

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 organizing 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 refers to 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 cannot 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

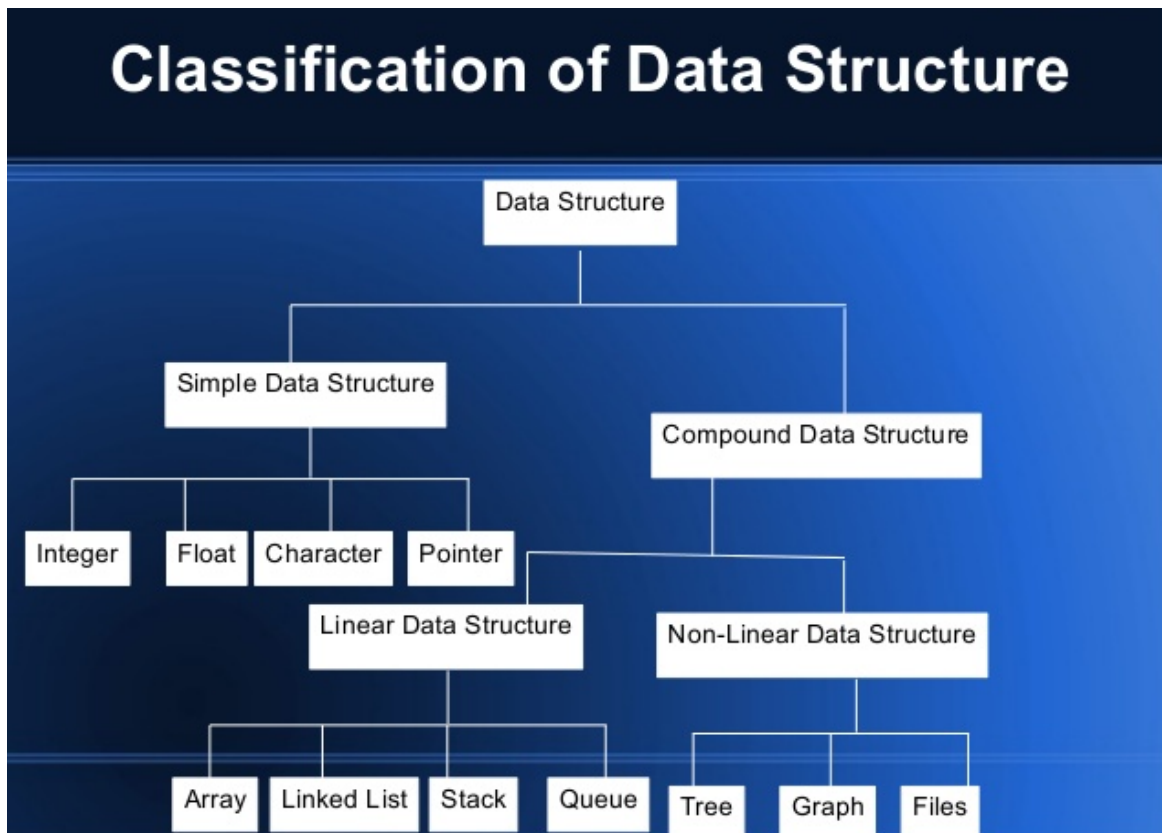
Basic types of Data Structures

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 examples 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

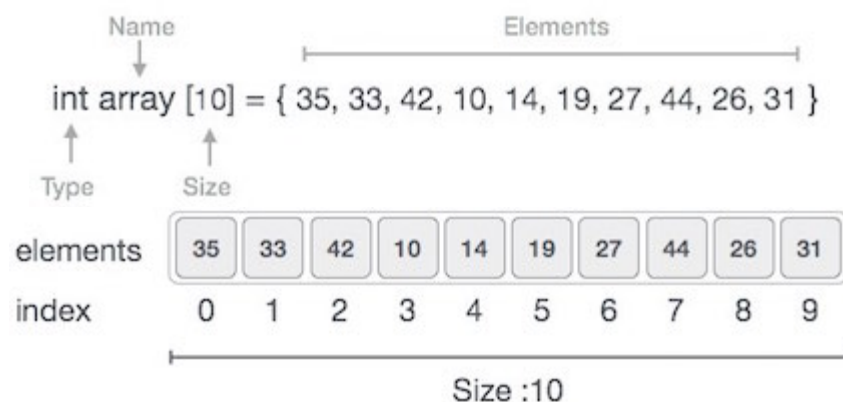
Data Structure - Arrays

Array is a container which can hold fix number of items and these items should be of same type. Most of the datastructures 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 mean 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

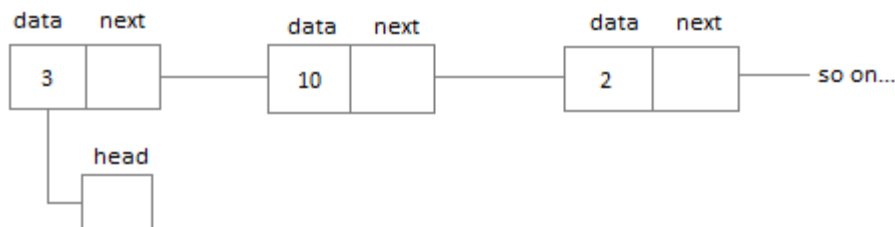
Linked lists are used to implement stacks, queues, graphs, etc.

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

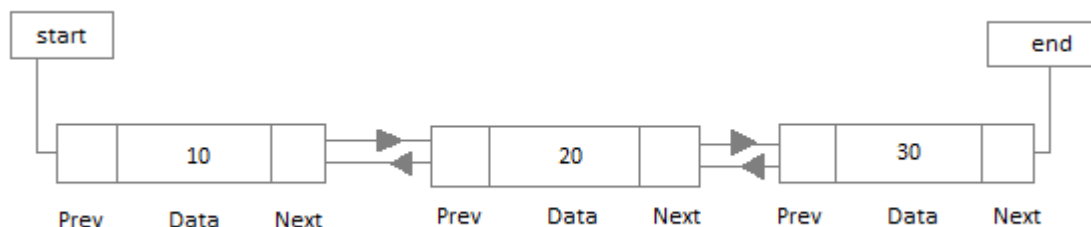
In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

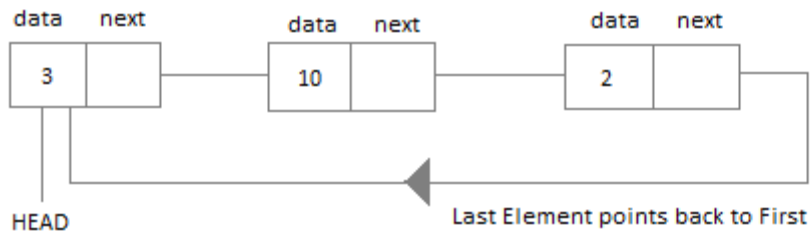
Singly Linked List : Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



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.

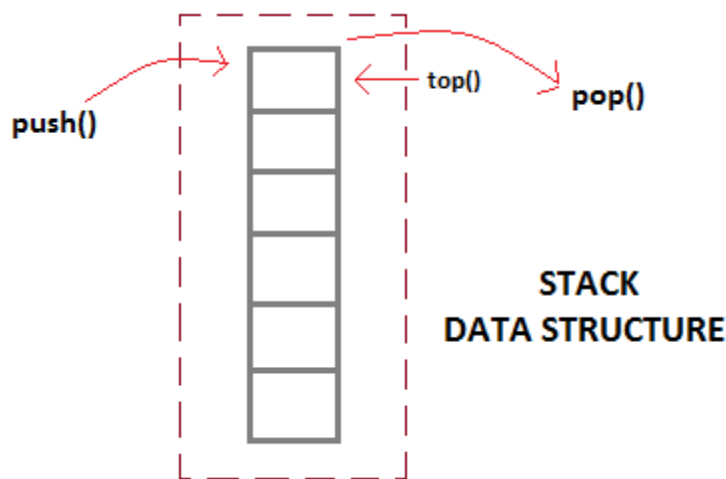


Data Structure – Stack

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

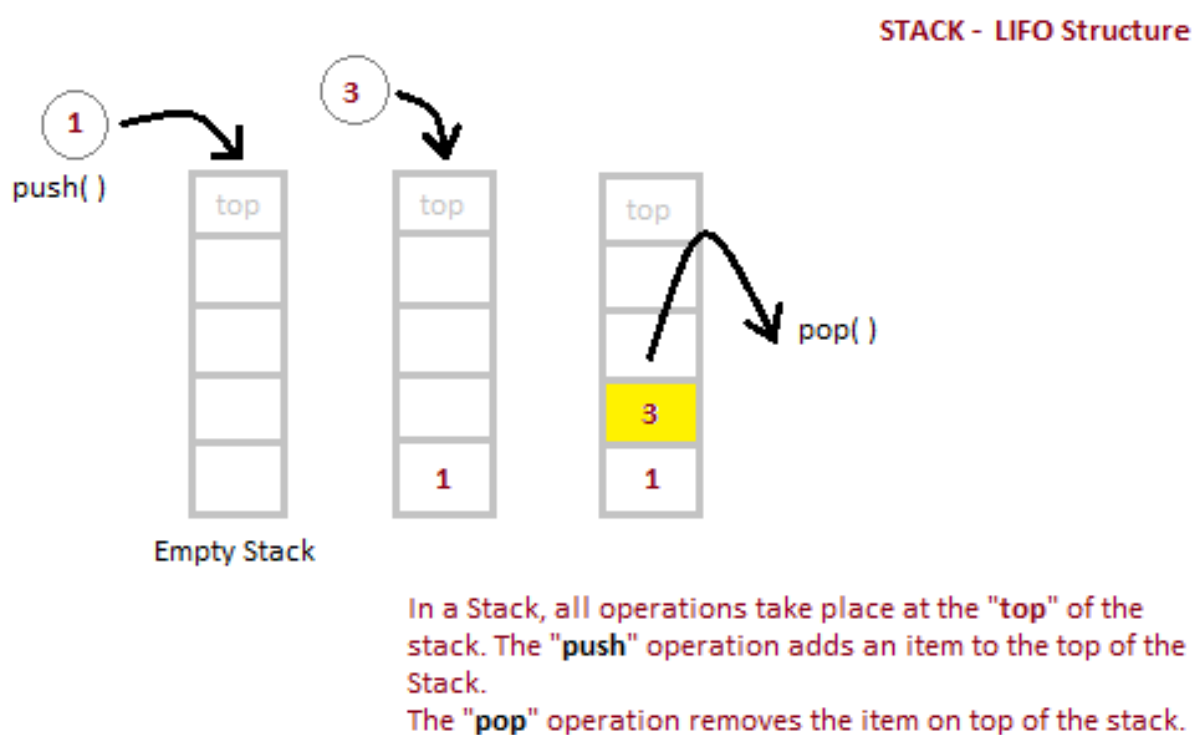
1. Stack is an ordered list of similar data type.
2. Stack is a LIFO structure. (Last in First out).
3. push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
4.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



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

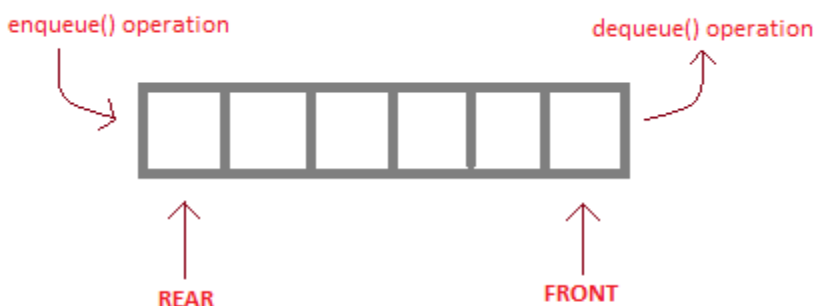
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- 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 existing 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.

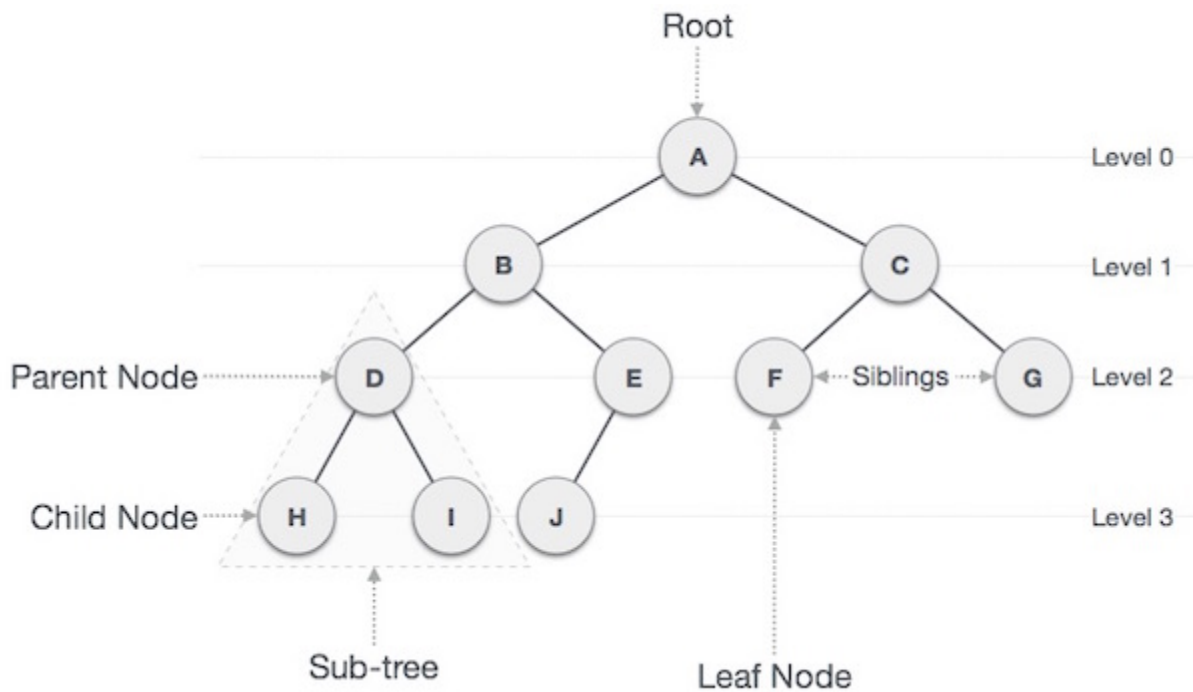
Analysis of Queue

- 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

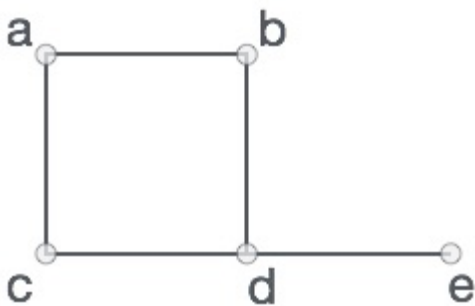
Following are important terms with respect to tree.

- **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 descendants 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,

$$V = \{a, b, c, d, e\}$$

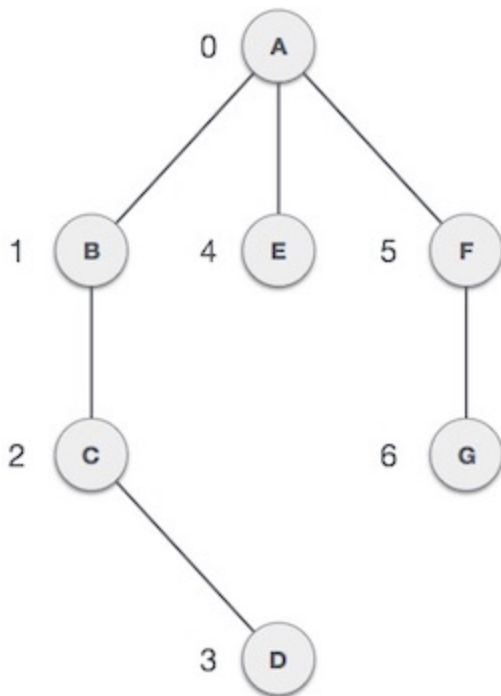
$$E = \{ab, ac, bd, cd, de\}$$

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 pseudocode 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 datastructure.
- Sort – Algorithm to sort items in certain order
- Insert – Algorithm to insert item in a datastructure
- 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. It's 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 –

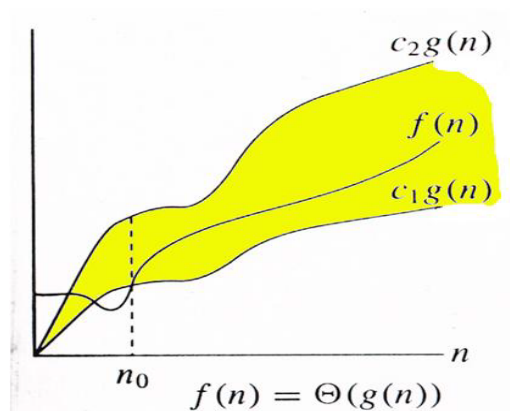
Asymptotic Notations

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.

Dropping lower order terms is always fine because there will always be a n_0 after which beats irrespective of the constants involved. For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.



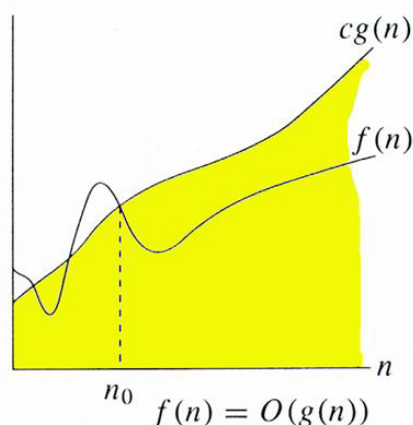
The above definition means, if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$ for large values of n ($n \geq n_0$). The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0 .

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 $\Theta(n^2)$.
2. The best case time complexity of Insertion Sort is $\Theta(n)$.

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.

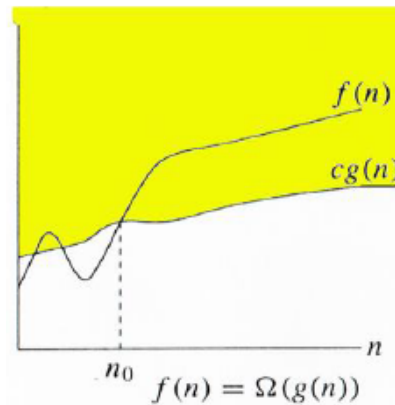


3) Ω Notation:

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 $\Omega(g(n))$ the set of functions.



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 $\Theta(n)$.

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 i th position: i comparisons

...

So average all these possibilities: $(1+2+3+\dots+n)/n = [n(n+1)/2] / n = (n+1)/2$ comparisons. The average number of comparisons is $(n+1)/2 = \Theta(n)$.

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 $\Theta(1)$

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

```
// 2nd iteration
```

```
// 3rd iteration
```

```
...
```

```
// ith iteration
```

```
...
```

Note that $k = \log n$, and that $T(1) = 2$.

So $T(n) = 2\log n + 2 = O(\log n)$

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

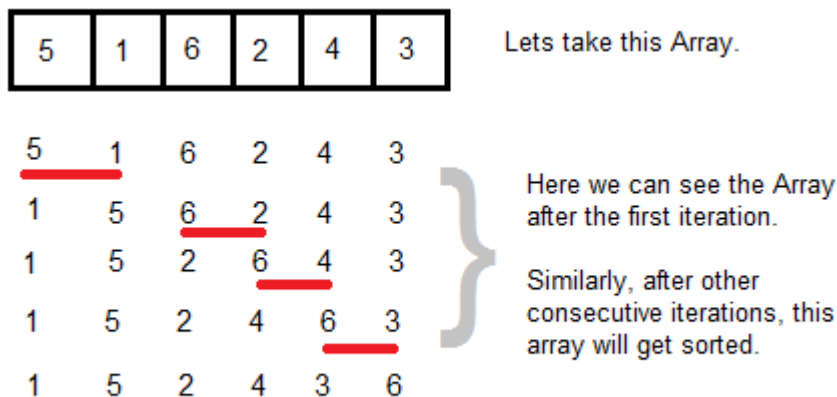
Bubble Sort

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



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 an 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 we 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

$$(n-1)+(n-2)+(n-3)+\dots+3+2+1$$

$$\text{Sum} = n(n-1)/2$$

$$\text{i.e } O(n^2)$$

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

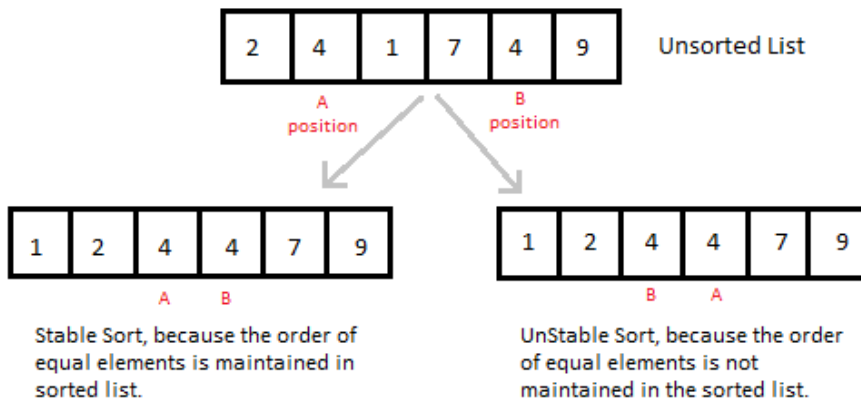
The main advantage of Bubble Sort is the simplicity of the algorithm.Space complexity for Bubble Sort is **$O(1)$** , because only single additional memory space is required for **temp** variable

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

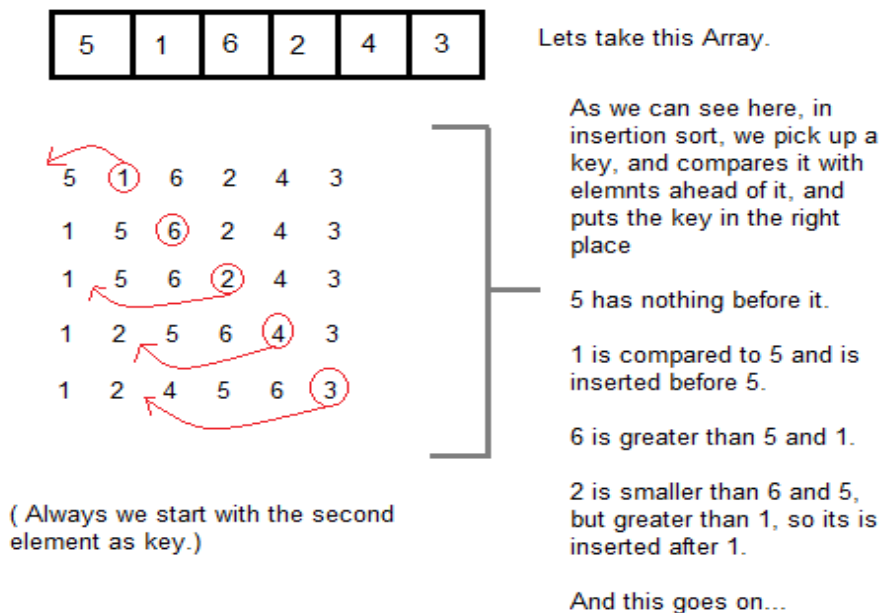
Insertion Sorting

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, insertion 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



How Insertion Sorting Works



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 let's 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 : $O(1)$