DAA

What is an Algorithm?

A finite set of instructions that specifies a sequence of operations to be carried out to solve a specific problem or class of problems is called an Algorithm.

## What is meant by Algorithm Analysis?

Algorithm analysis refers to how to investigate the algorithm's effectiveness in terms of time and space complexity. The fundamental purpose of algorithm evaluation is to decide how much time and space an algorithm needs to solve the problem as a feature of the scale of the input.

Types of Algorithm Analysis:

There are numerous types of algorithm analysis which can be generally used to measure the performance and efficiency of algorithms:

1. Time complexity evaluation: This kind of analysis measures the running time of an algorithm as a characteristic of the input length. It typically entails counting the number of primary operations completed by way of the algorithm, such as comparisons, mathematics operations, and reminiscence accesses.
2. Space complexity evaluation: This form of evaluation measures the amount of memory required via an algorithm as a characteristic of the entered size. It typically includes counting the variety of variables and information systems utilized by the algorithm, as well as the size of each of these record structures.
3. Worst-case evaluation: This type of analysis measures the worst-case running time or space utilization of an algorithm, assuming the enter is the maximum toughest viable for the algorithm to deal with.
4. Average-case analysis: This kind of evaluation measures the predicted walking time or area usage of an algorithm, assuming a probabilistic distribution of inputs.
5. Best-case evaluation: This form of analysis measures the nice case running time or area utilization of an algorithm, assuming the input is the easiest possible for the algorithm to address.
6. Asymptotic analysis: This sort of analysis measures the overall performance of an algorithm as the enter size methods infinity. It normally includes the usage of mathematical notation to describe the boom fee of the algorithm's strolling time or area usage, including O(n), Ω(n), or Θ(n).

Advantages of design and analysis of algorithm:

There are numerous blessings of designing and studying algorithms:

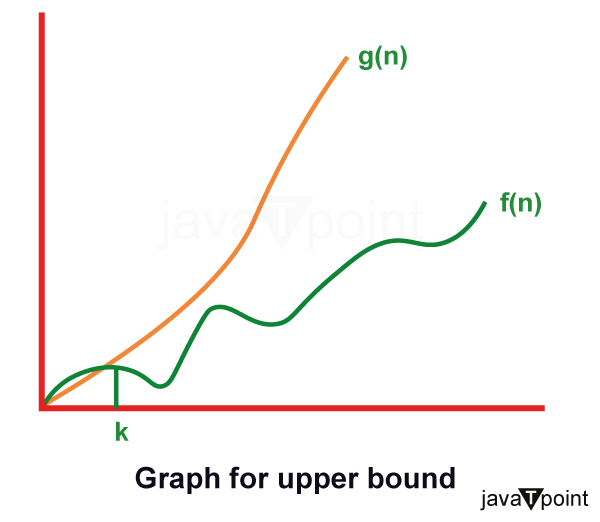
1. Improved efficiency: A properly designed algorithm can notably improve the performance of a program, leading to quicker execution instances and reduced resource utilization. By studying algorithms and identifying regions of inefficiency, developers can optimize the algorithm to lessen its time and space complexity.
2. Better scalability: As the size of the input information increases, poorly designed algorithms can quickly turn out to be unmanageable, leading to slow execution times and crashes. By designing algorithms that scale well with increasing input sizes, developers can make certain that their packages stay usable while the facts they take care of grow.
3. Improved code exceptional: A nicely designed algorithm can result in better code first-rate standards because it encourages developers to think seriously about their application's shape and organization. By breaking down complicated issues into smaller, extra-manageable subproblems, builders can create simpler code to recognize and maintain.
4. Increased innovation: By knowing how algorithms work and how they can be optimized, developers can create new and progressive solutions to complex problems. This can lead to new merchandise, services, and technologies which can have a considerable impact on the arena.
5. Competitive benefit: In industries where pace and performance are vital, having properly designed algorithms can provide an extensive competitive advantage. By optimizing algorithms to lessen expenses and enhance performance, groups can gain a facet over their competitors.

## Asymptotic Notations:

Asymptotic Notation is a way of comparing functions that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

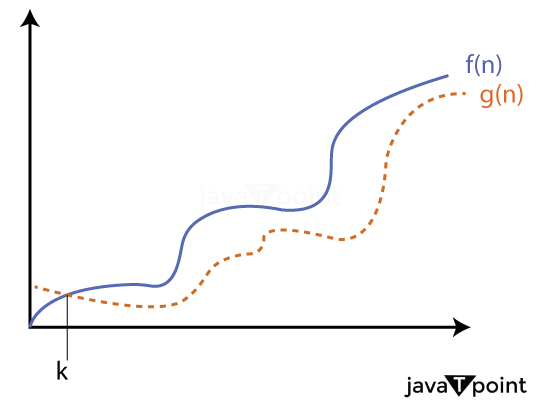
**1. Big-oh notation:** Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exists positive constant c and such that

1. f (n) ⩽ k.g (n)f(n)⩽k.g(n) for n>n0n>n0 in all case



Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)

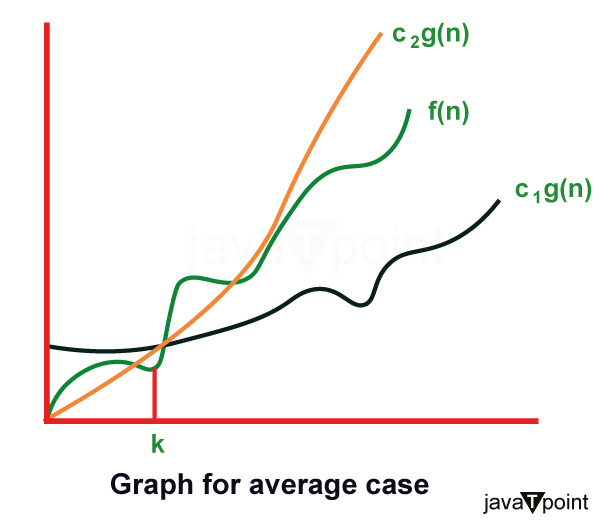
1. **Omega () Notation:** The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n0 such that



F (n) ≥ k\* g (n) for all n, n≥ n0

1. **Theta (θ):** The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k1, k2 and k0 such that:

k1 \* g (n) ≤ f(n)≤ k2 g(n)for all n, n≥ n0



## What is Recurrence Relation?

A **recurrence relation** is a mathematical expression that defines a sequence in terms of its previous terms. In the context of algorithmic analysis, it is often used to model the time complexity of recursive algorithms.

The general form of a **Recurrence Relation:**   
where **f**is a function that defines the relationship between the current term and the previous terms

Significance of Recurrence Relations in DSA:

Recurrence Relations play a significant role in analyzing and optimizing the complexity of algorithms. Having a strong understanding of Recurrence Relations plays a great role in developing the problem-solving skills of an individual. Some of the common uses of Recurrence Relations are:

* Time Complexity Analysis
* Generalizing Divide and Conquer Algorithms
* Analyzing Recursive Algorithms
* Defining State and Transitions for Dynamic Programming

Common Examples of Recurrence Relations:

| Example | Recurrence Relation |
| --- | --- |
| [Fibonacci Sequence](https://www.geeksforgeeks.org/fibonacci-sequence/) | F(n) = F(n-1) + F(n-2) |
| [Factorial of a number n](https://www.geeksforgeeks.org/program-for-factorial-of-a-number/) | F(n) = n \* F(n-1) |
| [Merge Sort](https://www.geeksforgeeks.org/merge-sort/) | T(n) = 2\*T(n/2) + O(n) |
| [Tower of Hanoi](https://www.geeksforgeeks.org/time-complexity-analysis-tower-hanoi-recursion/) | H(n) = 2\*H(n-1) + 1 |
| [Binary Search](https://www.geeksforgeeks.org/binary-search/) | T(n) = T(n/2) + 1 |

[Types of Recurrence Relations:](https://www.geeksforgeeks.org/different-types-recurrence-relations-solutions/)

Various types of Recurrence Relations are:

1. Linear Recurrence Relations
2. Divide and Conquer Recurrences
3. Substitution Recurrences
4. Homogeneous Recurrences
5. Non-Homogeneous Recurrences

## What is Master's theorem?

Master’s theorem is one of the many methods that are applied to calculate the time complexities of algorithms. In analysis, time complexities are calculated to find out the best optimal logic of an algorithm. Master’s theorem is applied to recurrence relations.

## Problem Statement

Master’s theorem can only be applied to decreasing and dividing recurring functions. If the relation is not decreasing or dividing, the master’s theorem must not be applied.

### Master’s Theorem for Dividing Functions

Consider a relation of type −

T(n) = aT(n/b) + f(n)

where, a >= 1 and b > 1,

n − size of the problem

a − number of sub-problems in the recursion

n/b − the size of the sub-problems based on the assumption that all sub-problems are of the same size.

f(n) − represents the cost of work done outside the recursion -> Θ(nk log p) ,where k >= 0 and p is a real number;

If the recurrence relation is in the above-given form, then there are three cases in the master theorem to determine the asymptotic notations −

* If a > bk , then T(n)= Θ (nlogb a ) [ logb a = log a / log b. ]
* If a = bk
  + If p > -1, then T(n) = Θ (nlogb a logp+1 n)
  + If p = -1, then T(n) = Θ (n logb a log log n)
  + If p < -1, then T(n) = Θ (n logb a)
* If a < bk,
  + If p >= 0, then T(n) = Θ (nk logp n).
  + If p < 0, then T(n) = Θ (nk)

### Master’s Theorem for Decreasing Functions

Consider a relation of type −

T(n) = aT(n-b) + f(n)

where, a >= 1 and b > 1, f(n) is asymptotically positive

Here,

n − size of the problem

a − number of sub-problems in the recursion

n-b − the size of the sub-problems based on the assumption that all sub-problems are of the same size.

f(n) − represents the cost of work done outside the recursion -> Θ(nk), where k >= 0.

If the recurrence relation is in the above-given form, then there are three cases in the master theorem to determine the asymptotic notations −

* if a = 1, T(n) = O (nk+1)
* if a > 1, T(n) = O (an/b \* nk)
* if a < 1, T(n) = O (nk)

## Examples

Few examples to apply Master’s theorem on dividing recurrence relations −

### Example 1

Consider a recurrence relation given as T(n) = 8T(n/2) + n2

In this problem, a = 8, b = 2 and f(n) = Θ(nk log p) = n2,

giving us k = 2 and p = 0.

a = 8 > bk = 22 = 4,

Hence, case 1 must be applied to this equation.

To calculate, T(n) = Θ (log a )

= nlog28

= n( log 8 / log 2 )

= n3

Therefore, T(n) = Θ(n3) is the tight bound for this equation.

### Example 2

Consider a recurrence relation given as T(n) = 4T(n/2) + n2

In this problem, a = 4, b = 2 and f(n) = Θ(nk log p) = n2,

giving us k = 2 and p = 0.

a = 4 = bk = 22 = 4, p > -1

Hence, case 2(i) must be applied to this equation.

To calculate, T(n) = Θ (nlogb a logp+1 n)

= nlog24 log0+1n

= n2logn

Therefore, T(n) = Θ(n2logn) is the tight bound for this equation.

### Example 3

Consider a recurrence relation given as T(n) = 2T(n/2) + n/log n

In this problem, a = 2, b = 2 and f(n) = Θ(nk log p) = n/log n,

giving us k = 1 and p = -1.

a = 2 = bk = 21 = 2, p = -1

Hence, case 2(ii) must be applied to this equation.

To calculate, T(n) = Θ (n log a log log n)

= nlog44 log logn

= n.log(login)

Therefore, T(n) = Θ(n.log(logn)) is the tight bound for this equation.

### Example 4

Consider a recurrence relation given as T(n) = 16T(n/4) + n2/log2n

In this problem, a = 16, b = 4 and f(n) = Θ(nk log p) = n2/log2n,

giving us k = 2 and p = -2.

a = 16 = bk = 42 = 16, p < -1

Hence, case 2(iii) must be applied to this equation.

To calculate, T(n) = Θ (n log a)

= nlog416

= n2

Therefore, T(n) = Θ(n2) is the tight bound for this equation.

### Example 5

Consider a recurrence relation given as T(n) = 2T(n/2) + n2

In this problem, a = 2, b = 2 and f(n) = Θ(nk log p) = n2,

giving us k = 2 and p = 0.

a = 2 < bk = 22 = 4, p = 0

Hence, case 3(i) must be applied to this equation.

To calculate, T(n) = Θ (nk log n)

= n2 log0n

= n2

Therefore, T(n) = Θ(n2) is the tight bound for this equation.

Few examples to apply Master’s theorem in decreasing recurrence relations −

### Example 1

Consider a recurrence relation given as T(n) = T(n-1) + n2

In this problem, a = 1, b = 1 and f(n) = O(nk) = n2, giving us k = 2.

Since a = 1, case 1 must be applied for this equation.

To calculate, T(n) = O(nk+1)

= n2+1

= n3

Therefore, T(n) = O(n3) is the tight bound for this equation.

### Example 2

Consider a recurrence relation given as T(n) = 2T(n-1) + n

In this problem, a = 2, b = 1 and f(n) = O(nk) = n, giving us k = 1.

Since a > 1, case 2 must be applied for this equation.

To calculate, T(n) = O(an/b \* nk)

= O(2n/1 \* n1)

= O(n2n)

Therefore, T(n) = O(n2n) is the tight bound for this equation.

### Example 3

Consider a recurrence relation given as T(n) = n4

In this problem, a = 0 and f(n) = O(nk) = n4, giving us k = 4

Since a < 1, case 3 must be applied for this equation.

To calculate, T(n) = O(nk)

= O(n4)

= O(n4)

Therefore, T(n) = O(n4) is the tight bound for this equation.

**Catalan numbers** are defined as a mathematical sequence that consists of positive integers, which can be used to find the number of possibilities of various combinations.

The **nth**term in the sequence denoted**Cn**, is found in the following formula:

The first few Catalan numbers for n = 0, 1, 2, 3, … are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, … 

## **Program for nth Catalan Number using Recursion:**

## import java.io.\*;

## class CatalnNumber {

## // A recursive function to find the nth Catalan number

## int catalan(int n)

## {

## int res = 0;

## // Base case

## if (n <= 1) {

## return 1;

## }

## for (int i = 0; i < n; i++) {

## res += catalan(i) \* catalan(n - i - 1);

## }

## return res;

## }

## // Driver Code

## public static void main(String[] args)

## {

## CatalnNumber cn = new CatalnNumber();

## for (int i = 0; i < 10; i++) {

## System.out.print(cn.catalan(i) + " ");

## }

## }

## }

Output

1 1 2 5 14 42 132 429 1430 4862

## Tree:

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:

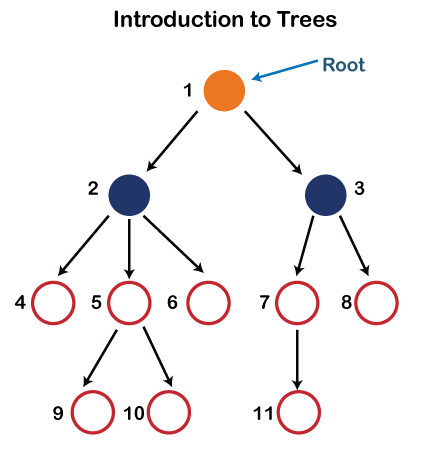
## 

The above tree shows the **organizational hierarchy** of some companies. In the above structure, John is the **CEO** of the company, and John has two direct reports named Steve and Rohan. Steve has three direct reports named Lee, Bob, and Ella where Steve is a manager. Bob has two direct reports named Sal and Emma. **Emma** has two direct reports named Tom and Raj. Tom has one direct report named Bill. This particular logical structure is known as a Tree. Its structure is similar to the real tree, so it is named a Tree. In this structure, the root is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data hierarchically.

Notes:

* A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
* A tree data structure is a non-linear data structure because it does not store sequentially. It is a hierarchical structure as elements in a Tree are arranged on multiple levels.
* In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
* Each node contains some data and the link or reference of other nodes that can be called children.

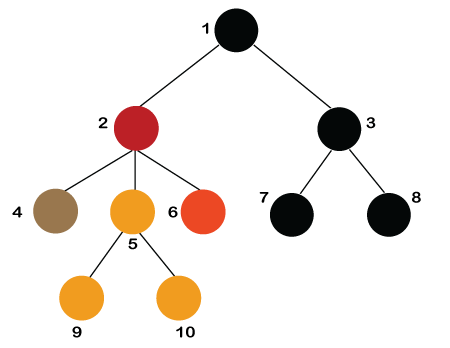
Important terms:



* Root: The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a parent-child relationship.
* Child node: If the node is a descendant of any node, then the node is known as a child node.
* Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.
* Sibling: The nodes that have the same parent are known as siblings.
* Leaf Node:- The node of the tree, that doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
* Internal nodes: A node has at least one child node known as an internal
* Ancestor node:- An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
* Descendant: The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

### Properties of Tree data structure

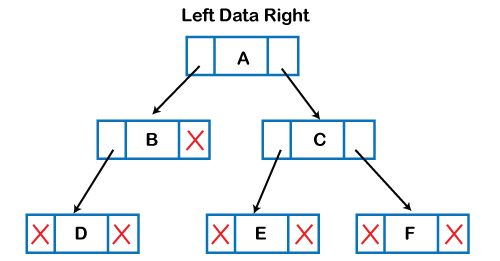
* **Recursive data structure:** The tree is also known as a recursive data structure. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a root node. The root node of the tree contains a link to all the roots of its subtrees. The left subtree is shown in the yellow color in the below figure, and the right subtree is shown in the red color. The left subtree can be further split into subtrees shown in three different colors. Recursion means reducing something in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.



* Number of edges: If there are n nodes, then there would be n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have at least one incoming link known as an edge. There would be one link for the parent-child relationship.
* Depth of node x: The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one unit of length to the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
* Height of node x: The height of node x can be defined as the longest path from the node x to the leaf node.

### Implementation of Tree

The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

In programming, the structure of a node can be defined as:

1. struct node
2. {
3. int data;
4. struct node \*left;
5. struct node \*right;
6. }

### Applications of trees

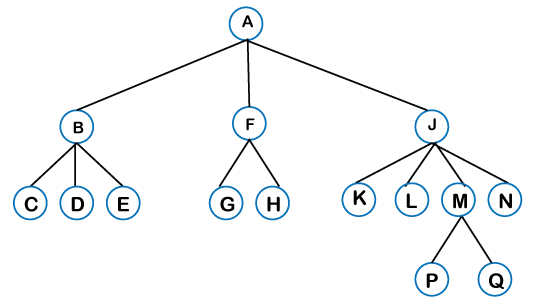
The following are the applications of trees:

* **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system is stored on the disc drive, the file and folder are in the form of naturally hierarchical data and stored in the form of trees.
* **Organize data:** It is used to organize data for efficient insertion, deletion, and searching. For example, a binary tree has a long time for searching an element.
* **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell-checking.
* **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
* **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
* **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

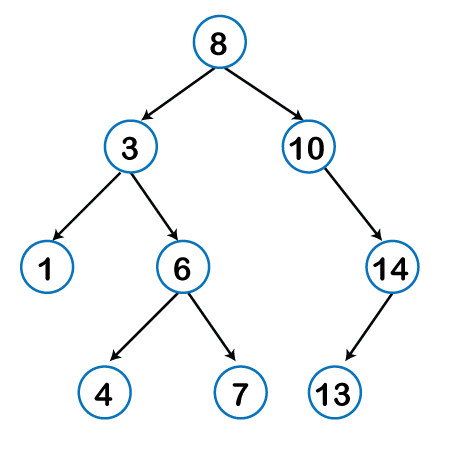
### Types of Tree Data Structure

**The following are the types of a tree data structure:**

* **General tree:** The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or a maximum number of nodes. There is no restriction imposed on the degree of the node (the number of nodes that a node can contain). The topmost node in a general tree is known as a root node. The children of the parent node are known as subtrees.



* There can be n number of subtrees in a general tree. In the general tree, the subtrees are unordered as the nodes in the subtree cannot be ordered.  
  Every non-empty tree has a downward edge, and these edges are connected to the nodes known as child nodes. The root node is labeled with level 0. The nodes that have the same parent are known as siblings.
* [Binary tree](https://www.javatpoint.com/binary-tree): Here, the binary name itself suggests two numbers, i.e., 0 and 1. In a binary tree, each node in a tree can have two child nodes. Here, utmost means whether the node has 0 nodes, 1 node, or 2 nodes.



* [Binary Search tree](https://www.javatpoint.com/binary-search-tree): Binary search tree is a non-linear data structure in which one node is connected to n number of nodes. It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).  
  Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

A node can be created with the help of a user-defined data type known as struct, as shown below:

1. struct node
2. {
3. int data;
4. struct node \*left;
5. struct node \*right;
6. }

The above is the node structure with three fields: data field, the second field is the left pointer of the node type, and the third field is the right pointer of the node type.

* [AVL tree](https://www.javatpoint.com/avl-tree)

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the binary tree as well as of the binary search tree. It is a self-balancing binary search tree that was invented by Adelson Velsky Lindas. Here, self-balancing means balancing the heights of the left subtree and the right subtree. This balancing is measured in terms of the balancing factor

We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor. The balancing factor can be defined as the difference between the height of the left subtree and the height of the right subtree. The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

* [B-tree](https://www.javatpoint.com/b-tree)

B-tree is a balanced m-way tree where m defines the order of the tree. Till now, we read that the node contains only one key but b-tree can have more than one key, and more than 2 children. It always maintains the sorted data. In a binary tree, it is possible that leaf nodes can be at different levels, but in a b-tree, all the leaf nodes must be at the same level.

If an order is m then the node has the following properties:

* Each node in a b-tree can have maximum m children
* For minimum children, a leaf node has 0 children, a root node has a minimum of 2 children and an internal node has a minimum ceiling of m/2 children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain a maximum of 3 children.
* Each node has maximum (m-1) keys.

The root node must contain a minimum of 1 key and all other nodes must contain at least a ceiling of m/2 minus 1 key.

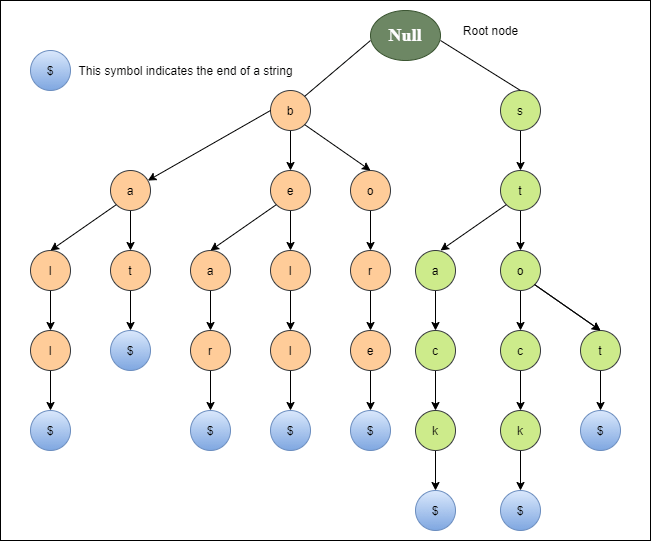
# What is a Trie data structure?

The word "**Trie**" is an excerpt from the word "**retrieval**". Trie is a sorted tree-based data structure that stores a set of strings. It has the number of pointers equal to the number of characters of the alphabet in each node. It can search for a word in the dictionary with the help of the word's prefix. For example, if we assume that all strings are formed from the letters '**a**' to '**z**' in the English alphabet, each trie node can have a maximum of **26** points.

Trie is also known as the digital tree or prefix tree. The position of a node in the Trie determines the key with which that node is connected.

## Properties of the Trie for a set of the string:

1. The root node of the tree always represents the null node.
2. Each child of nodes is sorted alphabetically.
3. Each node can have a maximum of **26** children (A to Z).
4. Each node (except the root) can store one letter of the alphabet.



The diagram depicts a true representation of the bell, bear, bore, bat, ball, stop, stock, and stack.

## Basic operations of Trie

There are three operations in the Trie:

1. Insertion of a node
2. Searching a node
3. Deletion of a node

### Insert of a node in the Trie

The first operation is to insert a new node into the tree. Before we start the implementation, it is important to understand some points:

1. Every letter of the input key (word) is inserted as an individual in the Trie\_node. Note that children point to the next level of Trie nodes.
2. The key character array acts as an index of children.
3. If the present node already has a reference to the present letter, set the present node to that referenced node. Otherwise, create a new node, set the letter to be equal to the present letter, and even start the present node with this new node.
4. The character length determines the depth of the trie.

**Implementation of inserting a new node in the Trie**

1. public class Data\_Trie {
2. private Node\_Trie root;
3. public Data\_Trie(){
4. this.root = new Node\_Trie();
5. }
6. public void insert(String word){
7. Node\_Trie current = root;
8. int length = word.length();
9. for (int x = 0; x < length; x++){
10. char L = word.charAt(x);
11. Node\_Trie node = current.getNode().get(L);
12. if (node == null){
13. node = new Node\_Trie ();
14. current.getNode().put(L, node);
15. }
16. current = node;
17. }
18. current.set word(true);
19. }
20. }

### Searching a node in Trie

The second operation is to search for a node in a Trie. The searching operation is similar to the insertion operation. The search operation is used to search for a key in the trie. The implementation of the search operation is shown below.

Implementation of search a node in the Trie

1. class Search\_Trie {
3. private Node\_Trie Prefix\_Search(String W) {
4. Node\_Trie node = R;
5. for (int x = 0; x < W.length(); x++) {
6. char curLetter = W.charAt(x);
7. if (node.containsKey(our letter))
8. {
9. node = node.get(our letter);
10. }
11. else {
12. return null;
13. }
14. }
15. return node;
16. }
18. public boolean search(String W) {
19. Node\_Trie node = Prefix\_Search(W);
20. return node != null && node.send();
21. }
22. }

### Deletion of a node in the Trie

The Third operation is the deletion of a node in the Trie. Before we begin the implementation, it is important to understand some points:

1. If the key is not found in the trie, the delete operation will stop and exit it.
2. If the key is found in the trie, delete it from the trie.

**Implementation of deleting a node in the Trie**

1. public void Node\_delete(String W)
2. {
3. Node\_delete(R, W, 0);
4. }
6. private boolean Node\_delete(Node\_Trie current, String W, int Node\_index) {
7. if (Node\_index == W.length()) {
8. if (!current.isEndOfWord()) {
9. return false;
10. }
11. current.setEndOfWord(false);
12. return current.getChildren().isEmpty();
13. }
14. char A = W.charAt(Node\_index);
15. Node\_Trie node = current.getChildren().get(A);
16. if (node == null) {
17. return false;
18. }
19. boolean Current\_Node\_Delete = Node\_delete(node, W, Node\_index + 1) && !node.pseudoword();
21. if (Current\_Node\_Delete) {
22. current.getChildren().remove(A);
23. return current.getChildren().isEmpty();
24. }
25. return false;
26. }

## Applications of Trie

1. **Spell Checker**
2. **Auto-complete**
3. **. Browser history**

Disadvantages of Trie

1. It requires more memory to store the strings.
2. It is slower than the hash table

Advantages of Trie

1. It can be inserted faster and search the string than hash tables and binary search trees.
2. It provides an alphabetical filter of entries by the key of the node.

# The Knuth-Morris-Pratt (KMP)Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string-matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that has previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs

String:

In data structures, a **string** is a sequence of characters used to represent text. **Strings** are commonly used for storing and manipulating textual data in computer programs. They can be manipulated using various operations like **concatenation, substring extraction, and comparison**.

## String Data Type

In most programming languages, strings are treated as a distinct **data type**. This means that strings have their own set of operations and properties. They can be declared and manipulated using specific string-related functions and methods.

**Note:**In some languages, strings are implemented as arrays of characters, making them a derived data type.

String Operations

Strings support a wide range of operations, including concatenation, substring extraction, length calculation, and more. These operations allow developers to manipulate and process string data efficiently.

Below are fundamental operations commonly performed on strings in programming.

* Concatenation: Combining two strings to create a new string.
* Length: Determining the number of characters in a string.
* Access: Accessing individual characters in a string by index.
* Substring: Extracting a portion of a string.
* Comparison: Comparing two strings to check for equality or order.
* Search: Finding the position of a specific substring within a string.
* Modification: Changing or replacing characters within a string.

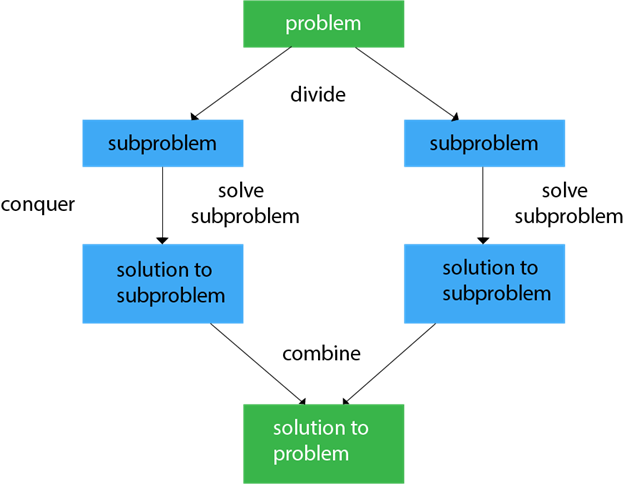
Note: <https://www.geeksforgeeks.org/string-data-structure/>

# Divide and Conquer:

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, and recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

**Examples:** The specific computer algorithms are based on the divide-and-conquer approach

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

## Applications of Divide and Conquer Approach:

1. Binary Search: The binary search algorithm is a search algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually be eliminated, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. Quicksort: It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sorts the arrays recursively.
3. Merge Sort: It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. Closest Pair of Points: It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric space, given n points, such that the distance between the pair of points should be minimal.
5. Strassen's Algorithm: It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when working on large matrices.
6. Cooley-Tukey Fast Fourier Transform (FFT) algorithm: The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).
7. Karatsuba algorithm for fast multiplication: It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and published in 1962. It multiplies two n-digit numbers in such a way by reducing them to at most single-digit

## Advantages of Divide and Conquer

* Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solving them recursively. This algorithm is much faster than other algorithms.
* It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
* It is more proficient than its counterpart Brute Force technique.
* Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

Disadvantages of Divide and Conquer

* Since most of its algorithms are designed by incorporating recursion, it necessitates high memory management.
* An explicit stack may overuse the space.
* It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

# Max - Min Problem

**Problem:** Analyze the algorithm to find the maximum and minimum elements from an array.

**Algorithm: Max?Min Element (a [])**

Max: a [i]

Min: a [i]

For i= 2 to n do

If a[i]> max then

max = a[i]

if a[i] < min then

min: a[i]

return (max, min)

### Analysis:

**Method 1:** if we apply the general approach to the array of size n, the number of comparisons required is 2n-2.

**Method-2:** In another approach, we will divide the problem into sub-problems and find the max and min of each group, now max. Each group will compare with the only max of another group and min with min.

Let n = be the size of items in an array

Let T (n) = time required to apply the algorithm on an array of size n. Here we divide the terms as T(n/2).

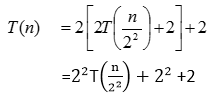
2 here tends to the comparison of the minimum with minimum and maximum with maximum as in the above example.



Max - Min Problem

T (n) = 2 T Max - Min Problem → Eq (i)

T (2) = 1, the time required to compare two elements/items. (Time is measured in units of the number of comparisons)



Similarly, apply the same procedure recursively on each subproblem or anatomy

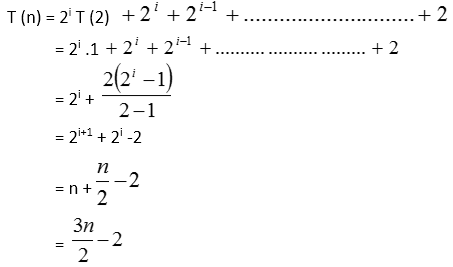
{Use recursion means, we will use some stopping condition to stop the algorithm}



Max - Min Problem

Recursion will stop, when Max - Min Problem → (Eq. 4)

Put the equation 4 into equation 3.



The number of comparisons requires applying the divide and conquering algorithm on n elements/items = 

Number of comparisons requires applying general approach on n elements = (n-1) + (n-1) = 2n-2

From this example, we can analyze, that how to reduce the number of comparisons by using this technique.

Analysis: suppose we have an array of size 8 elements.

Method1: requires (2n-2), (2x8)-2=14 comparisons

Method2: requires Max - Min Problem

It is evident; that we can reduce the number of comparisons (complexity) by using a proper technique