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.

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.

# 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 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.

Example Let's try to learn algorithm-writing by using an example.
Problem Design an algorithm to add two numbers and display 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 and b
step 3  c $\leftarrow a + b$
step 4  display c
step 5  STOP
```

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy of the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing step numbers, is optional.
We design an algorithm to get solution of a given problem. A problem can be solved in more than one ways.
one problem many solutions Hence, many solution algorithms can be derived for a given problem. Next step is to analyze those proposed solution algorithms and implement the best suitable.

# Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below

**A priori analysis** This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.

**A posterior analysis** This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

# Algorithm Analysis

We shall learn here a priori algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

# Algorithm 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. 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 and constant used, 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 stack space etc.
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

```
Algorithm: SUM(A, B)
Step 1 -  START
Step 2 -  C $\leftarrow A + B + 10$
Step 3 -  Stop
```

Here we have three variables A, B and C and one constant. Hence S(P) = 1+3. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

# Time Complexity

- Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.
- For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c*n, where c is the time taken for addition of two bits. Here, we observe that T(n) grows linearly as input size increases.

# Brute Force Search

- The brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

# Set Theory Revision: The Power Set

The Power Set is the set of all possible subsets of a set. If there are $n$ elements in the set, then there are $2^n$ elements in the power set.

$$\{A, B, C, D, E\}$$

# Combinatorial explosion

The main disadvantage of the brute-force method is that, for many real-world problems, the number of natural candidates is prohibitively large. For instance, if we look for the divisors of a number as described above, the number of candidates tested will be the given number n. So if n has sixteen decimal digits, say, the search will require executing at least 1015 computer instructions, which will take several days on a typical PC. If n is a random 64-bit natural number, which has about 19 decimal digits on the average, the search will take about 10 years. This steep growth in the number of candidates, as the size of the data increases, occurs in all sorts of problems.

For instance, if we are seeking a particular rearrangement of 10 letters, then we have 10! = 3,628,800 candidates to consider, which a typical PC can generate and test in less than one second. However, adding one more letter which is only a 10% increase in the data size will multiply the number of candidates by 11 a 1000% increase.
For 20 letters, the number of candidates is 20!, which is about 2.41018 or 2.4 quintillion; and the search will take about 10 years. This unwelcome phenomenon is commonly called the combinatorial explosion, or the curse of dimensionality.

# Data Structures - Asymptotic Analysis

- Asymptotic analysis of an algorithm, refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.
- Asymptotic analysis are input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is computed as f(n) and may be for another operation it is computed as g(n2). Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly the running time of both operations will be nearly same if n is significantly small.

Usually, 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. |

Asymptotic Notations Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

- Notation
- $\omega$ Notation
- $\theta$ Notation

# Big O notation

- Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
- It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann-Landau notation or asymptotic notation.

# Big O notation

- In computer science, big O notation is used to classify algorithms by how they respond to changes in input size, such as how the processing time of an algorithm changes as the problem size becomes extremely large.
- In analytic number theory it is used to estimate the "error committed" while replacing the asymptotic size of an arithmetical function by the value it takes at a large finite argument.
- A famous example is the problem of estimating the remainder term in the prime number theorem.

# Big O notation

- Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.
- The letter O is used because the growth rate of a function is also referred to as order of the function. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function.
- Associated with big O notation are several related notations, using the symbols o, , , and , to describe other kinds of bounds on asymptotic growth rates.

# Formal definition

Let f and g be two functions defined on some subset of the real numbers. One writes

$$f(x) = O(g(x)) \text{ as } x \to \infty \, f(x) = O(g(x)) \text{ as } x \to \infty$$

if and only if there is a positive constant M such that for all sufficiently large values of x, the absolute value of f(x) is at most M multiplied by the absolute value of g(x). That is, f(x) = O(g(x)) if and only if there exists a positive real number M and a real number x0 such that

$|f(x)| \leq \ M|g(x)|$ for all $x \geq x_0|f(x)| \leq \ M|g(x)|$ for all $x \geq x_0$.

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to infinity is left unstated, and one writes more simply that f(x) = O(g(x)).

The notation can also be used to describe the behavior of f near some real number a (often, a $= 0$): we say

$f(x) = O(g(x))$ as $x \to a$ $f(x) = O(g(x))$ as $x \to a$ *ifandonlyifthereexist*

$|f(x)| \leq M|g(x)|$ for $|x - a| < \delta$ $|f(x)| \leq M|g(x)|$ for $|x - a| < \delta$.

If g(x) is non-zero for values of x sufficiently close to a, both of these definitions can be unified using the limit superior:
$f(x) = O(g(x))$ as $x \to a$ $f(x) = O(g(x))$ as $x \to a$, if and only if $\limsup_{x \to a} \left| \frac{f(x)}{g(x)} \right| < \infty$ $\limsup_{x \to a} \left| \frac{f(x)}{g(x)} \right| < \infty$. Additionally, the notation O(g(x)) is also used to denote the set of all functions f(x) that satisfy the relation f(x)=O(g(x)). In this case we write

$$f(x) \in O(g(x)) f(x) \in O(g(x))$$

.

# Big Oh Notation,

- The (n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or longest amount of time an algorithm can possibly take to complete.

# Example

In typical usage, the formal definition of O notation is not used directly; rather, the O notation for a function f is derived by the following simplification rules:

- If f(x) is a sum of several terms, if there is one with largest growth rate, it can be kept, and all others omitted.
- If f(x) is a product of several factors, any constants (terms in the product that do not depend on x) can be omitted.

- For example, let f(x) = 6x4  2x3 + 5, and suppose we wish to simplify this function, using O notation, to describe its growth rate as x approaches infinity.
- This function is the sum of three terms: $6x^4$, $2x^3$, and 5. Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of x, namely $6x^4$.
- Now one may apply the second rule: 6x4 is a product of 6 and x4 in which the first factor does not depend on x. Omitting this factor results in the simplified form x4.
- Thus, we say that f(x) is a "big-oh" of (x4). Mathematically, we can write f(x) = O(x4).

# Big O-Notation

One may confirm this calculation using the formal definition: let $f(x) = 6x^4 2x^3 + 5$ and $g(x) = x^4$. Applying the formal definition from above, the statement that f(x) = O(x4) is equivalent to its expansion,

$$|f(x)| \leq \ M|x^4||f(x)| \leq \ M|x^4|$$

for some suitable choice of x0 and M and for all x ¿ x0. To prove this, let x0 = 1 and M = 13. Then, for all x ¿ x0:

$$\begin{aligned} |6x^4 - 2x^3 + 5| &\leq 6x^4 + |2x^3| + 5 \\ &\leq 6x^4 + 2x^4 + 5x^4 \\ &= 13x^4 \end{aligned} \qquad \begin{aligned} |6x^4 - 2x^3 + 5| &\leq 6x^4 + |2x^3| + 5 \\ &\leq 6x^4 + 2x^4 + 5x^4 \\ &= 13x^4 \end{aligned}$$

so

$$|6x^4 - 2x^3 + 5| \leq 13\,x^4 . |6x^4 - 2x^3 + 5| \leq 13\,x^4 .$$

# Big O-Notation

For example, for a function f(n)

```
(f(n)) = { g(n) : there exists c > 0 and n0 such that g(n)

        f(n) for all n > n0. }
```

# Omega Notation

**Omega Notation, $\Omega$**

- The $\theta(n)$ is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or best amount of time an algorithm can possibly take to complete.

# Omega Notation

For example, for a function f(n)
(f(n))  g(n) : there exists c ¿ 0 and n0 such that g(n)  c.f(n) for all n ¿ n0.
Theta Notation,  The (n) is the formal way to express both the lower bound and upper bound of an algorithm's running time. It is represented as following

Theta Notation (f(n)) = g(n) if and only if g(n) = (f(n)) and g(n) = (f(n)) for all n ¿ n0.

Common Asymptotic Notations

- constant (1)
- logarithmic (log n)
- linear (n)
- n log n (n log n)
- quadratic (n2)
- cubic (n3)
- polynomial n(1)
- exponential 2(n)

# Greedy Algorithms

Image result for greedy algorithm A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

# Greedy Algorithms

An algorithm is designed to achieve optimum solution for given problem. In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide optimum solution is chosen. Greedy algorithms tries to find localized optimum solution which may eventually land in globally optimized solutions. But generally greedy algorithms do not provide globally optimized solutions.

Counting Coins This problem is to count to a desired value by chosing least possible coins and greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of 1, 2, 5 and 10 and we are asked to count 18 then the greedy procedure will be

1 Select one 10 coin, remaining count is 8
2 Then select one 5 coin, remaining count is 3
3 Then select one 2 coin, remaining count is 1
3 And finally selection of one 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result. For currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins then necessary. For example greedy approach will use $10 + 1 + 1 + 1 + 1 + 1$ total 6 coins. Where the same problem could be solved by using only 3 coins ($7 + 7 + 1$)

Hence, we may conclude that greedy approach picks immediate optimized solution and may fail where global optimization is major concern.

# Networking Algorithms

Examples Most networking algorithms uses greedy approach. Here is the list of few of them
Travelling Salesman Problem Prim's Minimal Spanning Tree Algorithm Kruskal's Minimal Spanning Tree Algorithm Dijkstra's Minimal Spanning Tree Algorithm Graph - Map Coloring Graph - Vertex Cover Knapsack Problem Job Scheduling Problem These and there are lots of similar problems which uses greedy approach to find an optimum solution.

# Divide and Conquer

n divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub-problems into even smaller sub-problems, we may eventually reach at a stage where no more dividation is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of original problem.

# Divide and Conquer

Broadly, we can understand divide-and-conquer approach as three step process.
Divide/Break This step involves breaking the problem into smaller sub-problems. Sub-problems should represent as a part of original problem. This step generally takes recursive approach to divide the problem until no sub-problem is further dividable. At this stage, sub-problems become atomic in nature but still represents some part of actual problem.

Conquer/Solve This step receives lot of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.

Merge/Combine When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

This algorithmic approach works recursively and conquer and merge steps works so close that they appear as one.

Examples The following computer algorithms are based on divide-and-conquer programming approach  Merge Sort Quick Sort Binary Search Strassen's Matrix Multiplication Closest pair (points) There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

# Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem in smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

# Dynamic Programming

Dynamic programming is used where we have problems which can be divided in similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.
So we can say that
The problem should be able to be divided in to smaller overlapping sub-problem.
The optimum solution can be achieved by using optimum solution of smaller sub-problems.

# Dynamic Programming

Dynamic algorithms use memoization.
Comparison In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for overall optimization of the problem.
In contrast to divide and conquer algorithms, where solutions are combined to achieve overall solution, dynamic algorithms uses the output of smaller sub-problem and then try to optimize bigger sub-problem. Dynamic algorithms uses memoization to remember the output of already solved sub-problems.

# Dynamic Programming

Example The following computer problems can be solved using dynamic programming approach
Fibonacci number series Knapsack problem Tower of Hanoi All pair shortest path by Floyd-Warshall Shortest path by Dijkstra Project scheduling Dynamic programming can be used in both top-down and bottom-up manner. And ofcourse, most of the times, referring to previous solution output is cheaper than re-computing in terms of CPU cycles.

# Search Algorithms

Data Structure - Linear Search
Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every items is checked and if a match founds then that particular item is returned otherwise search continues till the end of the data collection.

# Search Algorithms

Linear Search Animation Algorithm Linear Search ( Array A, Value x)

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

Pseudocode

```
procedure linear_search (list, value)

for each item in the list

if match item == value

return the item's location

end if

end for

end procedure
```

Data Structure - Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly the data collection should be in sorted form.

# Binary Search

Binary search search a particular item by comparing the middle most item of the collection. If match occurs then index of item is returned. If middle item is greater than item then item is searched in sub-array to the right of the middle item other wise item is search in sub-array to the left of the middle item. This process continues on sub-array as well until the size of subarray reduces to zero.

# Binary Search

How binary search works? For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with an pictorial example. The below given is our sorted array and assume that we need to search location of value 31 using binary search.
Binary search First, we shall determine the half of the array by using this formula
mid = low + (high - low) / 2 Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So 4 is the mid of array.

# Binary Search

Binary search Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which is not a match. Because value is greater than 27 and we have a sorted array so we also know that target value must be in upper portion of the array.

# Binary Search

Binary search We change our low to mid + 1 and find the new mid value again.
low = mid + 1 mid = low + (high - low) / 2 Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.
Binary search The value stored at location 7 is not a match, rather it is less that what we are looking for. So the value must be in lower part from this location.

# Binary Search

Binary search So we calculate the mid again. This time it is 5.
Binary search We compare the value stored ad location 5 with our target value. We find that it is a match.
Binary search We conclude that the target value 31 is stored at location 5.
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

# Data Structure - Binary Search Tree

# Binary Search

A binary search tree (BST) is a tree in which all nodes follows the below mentioned properties

- The left sub-tree of a node has key less than or equal to its parent node's key.
- The right sub-tree of a node has key greater than or equal to its parent node's key.

Thus, a binary search tree (BST) divides all its sub-trees into two segments; left sub-tree and right sub-tree and can be defined as

$$left_subtree(keys) \leq node(key) \leq right_subtree(keys)$$

# Representation of a Binary Search Tree

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has key and associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.
An example of BST

Binary Search Tree We observe that the root node key (27) has all less-valued keys on the left sub-tree and higher valued keys on the right sub-tree.

Basic Operations Following are basic primary operations of a tree which are following.

Search  search an element in a tree.

Insert  insert an element in a tree.

Preorder Traversal  traverse a tree in a preorder manner.

Inorder Traversal  traverse a tree in an inorder manner.

Postorder Traversal  traverse a tree in a postorder manner.

Node Define a node having some data, references to its left and right child nodes.

```
struct node {
int data;
struct node *leftChild;
struct node *rightChild;
};
```

Dynamic Programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.
Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time.
Dynamic Programmingis a general approach to solving problems, much like divide-and-conquer is a general method, except that unlike divide-and-conquer, the subproblems will typically overlap.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
return elements[0] == null;
}
```

## O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
foreach (var element in elements)
{
if (element == value) return true;
}

return false;
}
```

O(N2)

O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

```
bool ContainsDuplicates(IList<string> elements)
{
for (var outer = 0; outer < elements.Count; outer++)
{
for (var inner = 0; inner < elements.Count; inner++)
{
// Don't compare with self
if (outer == inner) continue;

if (elements[outer] == elements[inner]) return true;
}
}

return false;
}
```

O(2N)

O(2N) denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2N) function is the recursive calculation of Fibonacci numbers:

int Fibonacci(int number) if (number ¡= 1) return number; return Fibonacci(number - 2) + Fibonacci(number - 1);

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as O(log N). The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets. This article only covers the very basics or Big O and logarithms. For a more in-depth explanation take a look at their respective Wikipedia entries: Big O Notation, Logarithms.

**Search Operation** Whenever an element is to be search. Start search from root node then if data is less than key value, search element in left subtree otherwise search element in right subtree. Follow the same algorithm for each node.

Insert Operation Whenever an element is to be inserted. First locate its proper location. Start search from root node then if data is less than key value, search empty location in left subtree and insert the data. Otherwise search empty location in right subtree and insert the data.

# Data Structure - AVL Trees

What if the input to binary search tree comes in sorted (ascending or descending) manner? It will then look like this
Unbalanced BST It is observed that BST's worst-case performance closes to linear search algorithms, that is (n). In real time data we cannot predict data pattern and their frequencies. So a need arises to balance out existing BST.

Named after their inventor **Adelson, Velski & Landis**, AVL trees are height balancing binary search tree. AVL tree checks the height of left and right sub-trees and assures that the difference is not more than 1. This difference is called Balance Factor.
Here we see that the first tree is balanced and next two trees are not balanced

# Unbalanced AVL Trees

In second tree, the left subtree of C has height 2 and right subtree has height 0, so the difference is 2. In third tree, the right subtree of A has height 2 and left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.
BalanceFactor = height(left-sutree)  height(right-sutree) If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

# AVL Rotations

To make itself balanced, an AVL tree may perform four kinds of rotations
Left rotation Right rotation Left-Right rotation Right-Left rotation First two rotations are single rotations and next two rotations are double rotations. Two have an unbalanced tree we at least need a tree of height 2. With this simple tree, let's understand them one by one.

# AVL Rotations

Left Rotation If a tree become unbalanced, when a node is inserted into the right subtree of right subtree, then we perform single left rotation
Left Rotation In our example, node A has become unbalanced as a node is inserted in right subtree of A's right subtree. We perform left rotation by making A left-subtree of B.

# AVL Rotations

Right Rotation AVL tree may become unbalanced if a node is inserted in the left subtree of left subtree. The tree then needs a right rotation.
Right Rotation As depicted, the unbalanced node becomes right child of its left child by performing a right rotation.
Left-Right Rotation Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is combination of left rotation followed by right rotation.

# AVL Rotations

State Action Right Rotation A node has been inserted into right subtree of left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. Left Rotation We first perform left rotation on left subtree of C. This makes A, left subtree of B. Left Rotation Node C is still unbalanced but now, it is because of left-subtree of left-subtree.

# AVL Rotations

Right Rotation We shall now right-rotate the tree making B new root node of this subtree. C now becomes right subtree of its own left subtree. Balanced Avl Tree The tree is now balanced. Right-Left Rotation Second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

# AVL Rotations

State Action Left Subtree of Right Subtree A node has been inserted into left subtree of right subtree. This makes A an unbalanced node, with balance factor 2. Subtree Right Rotation First, we perform right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes right subtree of A. Right Unbalanced Tree Node A is still unbalanced because of right subtree of its right subtree and requires a left rotation. Left Rotation A left rotation is performed by making B the new root node of the subtree. A becomes left subtree of its right subtree B. Balanced AVL Tree The tree is now balanced.