UNIT I

INTRODUCTION TO DATA STRUCTURES

Introduction – Basic terminology – Data structures – Data structure operations - ADT – Algorithms: Complexity, Time – Space trade off - Mathematical notations and functions - Asymptotic notations – Linear and Binary search - Bubble sort - Insertion sort

INTRODUCTION

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way.

Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of String data type and 26 is of integer data type.

We can organize this data as a record like Player record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33.

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

Basic Terminology of Data Organization:

Data : The term 'DATA' simply referes to a a value or a set of values. These values may present anything about something, like it may be roll no of a student, marks, name of an employee, address of person etc.

Data item : A data item refers to a single unit of value. For eg. roll no of a student, marks, name of an employee, address of person etc. are data items. Data items that can be divided into sub items are called **group items** (Eg. Address, date, name), where as those who can not be divided in to sub items are called **elementary items** (Eg. Roll no, marks, city, pin code etc.).

Entity - with similar attributes (e.g all employees of an organization) form an entity set

Information: processed data, Data with given attribute

Field is a single elementary unit of information representing an attribute of an entity

Record is the collection of field values of a given entity

File is the collection of records of the entities in a given entity set

Name	**Age**	**Sex**	**Roll Number**	**Branch**
A	**17**	**M**	**109cs0132**	**CSE**
B	**18**	**M**	**109ee1234**	**EE**

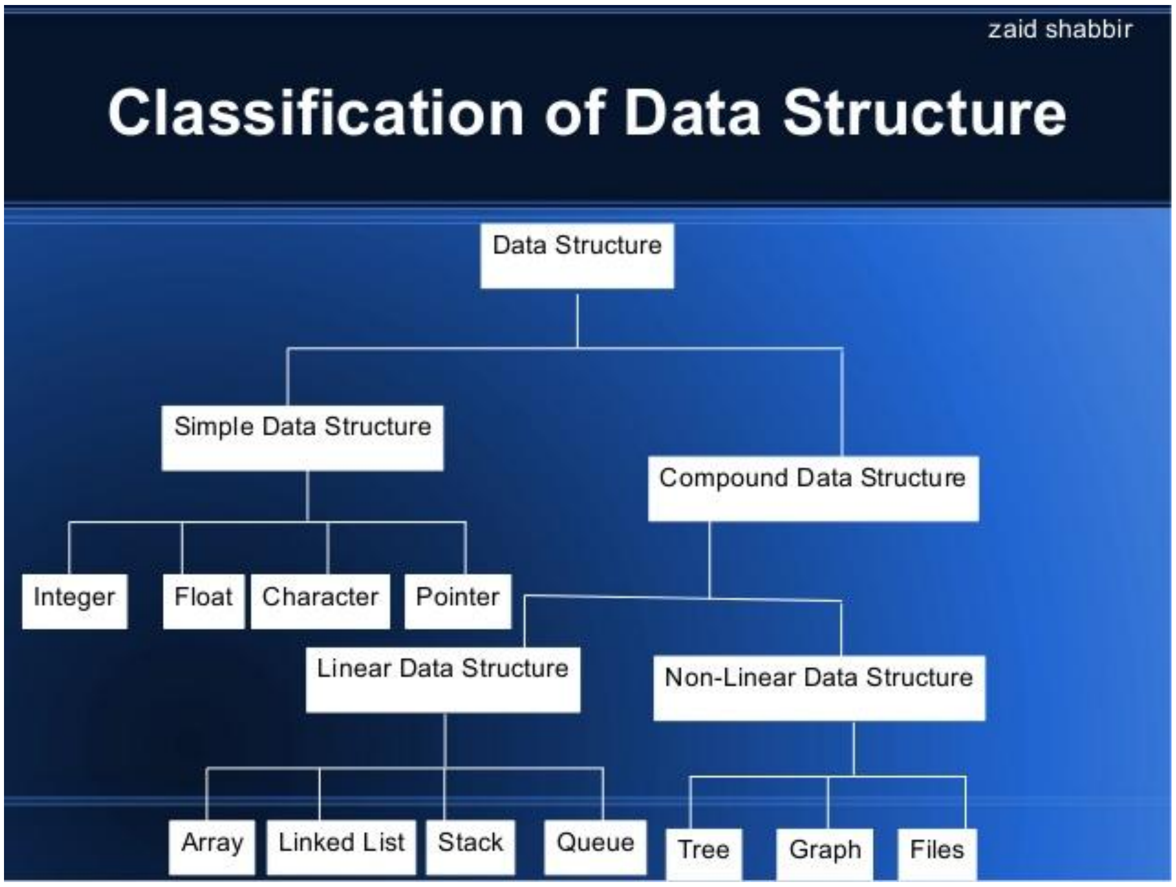
Basic types of Data Structures

Anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

- Array
- Linked List
- Stack
- Queue
- Tree
- Graph

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required.



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

(1) **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.

(2) **Searching:** Finding the location of a particular record with a given key value, or finding the location of all records which satisfy one or more conditions.

(3) **Inserting:** Adding a new record to the structure.

(4) **Deleting:** Removing the record from the structure.

(5) **Sorting:** Managing the data or record in some logical order (Ascending or descending order).

(6) **Merging:** Combining the record in two different sorted files into a single sorted file.

Abstract Data Types (ADT)

An abstract data type (ADT) refers to a set of data values and associated operations that are specified accurately, independent of any particular implementation. With an ADT, we know what a specific data type can do, but how it actually does it is hidden. Simply hiding the implementation

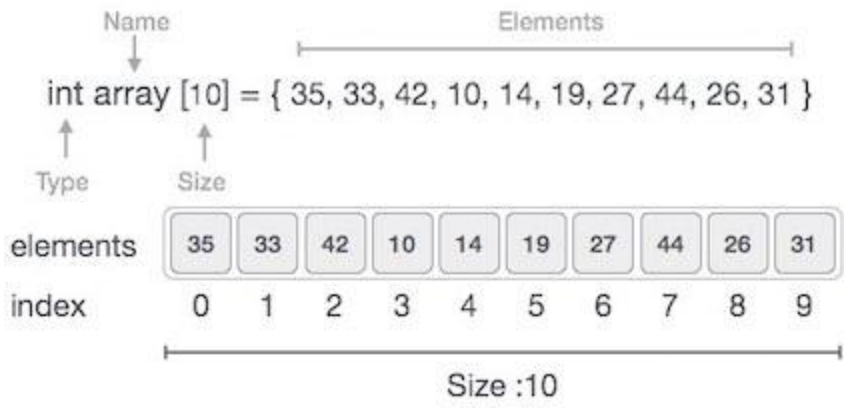
Data Structure - Arrays

Array is a container which can hold fix number of items and these items should be of same type. Most of the data structures make use of array to implement their algorithms. Following are important terms to understand the concepts 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 above shown 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 element at index 6 as 27.

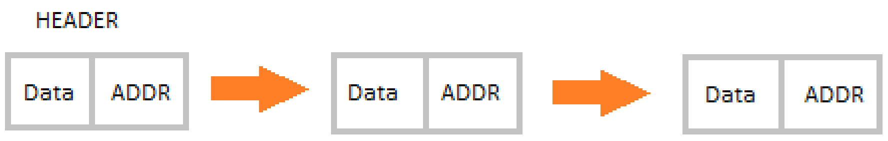
Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – add an element at given index.
- **Deletion** – delete an element at given index.
- **Search** – search an element using given index or by value.
- **Update** – update an element at given index.

Data Structure – Linked Lists

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

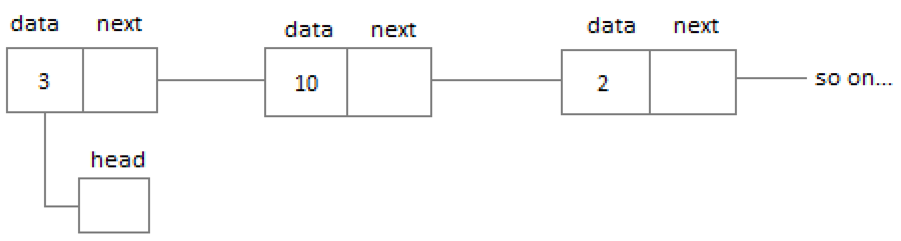
Linked lists are used to implement stacks, queues, graphs, etc.

Linked lists let you insert elements at the beginning and end of the list.

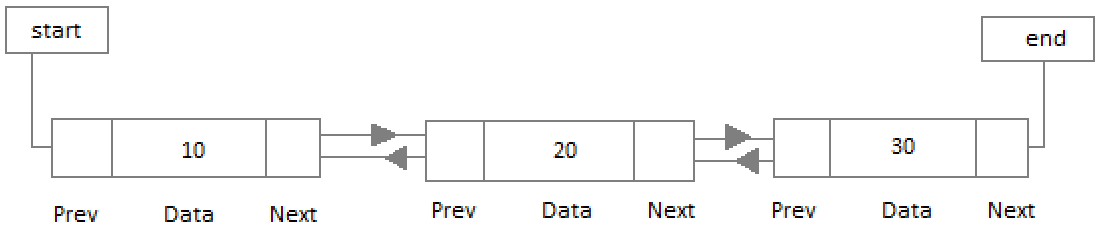
In Linked Lists we don’t need to know the size in advance.

Types of Linked Lists

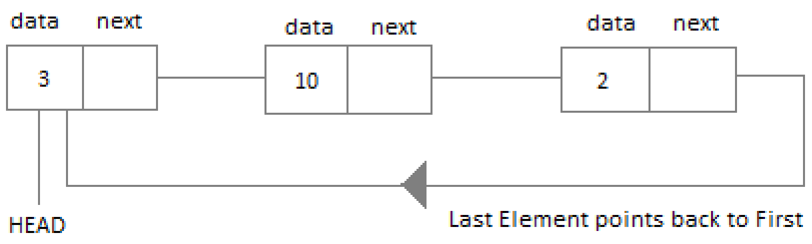
Singly Linked List : Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



Doubly Linked List : In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



Circular Linked List : In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.

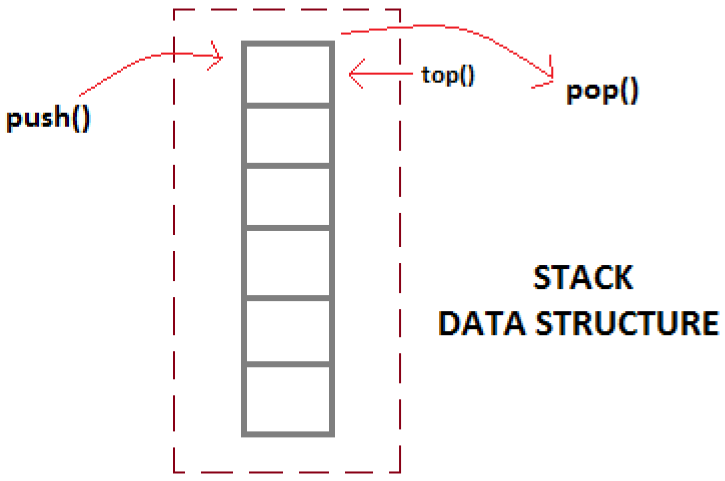


Data Structure – Stack

Stacks

Stack is an abstract data type with a bounded (predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.

Stack Data Structure



Basic features of Stack

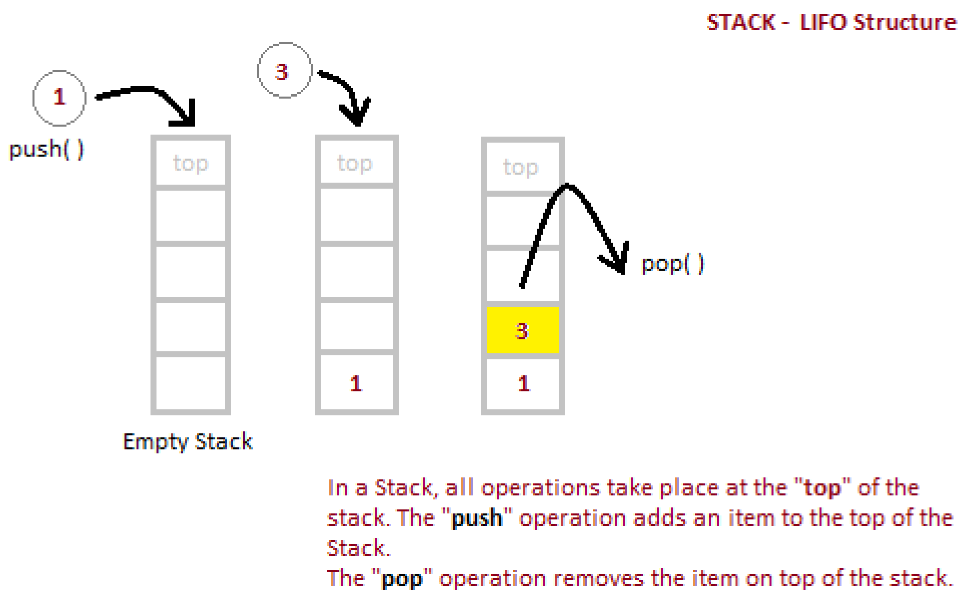
1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Applications of Stack

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- There are other uses also like : Parsing, Expression Conversion(Infix to Postfix, Postfix to Prefix etc) and many more.

Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.


STACK - LIFO Structure
1
push( )
top
Empty Stack
3
top
1
top
3
1
pop( )
In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.
The "pop" operation removes the item on top of the stack.

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Analysis of Stacks

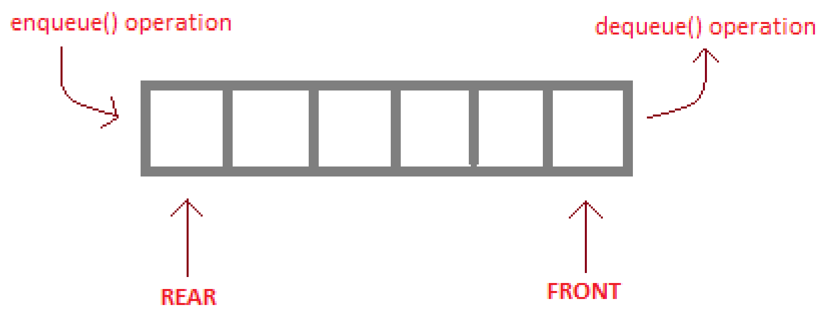
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- Push Operation : O(1)
- Pop Operation : O(1)
- Top Operation : O(1)
- Search Operation : O(n)

Queue Data Structures

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called REAR(also called tail), and the deletion of exisiting element takes place from the other end called as FRONT(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue.



enqueue( ) is the operation for adding an element into Queue.
dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO(First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek( ) function is oftenly used to return the value of first element without dequeuing it.

Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

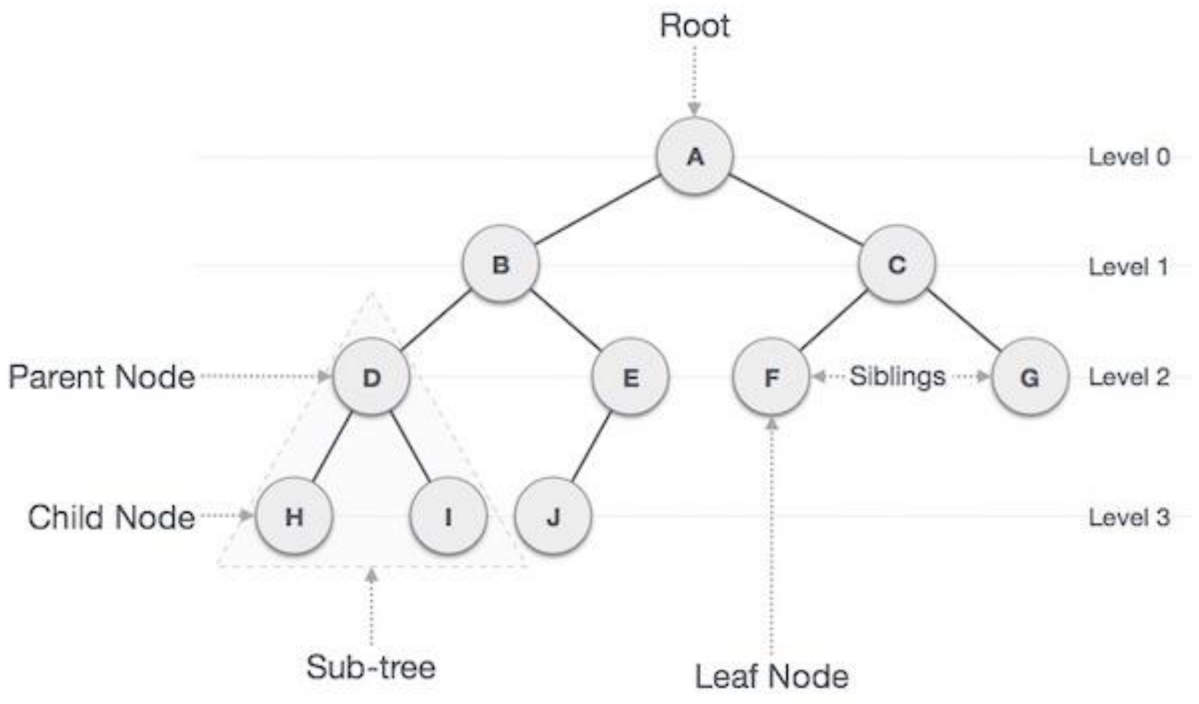
Analysis of Queue

- Enqueue : O(1)
- Dequeue : O(1)
- Size : O(1)

Data Structure - Tree

Tree represents nodes connected by edges. We'll going to discuss binary tree or binary search tree specifically.

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



Terms

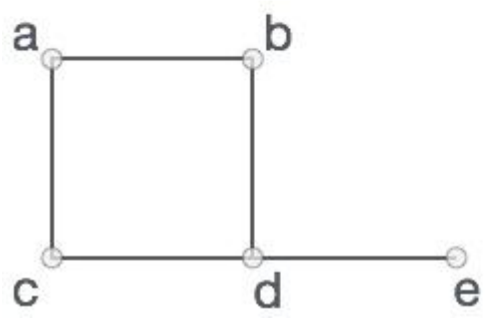
Following are important terms with respect to tree.

- **Path** – Path refers to sequence of nodes along the edges of a tree.
- **Root** – Node at the top of the tree is called root. There is only one root per tree and one path from root node to any node.
- **Parent** – Any node except root node has one edge upward to a node called parent.
- **Child** – Node below a given node connected by its edge downward is called its child node.
- **Leaf** – Node which does not have any child node is called leaf node.
- **Subtree** – Subtree represents descendents of a node.
- **Visiting** – Visiting refers to checking 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 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.

Data Structure - Graph

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}

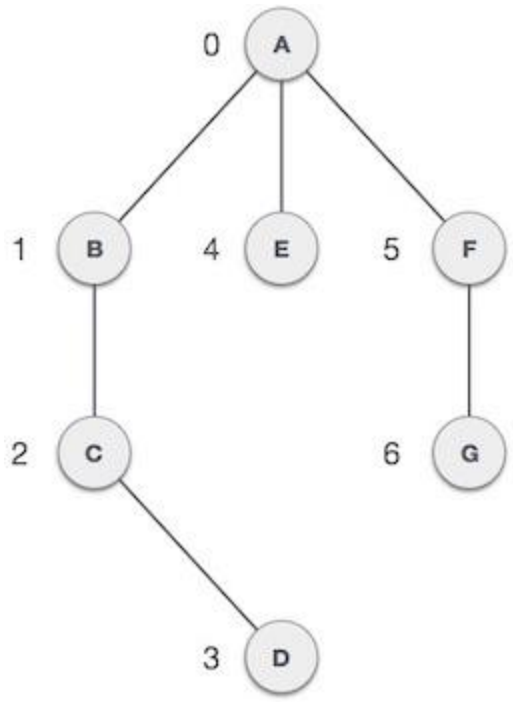
E = {ab, ac, bd, cd, de}

Graph Data Structure

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 −

- **Vertex** − Each node of the graph is represented as a vertex. In example given below, labeled circle represents vertices. So A to G are vertices. We can represent them using an array as shown in image below. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In example given below, lines from A to B, B to C and so on represents edges. We can use a two dimensional array to represent array as shown in image below. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In example given below, B is adjacent to A, C is adjacent to B and so on.
- **Path** – Path represents a sequence of edges between two vertices. In example given below, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph which are following.

- **Add Vertex** – add a vertex to a graph.
- **Add Edge** – add an edge between two vertices of a graph.
- **Display Vertex** – display a vertex of a graph.

Algorithms Basics

Algorithm is a step by step procedure, which defines a set of instructions to be executed in 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.

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high level description as pseudo code or using a flowchart.

From 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 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 below mentioned characteristics −

- Unambiguous − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/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.

Algorithm Analysis

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. **Time Complexity**
2. **Space Complexity**

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor –** The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor –** The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Space required by an algorithm is equal to the sum of the following two components –

- **A fixed part** that is a space required to store certain data and variables that are independent of the size of the problem. For example simple variables & constant used and program size etc.
- **A variable part** is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stacks space etc.

An algorithm generally requires space for following components:

- **Instruction Space:** It is the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** It is the space required to store all the constants and variables value.
- **Environment Space:** It is the space required to store the environment information needed to resume the suspended function.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I) Where C is the fixed part and S(I) is the variable part of the algorithm which depends on instance characteristic I. Following is a simple example that tries to explain the concept –

Asymptotic Notations

The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

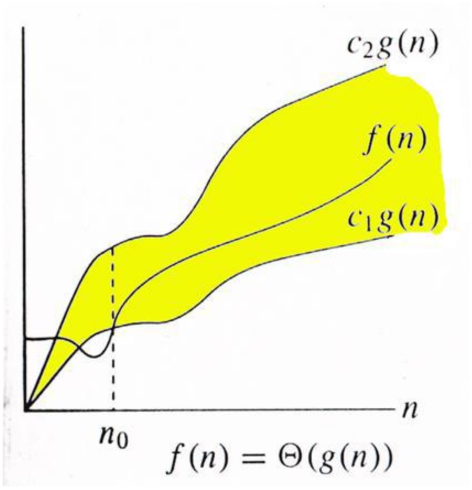
1) Θ Notation:

The theta notation bounds a function from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression. $3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ beats $\Theta(n^2)$ irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

$$\Theta((g(n)) = \{f(n): \textit{there exist positive constants } c1, c2 \textit{ and } n0 \textit{ such that}$$

$$0 <= c1 * g(n) <= f(n) <= c2 * g(n) \textit{ for all } n >= n0\}$$



The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.

2. Big O Notation:

The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time. If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

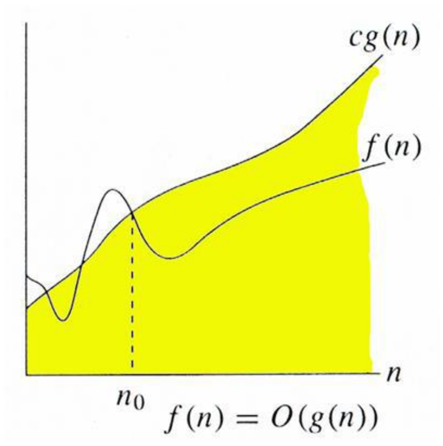
1. The worst case time complexity of Insertion Sort is Θ(n^2).

2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

$$O(g(n)) = \{ f(n): \textit{there exist positive constants } c \textit{ and } n0 \textit{ such that}$$

$$0 <= f(n) <= cg(n) \textit{ for all } n >= n0\}$$

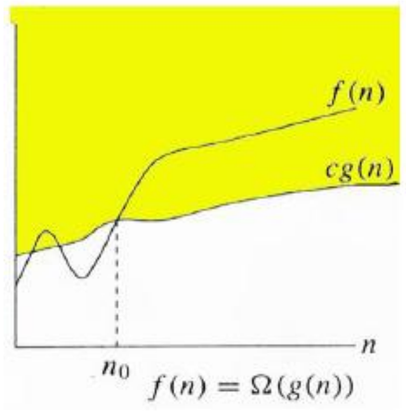


3) Ω Notation:

Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound. Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful; the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

$$\Omega\ (g(n))\ =\ \{f(n){:}\ there\ exist\ positive\ constants\ c\ and\ n0\ such\ that$$
$$0\ <=\ cg(n)\ <=\ f(n)\ for\ all\ n\ >=\ n0\}.$$



Mathematical Analysis of Recursive Algorithms

Many algorithms are recursive in nature. When we analyze them, we get a recurrence relation for time complexity. We get running time on an input of size n as a function of n and the running time on inputs of smaller sizes. For example in Merge Sort, to sort a given array, we divide it in two halves and recursively repeat the process for the two halves. Finally we merge the results. Time complexity of Merge Sort can be written as T(n) = 2T(n/2) + cn. There are many other algorithms like Binary Search, Tower of Hanoi, etc.
There are mainly three ways for solving recurrences.

1) Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.
For example consider the recurrence T(n) = 2T(n/2) + n
We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.
We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

T(n) = 2T(n/2) + n
<= cn/2Log(n/2) + n
<= cnLogn - cnLog2 + n
<= cnLogn - cn + n
<= cnLogn

2) Recurrence Tree Method:
In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series. For example consider the recurrence relation
T(n) = T(n/4) + T(n/2) + cn^2

```
        cn^2
      /      \
  T(n/4)     T(n/2)
```

If we further break down the expression T(n/4) and T(n/2), we get following recursion tree.

```
              cn^2
          /           \
     c(n^2)/16      c(n^2)/4
      /      \        /     \
 T(n/16)    T(n/8)  T(n/8)    T(n/4)
```

Breaking down further gives us following

```
               cn^2
           /            \
     c(n^2)/16          c(n^2)/4
      /       \          /      \
c(n^2)/256  c(n^2)/64  c(n^2)/64    c(n^2)/16
 /    \      /    \    /    \       /    \
```

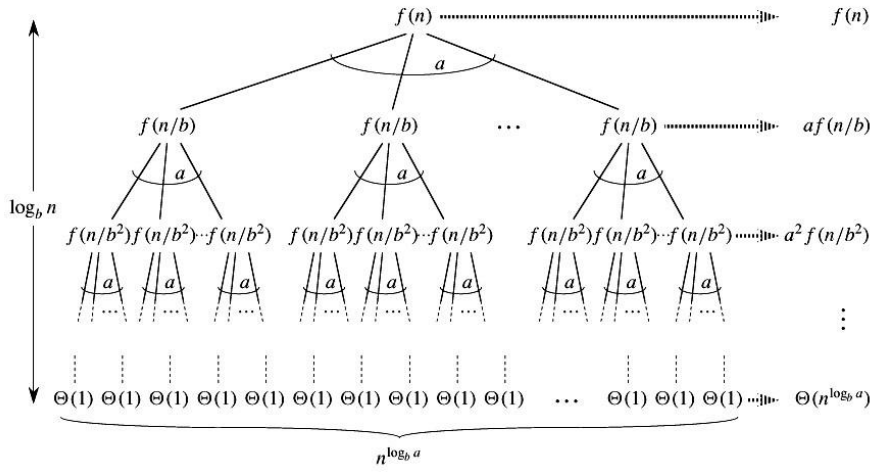
To know the value of T(n), we need to calculate sum of tree nodes level by level. If we sum the above tree level by level, we get the following series
T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + ...
The above series is geometrical progression with ratio 5/16. To get an upper bound, we can sum the infinite series. We get the sum as $(n^2)/(1 - 5/16)$ which is $O(n^2)$

3) Master Method:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
There are following three cases:

1. If $f(n) = \Theta(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$
2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$
3. If $f(n) = \Theta(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

How does this work?
Master method is mainly derived from recurrence tree method. If we draw recurrence tree of T(n) = aT(n/b) + f(n), we can see that the work done at root is f(n) and work done at all leaves is $\Theta(n^c)$ where c is $\text{Log}_b a$. And the height of recurrence tree is $\text{Log}_b n$



In recurrence tree method, we calculate total work done. If the work done at leaves is polynomially more, then leaves are the dominant part, and our result becomes the work done at leaves (Case 1). If work done at leaves and root is asymptotically same, then our result becomes height multiplied by work done at any level (Case 2). If work done at root is asymptotically more, then our result becomes work done at root (Case3).

Examples of some standard algorithms whose time complexity can be evaluated using Master Method

Merge Sort: T(n) = 2T(n/2) + Θ(n). It falls in case 2 as c is 1 and $\text{Log}_b a$] is also 1. So the solution is Θ(n Logn)
Binary Search: T(n) = T(n/2) + Θ(1). It also falls in case 2 as c is 0 and $\text{Log}_b a$ is also 0. So the solution is Θ(Logn)

Notes:

1) It is not necessary that a recurrence of the form $T(n) = aT(n/b) + f(n)$ can be solved using Master Theorem. The given three cases have some gaps between them. For example, the recurrence $T(n) = 2T(n/2) + n/\text{Log}n$ cannot be solved using master method.

2) Case 2 can be extended for $f(n) = \Theta(n^c \text{Log}^k n)$

If $f(n) = \Theta(n^c \text{Log}^k n)$ for some constant $k >= 0$ and $c = \text{Log}_b a$, then $T(n) = \Theta(n^c \text{Log}^{k+1} n)$

Practice Problems and Solutions on Master Theorem.(Refer Notes)

References:

http://en.wikipedia.org/wiki/Master_theorem
MIT Video Lecture on Asymptotic Notation | Recurrences | Substitution, Master Method
Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Mathematical Analysis of Nonrecursive Algorithms

The core of the algorithm analysis: to find out how the number of the basic operations depends on the size of the input.

There are four rules to count the operations:

Rule 1: for loops - the size of the loop times the running time of the body

The running time of a for loop is at most the running time of the statements inside the loop times the number of iterations.

```
        for( i = 0; i < n; i++)
                sum = sum + i;
```

a. Find the running time of statements when executed only once:

The statements in the loop heading have fixed number of operations, hence they have constant running time O(1) when executed only once. The statement in the loop body has fixed number of operations, hence it has a constant running time when executed only once.

b. Find how many times each statement is executed.

```
 for( i = 0; i < n; i++)          // i = 0; executed only once: O(1)
                                  // i < n; n + 1 times O(n)
                                 // i++ n times O(n)
                                 // total time of the loop heading:
                                 // O(1) + O(n) + O(n) = O(n)
        sum = sum + i;           // executed n times, O(n)
```

The loop heading plus the loop body will give: O(n) + O(n) = O(n).

Loop running time is: O(n)

Mathematical analysis of how many times the statements in the body are executed

$$C(n) = \sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

If

a. the size of the loop is n (loop variable runs from 0, or some fixed constant, to n) and

b. the body has constant running time (no nested loops)

then the time is O(n)

Rule 2: Nested loops – the product of the size of the loops times the running time of the body

The total running time is the running time of the inside statements times the product of the sizes of all the loops

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = 0; j < n; j++)
                sum++;
```

Applying Rule 1 for the nested loop (the 'j' loop) we get O(n) for the body of the outer loop. The outer loop runs n times, therefore the total time for the nested loops will be $O(n) * O(n) = O(n*n) = O(n^2)$

Mathematical analysis:

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

Inner loop:

$$S(i) = \sum_{j=0}^{n-1} 1 = (n-1) - 0 + 1 = n$$

Outer loop:

$$C(n) = \sum_{i=0}^{n-1} S(i) = \sum_{i=0}^{n-1} n = n * \sum_{i=0}^{n-1} 1 = n((n-1) - 0 + 1) = n^2$$

What happens if the inner loop does not start from 0?

```
sum = 0;
for( i = 0; i < n; i++)
        for( j = i; j < n; j++)
                sum++;
```

Here, the number of the times the inner loop is executed depends on the value of i

i = 0, inner loop runs n times

i = 1, inner loop runs (n-1) times

i = 2, inner loop runs (n-2) times

…

i = n – 2, inner loop runs 2 times

i = n – 1, inner loop runs once.

Thus we get: (1 + 2 + … + n) = $n*(n+1)/2 = O(n^2)$

General rule for nested loops:

Running time is the product of the size of the loops times the running time of the body.

Example:

```
sum = 0;
for( i = 0; i < n; i++)
         for( j = 0; j < 2n; j++)
                sum++;
```

We have one operation inside the loops, and the product of the sizes is $2n^2$

Hence the running time is $O(2n^2) = O(n^2)$

Note: if the body contains a function call, its running time has to be taken into consideration.

```
sum = 0;
        for( i = 0; i < n; i++)
                for( j = 0; j < n; j++)
                        sum = sum + function(sum);
```

Assume that the running time of function(sum) is known to be log(n).
Then the total running time will be O(n^2*log(n))

Rule 3: Consecutive program fragments

The total running time is the maximum of the running time of the individual fragments

```
sum = 0;
        for( i = 0; i < n; i++)
                sum = sum + i;
sum = 0;
for( i = 0; i < n; i++)
         for( j = 0; j < 2*n; j++)
                sum++;
```

The first loop runs in O(n) time, the second - O(n^2) time, the maximum is O(n^2)

Rule 4: If statement

```
if C
        S1;
else
        S2;
```

The running time is the maximum of the running times of S1 and S2.

Summary

Steps in analysis of nonrecursive algorithms:

- Decide on parameter n indicating input size
- Identify algorithm's basic operation
- Check whether the number of time the basic operation is executed depends on some additional property of the input. If so, determine worst, average, and best case for input of size n
- Count the number of operations using the rules above.

Exercise

a.

```
sum = 0;
for( i = 0; i < n; i++)
for( j = 0; j < n * n; j++)
sum++;
```

b.

```
sum = 0;
for( i = 0; i < n; i++)
for( j = 0; j < i; j++)
sum++;
```

c.

```
sum = 0;
for( i = 0; i < n; i++)
```

```
        for( j = 0; j < i*i; j++)
        for( k = 0; k < j; k++)
        sum++;
```

d.

```
        sum = 0;
        for( i = 0; i < n; i++)
        sum++;
        val = 1;
        for( j = 0; j < n*n; j++)
        val = val * j;
```

e.

```
        sum = 0;
        for( i = 0; i < n; i++)
        sum++;
        for( j = 0; j < n*n; j++)
        compute_val(sum,j);
```

The complexity of the function compute_val(x,y) is given to be O(nlogn)

Solutions:

(a) $O(n^3)$

(b) $O(n^2)$

(c) $O(n^5)$

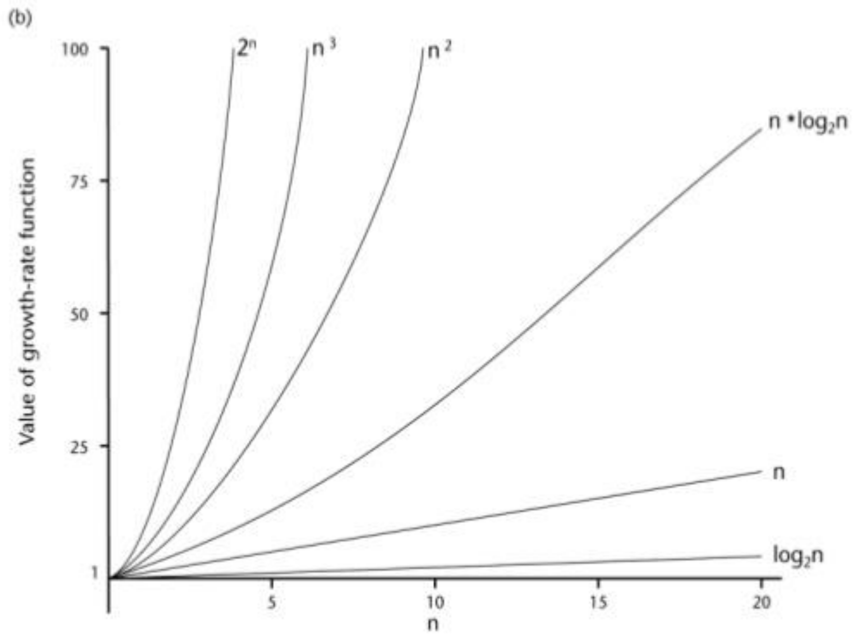
(d) $O(n^2)$

(e)$O(n^3 \log(n))$)

Order of Growth Functions

	constant	*logarithmic*	*linear*	*N-log-N*	*quadratic*	*cubic*	*exponential*
n	**O(1)**	**O(log *n*)**	**O(*n*)**	**O(*n* log *n*)**	**O(n^2)**	**O(n^3)**	**O(2^n)**
1	1	1	1	1	1	1	2
2	1	1	2	2	4	8	4
4	1	2	4	8	16	64	16
8	1	3	8	24	64	512	256
16	1	4	16	64	256	4,096	65536
32	1	5	32	160	1,024	32,768	4,294,967,296
64	1	6	64	384	4,069	262,144	1.84×10^{19}

A Comparison of Growth-Rate Functions (cont.)



CENG 213 Data Structures 21

Linear and Binary search

An algorithm is a step-by-step procedure or method for solving a problem by a computer in a given number of steps. The steps of an algorithm may include repetition depending upon the problem for which the algorithm is being developed. The algorithm is written in human readable and understandable form. To search an element in a given array, it can be done in two ways linear search and Binary search.

Linear Search

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

Pseudocode:-

```
# Input: Array D, integer key
# Output: first index of key in D, or -1 if not found
 For i = 0 to last index of D:
   if D[i] == key:
     return i
 return -1
```

Example with Implementation

To search the element 5 it will go step by step in a sequence order.

```
linear(a[n], key)
   for( i = 0; i < n; i++)
        if (a[i] == key)
            return i;
 return -1;
```

Asymptotic Analysis

Worst Case Analysis (Usually Done)

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (target in the above code) is not present in the array. When target is not present, the search() functions compares it with all the elements of array one by one. Therefore, the worst case time complexity of linear search would be Θ(n).

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of target not being present in array).

The key is equally likely to be in any position in the array

If the key is in the first array position: 1 comparison
If the key is in the second array position: 2 comparisons
...
If the key is in the ith postion: i comparisons
...

So average all these possibilities: (1+2+3+...+n)/n = [n(n+1)/2] /n = (n+1)/2 comparisons. The average number of comparisons is **(n+1)/2 = Θ(n).**

Best Case Analysis (Bogus)

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when Target is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1)

Binary Search

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

To search an element 13 from the sorted array or list.

```
binarysearch(a[n], key, low, high)
while(low<high)
{
mid = (low+high)/2;
if(a[mid]=key)
        return mid;
elseif (a[mid] > key)
        high=mid-1;
  else
        low=mid+1;
 }
return -1;
```

In the above program logic, we are first comparing the middle number of the list, with the target, if it matches we return. If it doesn't, we see whether the middle number is greater than or smaller than the target.

If the Middle number is greater than the Target, we start the binary search again, but this time on the left half of the list, that is from the start of the list to the middle, not beyond that.

If the Middle number is smaller than the Target, we start the binary search again, but on the right half of the list, that is from the middle of the list to the end of the list.

Complexity Analysis

Worst case analysis: The key is not in the array

Let T(n) be the number of comparisons done in the worst case for an array of size n. For the purposes of analysis, assume n is a power of 2, ie n = 2^k.

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

$= 2 + 2 + T(\frac{n}{2^2})$ // 2nd iteration

$= 2 + 2 + 2 + T(n/2^3)$ // 3rd iteration

...

$= i * 2 + T(n/2^i)$ // ith iteration

... $= k * 2 + T(1)$

Note that k = logn, and that T(1) = 2.

So T(n) = 2logn + 2 = O(logn)

So we expect binary search to be significantly more efficient than linear search for large values of n.

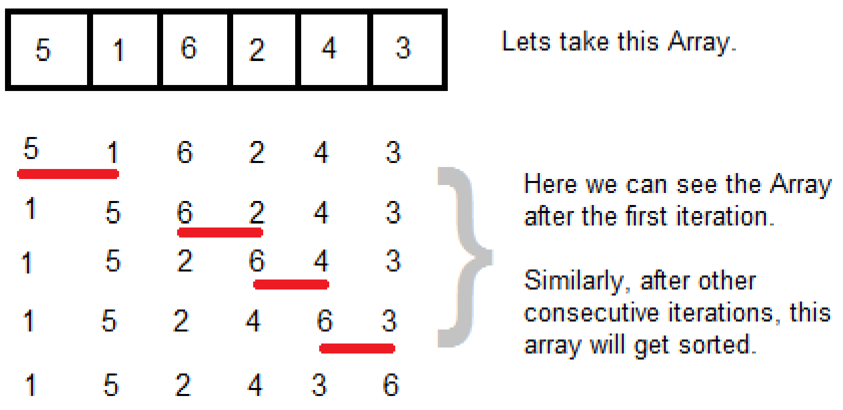
Bubble Sort

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Bubble Sort for Data Structures


Lets take this Array.
Here we can see the Array after the first iteration.
Similarly, after other consecutive iterations, this array will get sorted.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 2 6 4 3
1 5 2 4 6 3
1 5 2 4 3 6

Sorting using Bubble Sort Algorithm

Let's consider an array with values {5, 1, 6, 2, 4, 3}

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
 for(j=0; j<6-i-1; j++)
 {
  if( a[j] > a[j+1])
  {
   temp = a[j];
   a[j] = a[j+1];
   a[j+1] = temp;
  }
 }
}
//now you can print the sorted array after this
```

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and wew can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6; i++)
{
  int flag = 0;       //taking a flag variable
  for(j=0; j<6-i-1; j++)
  {
    if( a[j] > a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = temp;
      flag = 1;       //setting flag as 1, if swapping occurs
    }
  }
  if(!flag)           //breaking out of for loop if no swapping takes place
  {
    break;
  }
}
```

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

Complexity Analysis of Bubble Sorting

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

$(n-1)+(n-2)+(n-3)+.....+3+2+1$

Sum = $n(n-1)/2$

i.e $O(n^2)$

Hence the complexity of Bubble Sort is **$O(n^2)$**.

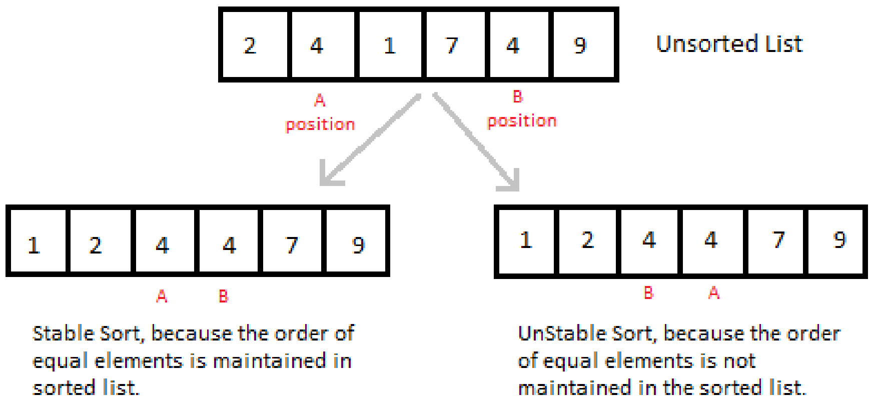
The main advantage of Bubble Sort is the simplicity of the algorithm. Space complexity for Bubble Sort is **O(1)**, because only single additional memory space is required for **temp** variable

Best-case Time Complexity will be **O(n)**, it is when the list is already sorted.

Insertion Sorting

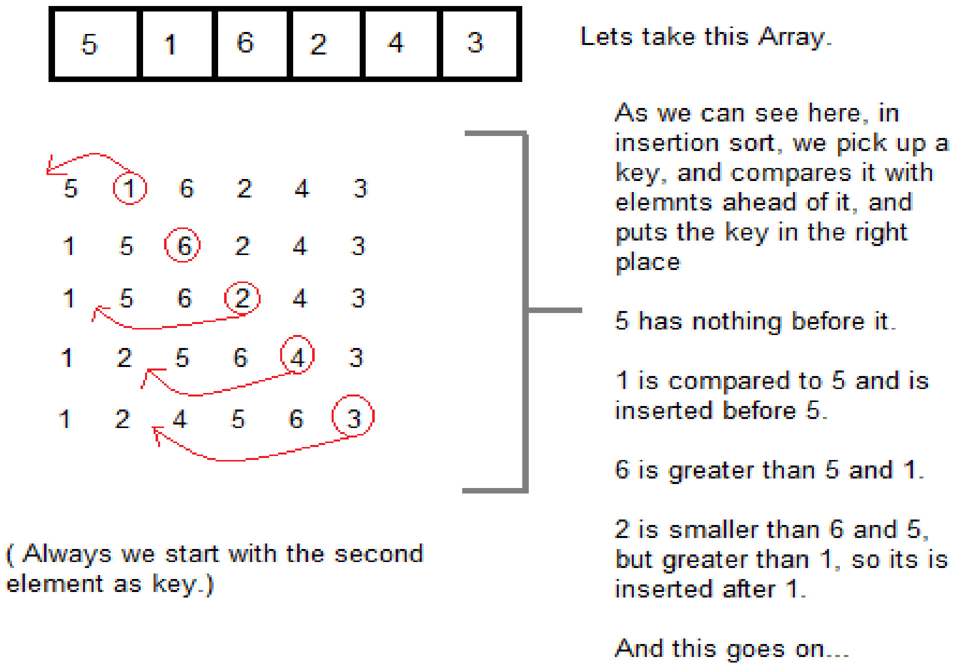
It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, inerstion sort also requires a single additional memory space.
6. It is Stable, as it does not change the relative order of elements with equal keys



2 4 1 7 4 9
Unsorted List
A position
B position
1 2 4 4 7 9
A B
Stable Sort, because the order of equal elements is maintained in sorted list.
1 2 4 4 7 9
B A
UnStable Sort, because the order of equal elements is not maintained in the sorted list.

How Insertion Sorting Works



5 1 6 2 4 3
Lets take this Array.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 6 2 4 3
1 2 5 6 4 3
1 2 4 5 6 3
As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place
5 has nothing before it.
1 is compared to 5 and is inserted before 5.
6 is greater than 5 and 1.
2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.
And this goes on...
(Always we start with the second element as key.)

Sorting using Insertion Sort Algorithm

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
 key = a[i];
 j = i-1;
 while(j>=0 && key < a[j])
 {
  a[j+1] = a[j];
  j--;
 }
 a[j+1] = key;
```

}

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable key, in which we put each element of the array, in each pass, starting from the second element, that is a[1].

Then using the while loop, we iterate, until j becomes equal to zero or we find an element which is greater than key, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

Complexity Analysis of Insertion Sorting

- Worst Case Time Complexity : O(n^2)
- Best Case Time Complexity : O(n)
- Average Time Complexity : O(n^2)
- Space Complexity : O(1)