. It has (2^(n+1)-1) nodes where n is the height of the tree.



Binary Tree Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Delete**- Delete a node from tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

Insertion in BST

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm steps

1. Check, whether the tree is empty or Not

2. if a new value is less, than the node's value:

   a. if a current node has no left child, place for insertion has been found;

   b. otherwise, handle the left child with the same algorithm.

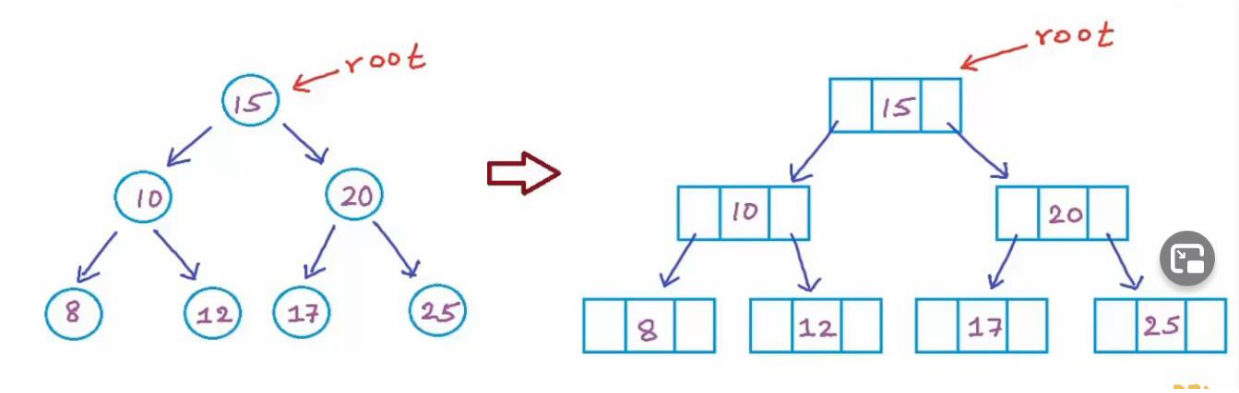
3. if a new value is greater, than the node's value:

   a. if a current node has no right child, place for insertion has been found;
   b. otherwise, handle the right child with the same algorithm.

BT Data Structure representation

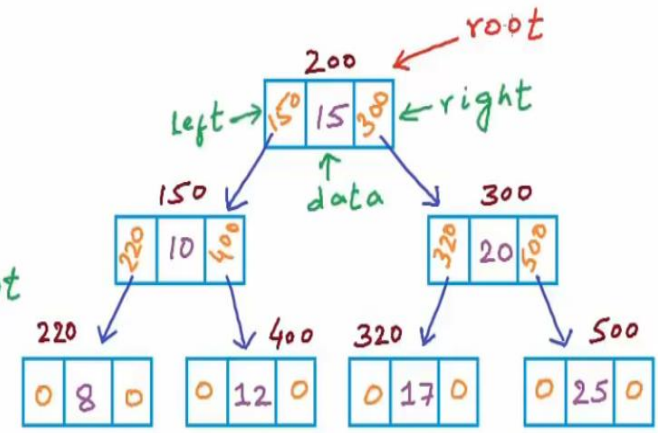
Left Pointer	Data	Right Pointer

Left pointer -> Sore the address of left -child
Data → Node value
Right pointer-> Sore the address of Right -child




```
struct BstNode {
    int data;
    BstNode* left;
    BstNode* right;
};

BstNode* rootPtr; // to store address of root node
```



Insertion Algorithm:

```
INSERT(T, n)
  temp = T.root
  y = NULL
  while temp != NULL
     y = temp
```

```
    if n.data < temp.data
       temp = temp.left
    else
       temp = temp.right
 n.parent = y
 if y==NULL
    T.root = n
 else if n.data < y.data
    y.left = n
 else
    y.right = n
```

Here, we starting from the root of the tree - temp = T.root and then moving to the left subtree if the data of the node to be inserted is less than the current node - if n.data < temp.data → temp = temp.left; Otherwise, we are moving right. variable y is used When the tree won't have any node, the new node will be the root of the tree and its parent will be NULL. So, initially the value of *y* is NULL. In this case, the loop will also not run. Otherwise, *y* will point to the last node. Lastly, we need to make the new node the child of *y*. If *y* is null, the new node will be the root of the tree, otherwise we will check if the data of the new node is larger or smaller than the data of *y*, and accordingly we will make it either the left or the right child.

Deletion in BST

General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has one child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
   a. right most node of it's left sub-tree or
   b. left most node of it's right sub-tree.
6. End

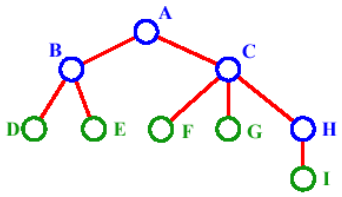
EXERCISE:

1. Construct a binary tree (initially empty) , insert nodes 15,10,20,25,8,12 with the help of following definition of Node in BST

```
Struct BSTNode{
  Int data;
  BSTNode* left;
 BSTNode* right; }  BSTNode* rootPtr;
```

2. Delete One left-child-node and one right-child- node from above tree. Analyze the change occur in the tree after deletion.

3. From the figure below , identify i. root ii. Height of tree iii. Degree iv. Size v. leafnode(s)



4. Draw a tree from the following representation: and identify i. Child of H ii. Height of Tree iii. Leaf node(s) iv. Parent of A.

D→F
D→B
K→J
K→L
B→A
B→C
H→ D
H→K
F→E
F→G
J→I

5. write down the applications of Tree data structure

Lab 10

Tree Traversal Algorithms

i. Inorder ii. Preorder iii. Postorder

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.

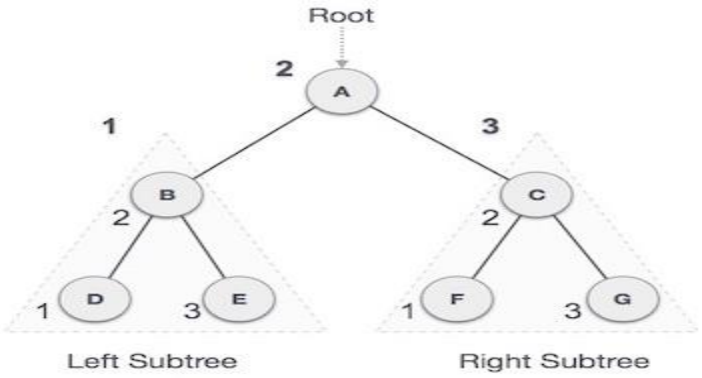


Figure 1

We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be − $D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

In-Order Algorithm

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.

Pre-order Traversal
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree. Refer Figure 1,We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be $-A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

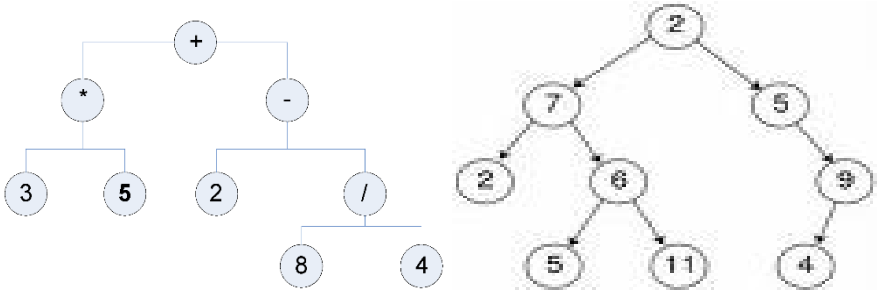
Pre-Order Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

Post-order Traversal
In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node. Refer to Figure 1, We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be $-D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Post- Order Algorithm
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

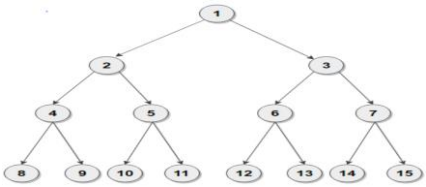
EXERCISE:

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



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

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



For example, there are two ways to print below tree –

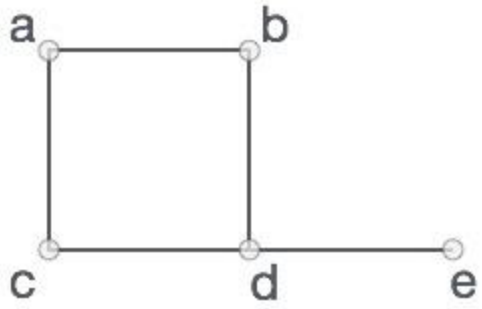
Variation 1: Print Top-Down

```
(1, 2, 3, 4, 7, 5, 6, 8, 15, 9, 14, 10, 13, 11, 12)
```

Lab 11

Graph Data Structure

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**. Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph :

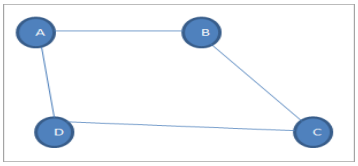


In the above graph,
V = {a, b, c, d, e}
E = {ab, ac, bd, cd, de}

Graph Terminologies

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

1. **Vertex:** Each node of the graph is called a vertex. In the above graph, A, B, C, and D are the vertices of the graph.
2. **Edge:** The link or path between two vertices is called an edge. It connects two or more vertices. The different edges in the above graph are AB, BC, AD, and DC.
3. **Adjacent node:** In a graph, if two nodes are connected by an edge then they are called adjacent nodes or neighbors. In the above graph, vertices A and B are connected by edge AB. Thus A and B are adjacent nodes.
4. **Degree of the node:** The number of edges that are connected to a particular node is called the degree of the node. In the above graph, node A has a degree 2.
5. **Path:** The sequence of nodes that we need to follow when we have to travel from one vertex to another in a graph is called the path. In our example graph, if we need to go from node A to C, then the path would be A->B->C.
6. **Cycle:** A path in which there are no repeated edges or vertices and the first and last vertices are the same is called a cycle. In the above graph, A->B->C->D->A is a cycle.



7. Complete Graph: A graph in which each node is connected to another is called the Complete graph. If N is the total number of nodes in a graph then the complete graph contains N(N-1)/2 number of edges.
8. Weighted graph: A positive value assigned to each edge indicating its length (distance between the vertices connected by an edge) is called weight. The graph containing weighted edges is called a weighted graph. The weight of an edge e is denoted by w(e) and it indicates the cost of traversing an edge.
9. Diagraph: A digraph is a graph in which every edge is associated with a specific direction and the traversal can be done in specified direction only.
10. Undirected graphs: When the edges in a graph have no direction, the graph is called *undirected.*

Basic Operations

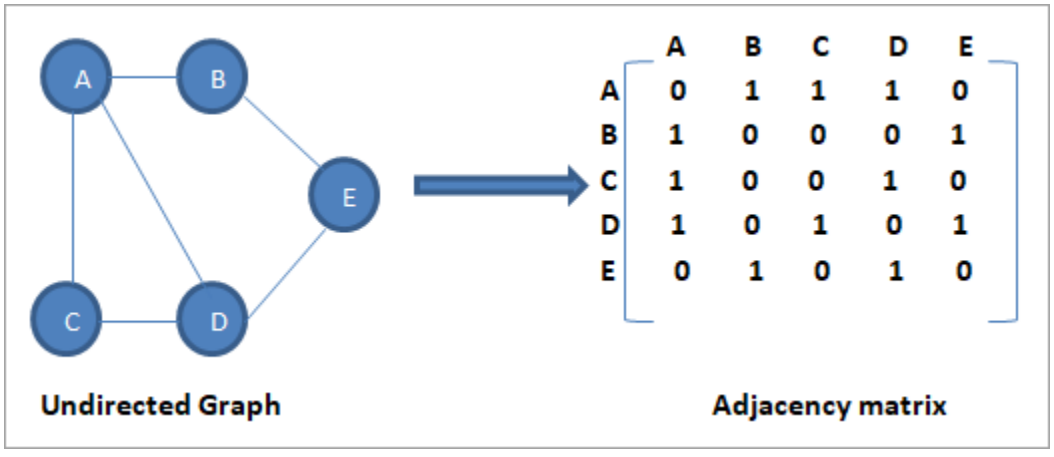
Following are basic primary operations of a Graph :

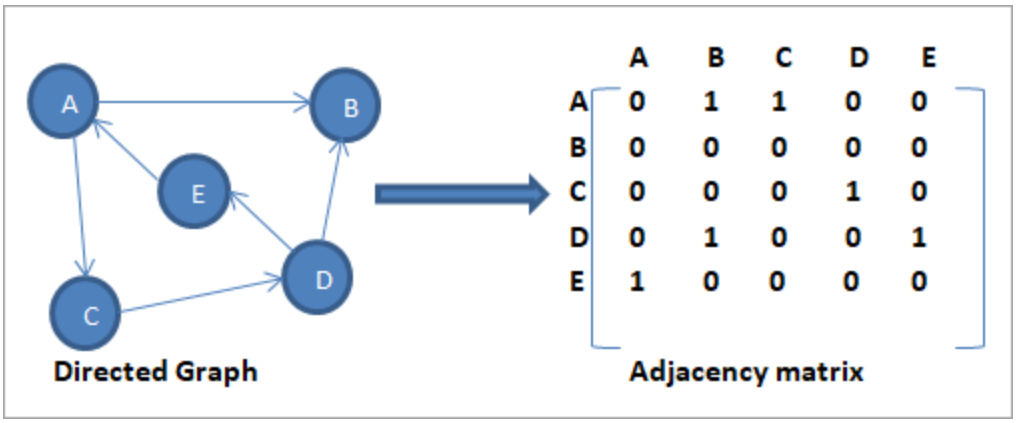
- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph.

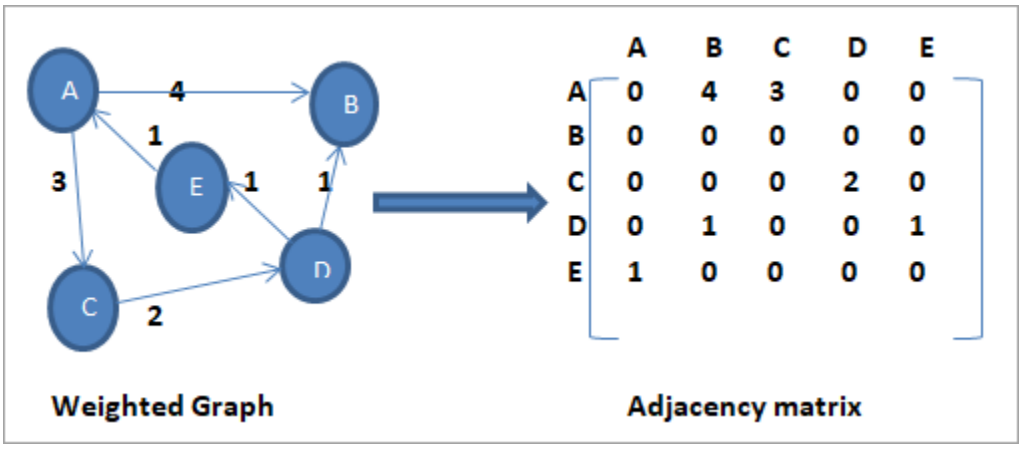
Graph Representation

i. Adjacency Matrix Method

An adjacency matrix is a matrix of size n x n where n is the number of vertices in the graph.
The rows and columns of the adjacency matrix represent the vertices in a graph. The matrix element is set to 1 when there is an edge present between the vertices. If the edge is not present then the element is set to 0.



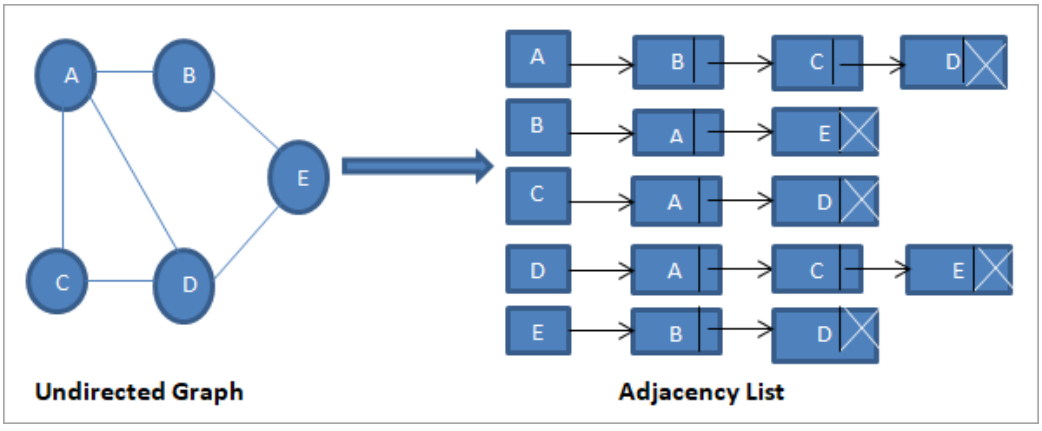


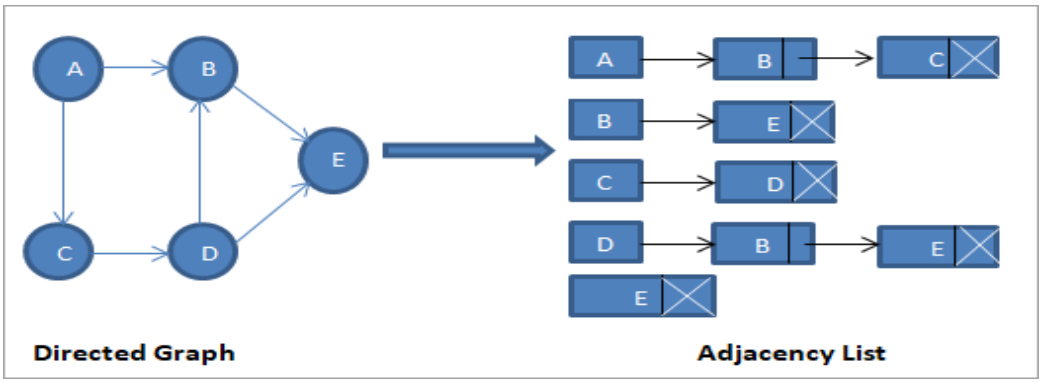


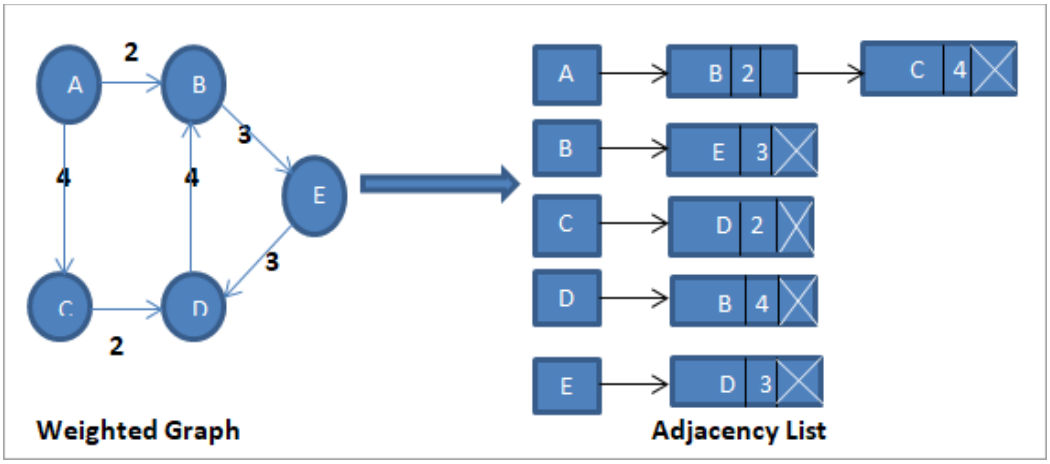
- There are |V| rows and |V| columns in an adjacency matrix, and so the matrix has $|V|^2$ entries
- This is space inefficient for sparse graphs

ii. Adjacency list Method

The adjacency list representation maintains each node of the graph and a link to the nodes that are adjacent to this node. When we traverse all the adjacent nodes, we set the next pointer to null at the end of the list.







- Using an adjacency list representation, each edge in a directed graph is represented by one item in one list; and there are as many lists as there are vertices
- Therefore the storage required is proportional to |V| + |E|, which is much better than $|V|^2$ for sparse graphs, and comparable to $|V|^2$ for dense graphs

Applications of Graph:

- In **Computer science** graphs are used to represent the flow of computation.
- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- 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.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind Google Page Ranking Algorithm.
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.

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.

Lab 12

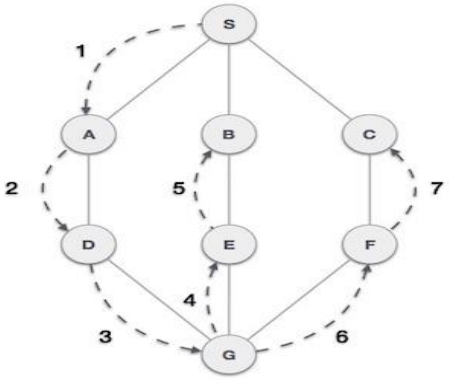
Graph Traversal Algorithms

i. DFS ii. BFS

There are two methods available for traversing in a graph data structure ,one of them is called Depth First Search (DFS) and second one is called Breadth First Search (BFS)algorithm .

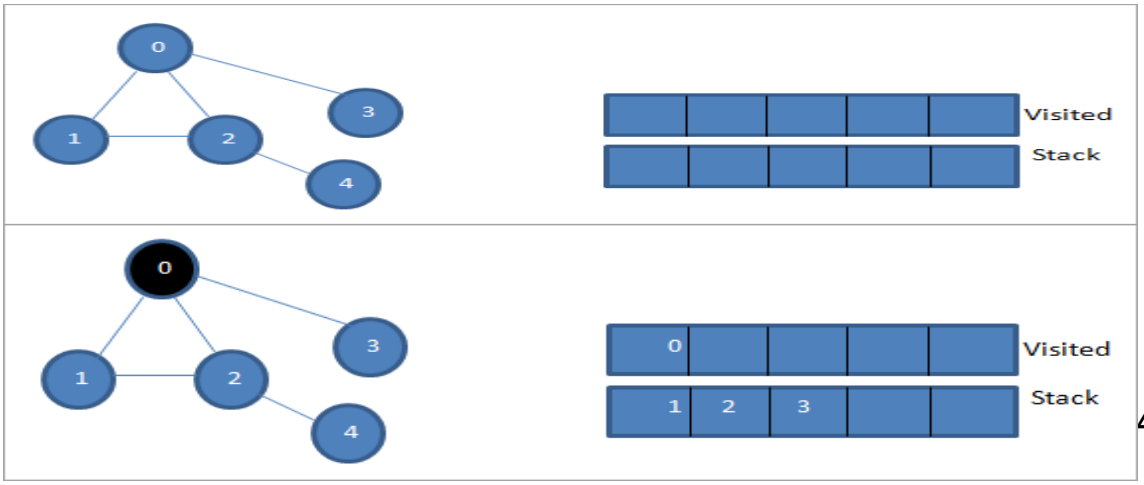
Depth First Search (DFS)

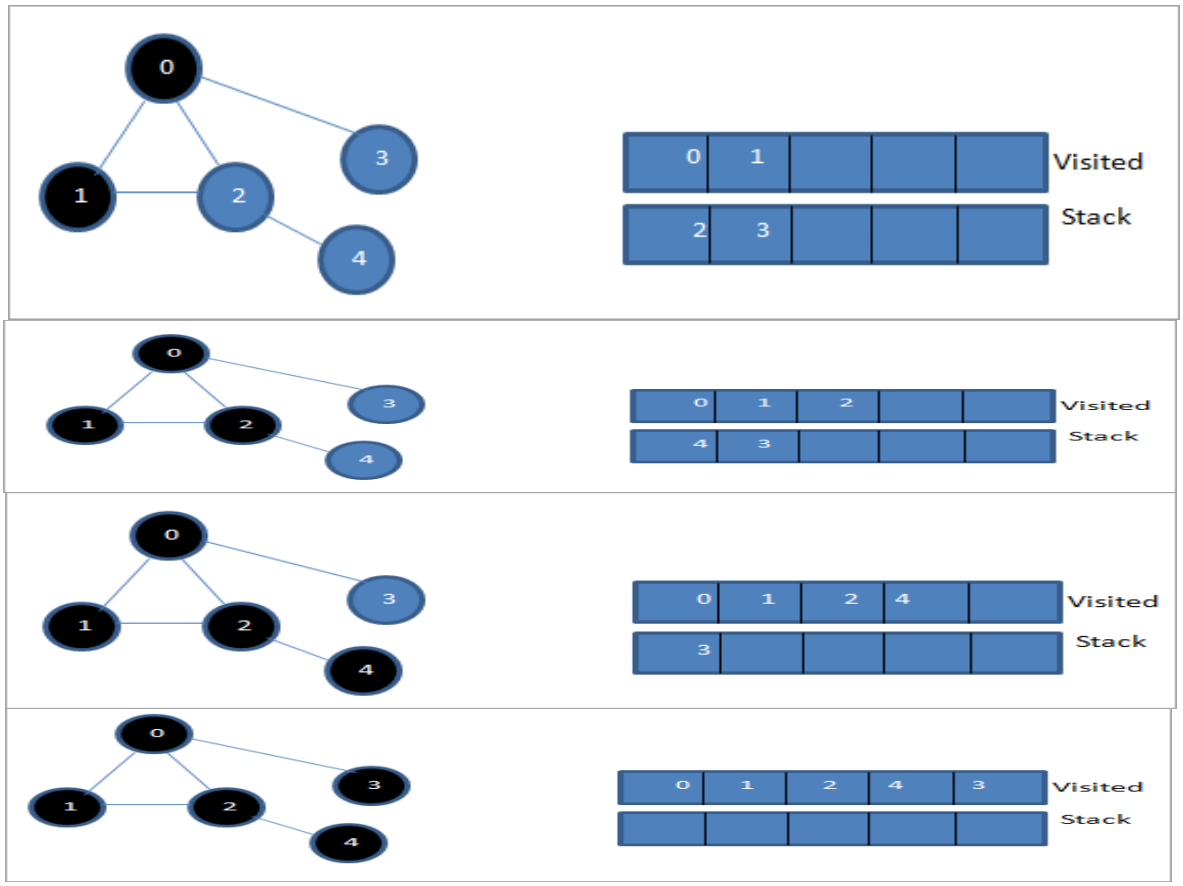
- DFS traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration. DFS can be implemented efficiently using a *stack*



DFS Algorithm

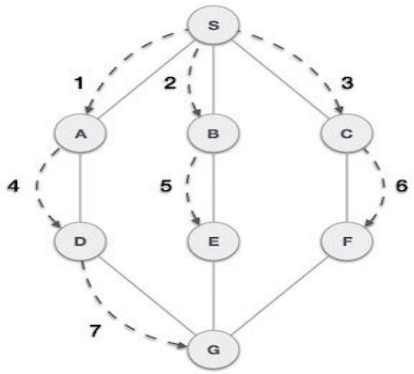
- **Step 1:** Insert the root node or starting node of a tree or a graph in the stack.
- **Step 2:** Pop the top item from the stack and add it to the visited list.
- **Step 3:** Find all the adjacent nodes of the node marked visited and add the ones that are not yet visited, to the stack.
- **Step 4**: Repeat steps 2 and 3 until the stack is empty.





Breadth First Search (BFS)

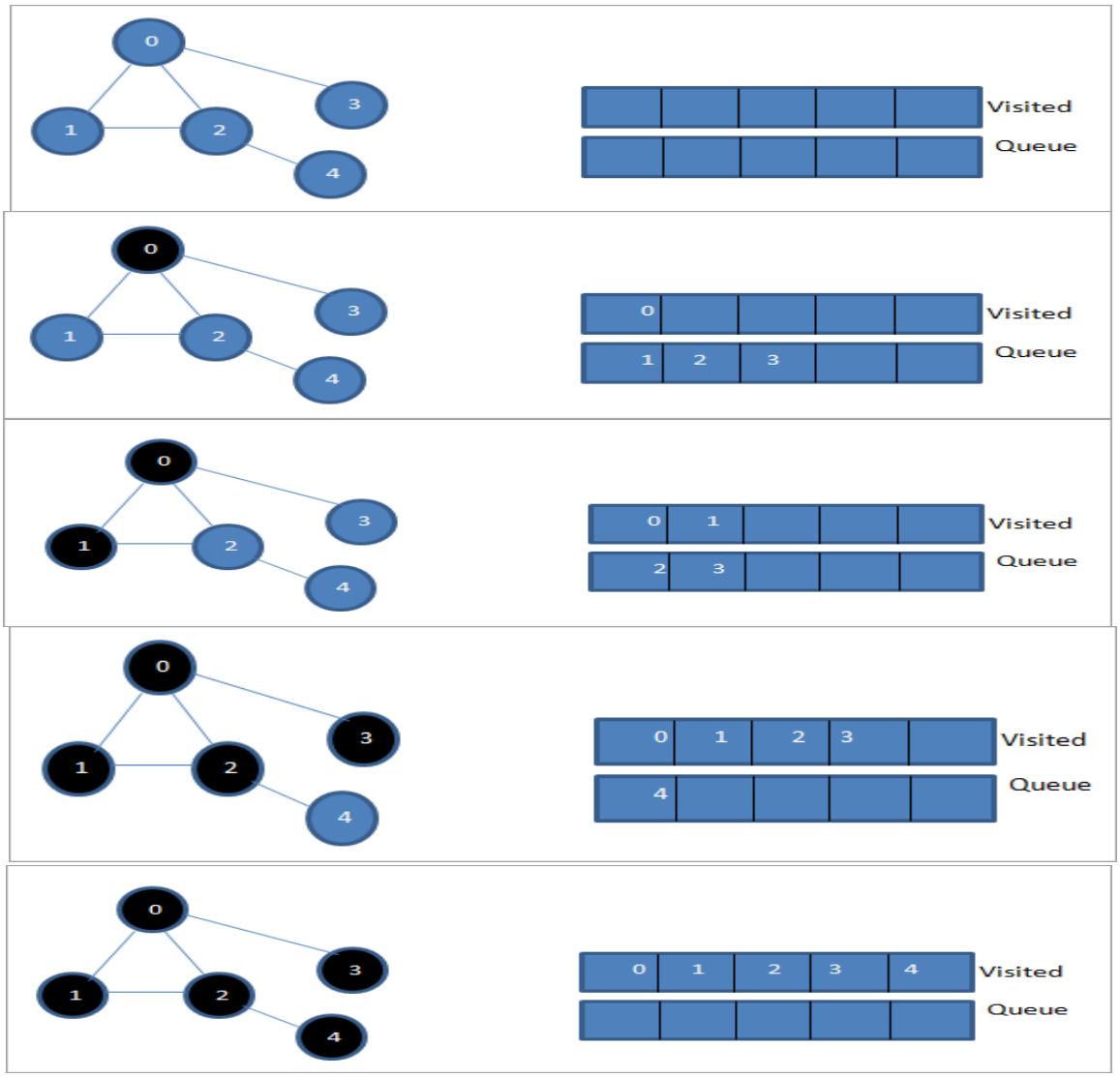
As the name BFS suggests, Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration. BFS can be implemented efficiently using a *Queue*



BFS Algorithm

Consider G as a graph which we are going to traverse using the BFS algorithm. Let S be the root/starting node of the graph.

- **Step 1:** Start with node S and enqueue it to the queue.
- **Step 2:** Repeat the following steps for all the nodes in the graph.
- **Step 3:** Dequeue S and process it.
- **Step 4:** Enqueue all the adjacent nodes of S and process them.
- [END OF LOOP]
- **Step 6:** EXIT



Differences between BFS and DFS

Sr. No.	Key	BFS	DFS
1	Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
2	Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
3	Suitability for decision tree	As BFS considers all neighbors so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
4	Speed	BFS is slower than DFS.	DFS is faster than BFS.
5	Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

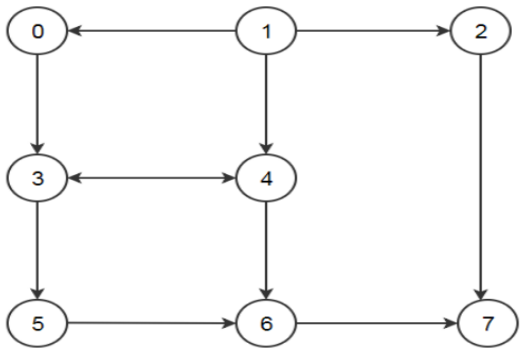
EXERCISE:

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

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

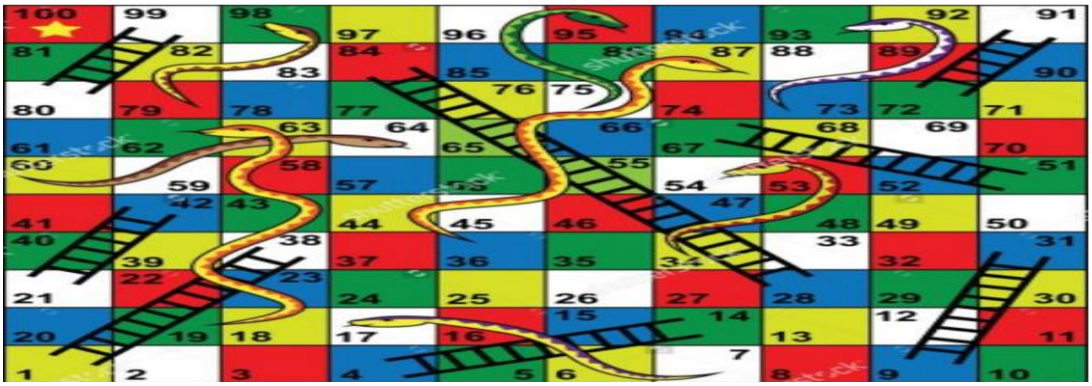
Given a directed graph and two vertices (say source and destination vertex), determine if the destination vertex is reachable from the source vertex or not. If the path exists from the source vertex to the destination vertex, print it.

For example, there exists two paths `{0–3–4–6–7}` and `{0–3–5–6–7}` from vertex 0 to vertex 7 in the following graph. Whereas there is no path from vertex 7 to any other vertex.



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).

For example, the game below requires at least 7 dice throws to win.



The idea is to consider the snake and ladder board as directed graph and run BFS from starting node which is vertex 0 as per game rules. We construct a directed graph keeping in mind below conditions –

1. For any vertex in the graph `v`, we have an edge from `v` to `v + 1`, `v + 2`, `v + 3`, `v + 4`, `v + 5`, `v + 6` as we can reach any of these nodes in one throw of dice from node `v`.

2. If any of these neighbors of `v` has a ladder or snake which takes you to position `x`, then `x` becomes the neighbor instead of the base of the ladder or head of the snake.

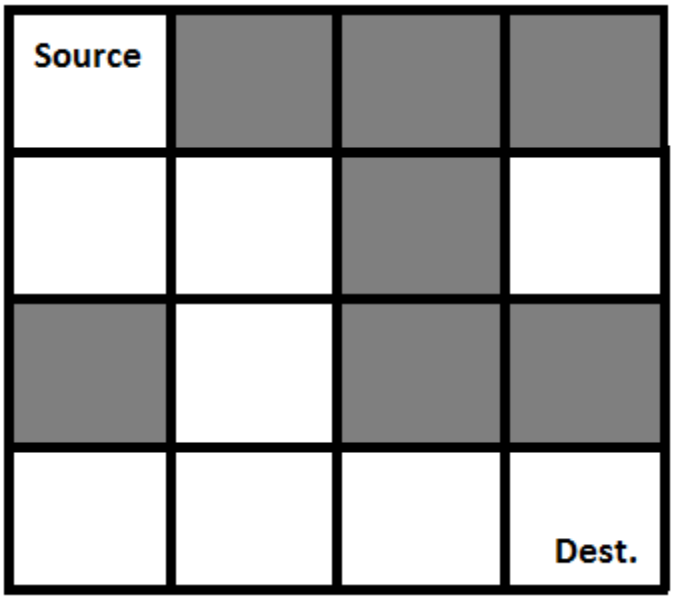
Now the problem is transformed into a Shortest path between two nodes in a directed graph problem. We represent the snake and ladder board using a map.

Lab no 13

Rat in a Maze

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward anddown.
In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

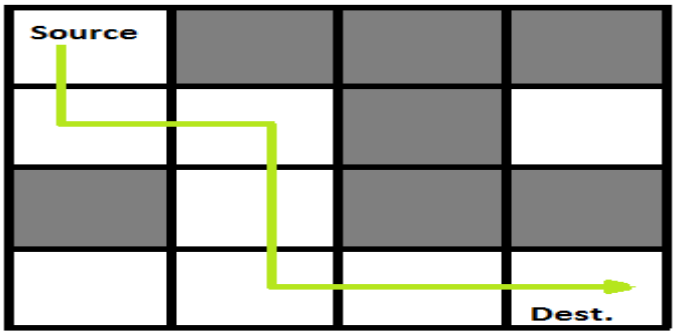
Following is an example maze.
Gray blocks are dead ends (value = 0).



Following is binary matrix representation of the above maze.

{1, 0, 0, 0}
{1, 1, 0, 1}
{0, 1, 0, 0}
{1, 1, 1, 1}

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrx.
{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}
All eateries in solution path are marked as 1.

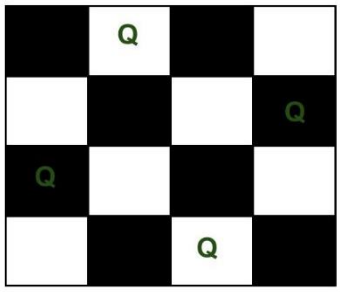
EXERCISE:

1. What approach do you used to solve problem and why?
2. Give the algorithm of above problem , Also analyze the time & space complexity of your algorithm.
3. Give implementation of Rat in a Maze problem.

Lab no 14

N- Queen Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}

EXERCISE:

1. What approach do you used to solve problem and why?
2. Give the algorithm of above problem , Also analyze the time & space complexity of your algorithm.
3. Give implementation of 8 Queen Problem.