Table of content

|  |  |  |
| --- | --- | --- |
| 1. Tree | 1. Basic terminology 2. Types of trees 3. Basic operation of trees 4. Why tree 5. Properties of tree 6. Need for tree 7. Application of tree 8. Advantages of tree 9. Disadvantages of tree 10. 1 point example of disadavantage |  |
| Binary tree | 1. Basic operations in binary tree 2. Auxilary operations 3. Application of binary tree |  |
| Binary tree traversal | 1. DFS & BFS 2. Properties of binary tree 3. Types of binary tree 4. Special types of binary tree 5. Application of binary tree 6. Real time application of binary tree 7. Advantages and disadvantages of binary tree |  |

Tree

A **tree data structure** is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

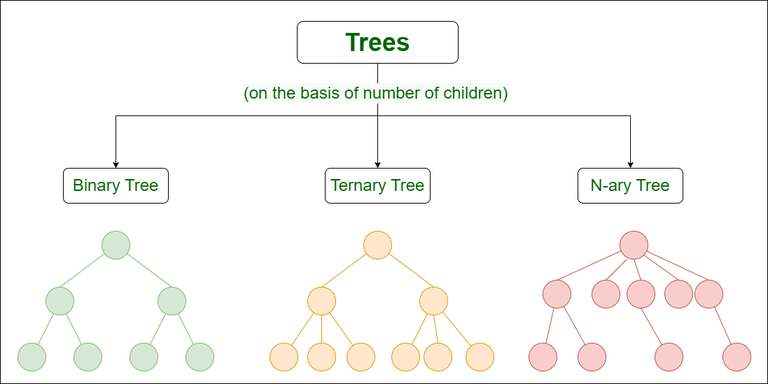
The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



**Basic Terminologies In Tree Data Structure:**

* **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.**{B}** is the parent node of **{D, E}**.
* **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
* **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
* **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{K, L, M, N, O, P, G}** are the leaf nodes of the tree.
* **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node.**{A,B}** are the ancestor nodes of the node**{E}**
* **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
* **Sibling:** Children of the same parent node are called siblings.**{D,E}** are called siblings.
* **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
* **Internal node:** A node with at least one child is called Internal Node.
* **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
* **Subtree**: Any node of the tree along with its descendant.

## Types of Tree data structures:



* [**Binary tree**](https://www.geeksforgeeks.org/types-of-trees-in-data-structures/)**:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
* [**Ternary Tree**](https://www.geeksforgeeks.org/ternary-tree/)**:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
* [**N-ary Tree or Generic Tree**](https://www.geeksforgeeks.org/generic-treesn-array-trees/)**:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

## Basic Operation Of Tree Data Structure:

* **Create** – create a tree in the data structure.
* **Insert** − Inserts data in a tree.
* **Search** − Searches specific data in a tree to check whether it is present or not.
* **Traversal**:
  + **Preorder Traversal** – perform Traveling a tree in a pre-order manner in the data structure.
  + **In order Traversal** – perform Traveling a tree in an in-order manner.
  + **Post-order Traversal** –perform Traveling a tree in a post-order manner.

## Why Tree is considered a non-linear data structure?

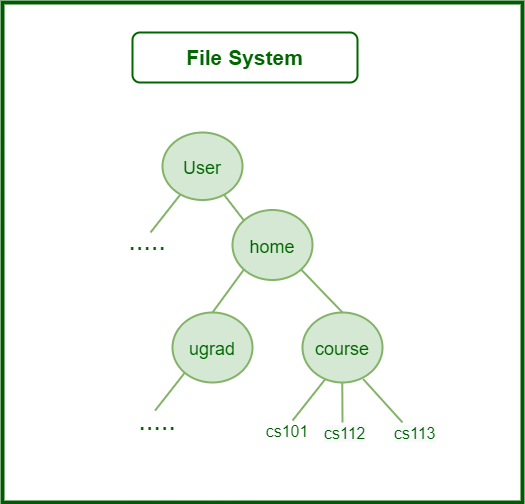
The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a [non-linear data structure](https://www.geeksforgeeks.org/difference-between-linear-and-non-linear-data-structures/).

## Properties of Tree Data Structure:

* **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
* **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
* **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
* **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
* **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

**Need for Tree Data Structure:**

**1.** One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:



*File System*

**2.** Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).   
**3.** Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).   
**4.** Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on the number of nodes as nodes are linked using pointers.

**Application of Tree Data Structure:**

* **File System:** This allows for efficient navigation and organization of files.
* **Data Compression**:[Huffman coding](https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/) is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
* **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
* **Database Indexing**: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

**Advantages of Tree Data Structure:**

* Tree offer **Efficient Searching** Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
* Trees provide a hierarchical representation of data, making it**easy to organize and navigate**large amounts of information.
* The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

**Disadvantages of Tree Data Structure:**

* Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to **inefficient search times.**
* Trees demand**more memory space requirements** than some other data structures like arrays and linked lists, especially if the tree is very large.
* The implementation and **manipulation of trees can be complex**and require a good understanding of the algorithms.

|  |
| --- |
| Unbalanced tree skewedAn unbalanced tree, particularly when referring to binary search trees (BSTs), means that the structure of the tree does not have an even distribution of nodes across its left and right subtrees. This imbalance can significantly affect the performance of various tree operations such as searching, inserting, and deleting.Characteristics of Unbalanced TreesIn an ideal, balanced BST, the nodes are distributed in such a way that the height of the tree (the longest path from the root node to a leaf node) is minimized, roughly maintaining a complexity of \(O(\log n)\) for search, insert, and delete operations, where \(n\) is the number of nodes in the tree. This efficient behavior is because the tree maintains a structure that allows each comparison (moving left for smaller values and right for larger values) to eliminate about half of the remaining possible nodes at each step.However, in an unbalanced tree:1. Skewed Structure: The tree might heavily lean towards the left or the right. Common unbalanced forms include:- Left-skewed tree (or left-heavy): All nodes have only left children, resembling a linked list. This happens when you insert nodes in descending order.- Right-skewed tree (or right-heavy): All nodes have only right children, also resembling a linked list. This happens when you insert nodes in ascending order.2. Increased Tree Height: Due to the skewed structure, the height of the tree approaches \(n\) (where \(n\) is the number of nodes), rather than \(\log n\) as in balanced trees.- Efficiency Loss: The primary drawback of an unbalanced tree is the degradation of performance in operations like searching, inserting, and deleting. Instead of operating in \(O(\log n)\) time complexity, these operations may degrade to \(O(n)\) in the worst case. For example, searching for an element in a completely skewed tree (which essentially becomes a linked list) will require potentially checking every element, just as in a linear search in an array or linked list.- Operational Impact: In real-world applications, where rapid data retrieval and modification are crucial (like databases, file systems), unbalanced trees can significantly slow down operations, impacting overall system performance.Consider a scenario where you are inserting the following series of numbers into a BST: [1, 2, 3, 4, 5]. If inserted in this order into a basic BST without any balancing mechanism, the tree will look like this:1\2\3\4\5This tree is a right-skewed tree. Searching for the number 5 would require traversing through all the nodes, making the operation (O(n)) rather than (O(log n)).The concept of unbalanced trees highlights the importance of balancing mechanisms in BSTs, such as those used in AVL trees or Red-Black trees, which automatically ensure the tree remains balanced after every insert and delete operation, thus maintaining the operation costs in(O(log n)) and ensuring efficient performance. |

Binary tree

*A binary tree is a tree data structure in which each node can have at most two children, which are referred to as the left child and the right child.*

The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom.

Binary trees have many applications in computer science, including data storage and retrieval, expression evaluation, network routing, and game AI. They can also be used to implement various algorithms such as searching, sorting, and graph algorithms.

## ****Basic Operations On Binary Tree:****

* Inserting an element.
* Removing an element.
* Searching for an element.
* Deletion for an element.
* Traversing an element. There are four (mainly three) types of traversals in a binary tree which will be discussed ahead.

## ****Auxiliary Operations On Binary Tree:****

* Finding the height of the tree
* Find the level of the tree
* Finding the size of the entire tree.

## ****Applications of Binary Tree:****

* In compilers, Expression Trees are used which is an application of binary trees.
* Huffman coding trees are used in data compression algorithms.
* Priority Queue is another application of binary tree that is used for searching maximum or minimum in O(1) time complexity.
* Represent hierarchical data.
* Used in editing software like Microsoft Excel and spreadsheets.
* Useful for indexing segmented at the database is useful in storing cache in the system,
* Syntax trees are used for most famous compilers for programming like GCC, and AOCL to perform arithmetic operations.
* For implementing priority queues.
* Used to find elements in less time (binary search tree)
* Used to enable fast memory allocation in computers.
* Used to perform encoding and decoding operations.
* Binary trees can be used to organize and retrieve information from large datasets, such as in inverted index and k-d trees.
* Binary trees can be used to represent the decision-making process of computer-controlled characters in games, such as in decision trees.
* Binary trees can be used to implement searching algorithms, such as in binary search trees which can be used to quickly find an element in a sorted list.
* Binary trees can be used to implement sorting algorithms, such as in heap sort which uses a binary heap to sort elements efficiently.

## ****Binary Tree Traversals:****

Tree Traversal algorithms can be classified broadly into two categories:

* Depth-First Search (DFS) Algorithms
* Breadth-First Search (BFS) Algorithms

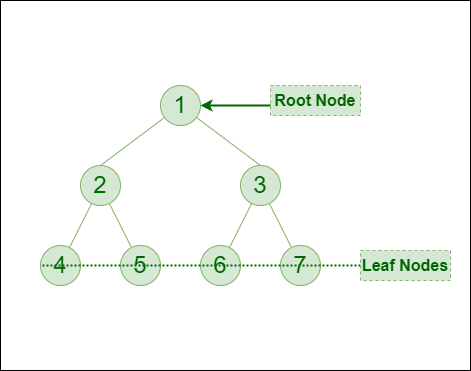
### Tree Traversal using Depth-First Search (DFS) algorithm can be further classified into three categories:

* **Preorder Traversal (current-left-right):** Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.
* **Inorder Traversal (left-current-right):**Visit the current node after visiting all nodes inside the left subtree but before visiting any node within the right subtree. Here, the traversal is left child – root – right child.  It means that the left child is traversed first then its root node and finally the right child.
* **Postorder Traversal (left-right-current):** Visit the current node after visiting all the nodes of the left and right subtrees. Here, the traversal is left child – right child – root.  It means that the left child has traversed first then the right child and finally its root node.

### Tree Traversal using Breadth-First Search (BFS) algorithm can be further classified into one category:

* **Level Order Traversal:** Visit nodes level-by-level and left-to-right fashion at the same level. Here, the traversal is level-wise. It means that the most left child has traversed first and then the other children of the same level from left to right have traversed.

Let us traverse the following tree with all four traversal methods:



*Binary Tree*

**Pre-order Traversal of the above tree:**1-2-4-5-3-6-7  
**In-order Traversal of the above tree:**4-2-5-1-6-3-7  
**Post-order Traversal of the above tree:**4-5-2-6-7-3-1  
**Level-order Traversal of the above tree:** 1-2-3-4-5-6-7

### ****1. The maximum number of nodes at level ‘l’ of a binary tree is 2l****

### ****2. The**** ****Maximum number of nodes in a binary tree of height ‘h’ is 2h – 1****

### ****3. In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is Log2(N+1)****

### ****4. In a Binary tree where every node has 0 or 2 children, the**** ****number of leaf nodes is always one more than nodes with two children****

### ****6. In a non-empty binary tree, if n is the total number of nodes and e is the total number of edges, then e = n-1****

### Some extra properties of binary tree are:

* **Each node in a binary tree can have at most two child nodes:** In a binary tree, each node can have either zero, one, or two child nodes. If a node has zero children, it is called a leaf node. If a node has one child, it is called a unary node. If a node has two children, it is called a binary node.
* **The node at the top of the tree is called the root node:** The root node is the first node in a binary tree and all other nodes are connected to it. All other nodes in the tree are either child nodes or descendant nodes of the root node.
* **Nodes that do not have any child nodes are called leaf nodes:** Leaf nodes are the endpoints of the tree and have no children. They represent the final result of the tree.
* **The height of a binary tree is defined as the number of edges from the root node to the deepest leaf node:** The height of a binary tree is the length of the longest path from the root node to any of the leaf nodes. The height of a binary tree is also known as its depth.
* **In a full binary tree, every node except the leaves has exactly two children**: In a full binary tree, all non-leaf nodes have exactly two children. This means that there are no unary nodes in a full binary tree.
* **In a complete binary tree, every level of the tree is completely filled except for the last level, which can be partially filled:** In a complete binary tree, all levels of the tree except the last level are completely filled. This means that there are no gaps in the tree and all nodes are connected to their parent nodes.
* **In a balanced binary tree, the height of the left and right subtrees of every node differ by at most 1:** In a balanced binary tree, the height of the left and right subtrees of every node is similar. This ensures that the tree is balanced and that the height of the tree is minimized.
* **The in-order, pre-order, and post-order traversal of a binary tree are three common ways to traverse the tree**: In-order, pre-order, and post-order are three different ways to traverse a binary tree. In-order traversal visits the left subtree, the node itself, and then the right subtree. Pre-order traversal visits the node itself, the left subtree, and then the right subtree. Post-order traversal visits the left subtree, the right subtree, and then the node itself.

## Types of Binary Tree based on the number of children:

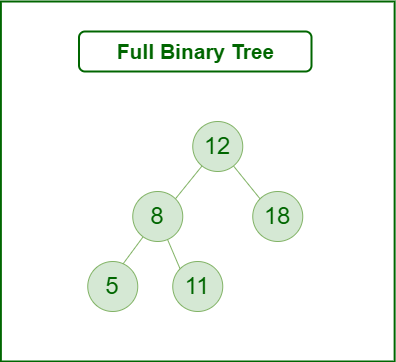
Following are the types of Binary Tree based on the number of children:

1. Full Binary Tree
2. Degenerate Binary Tree
3. Skewed Binary Trees

### ****1. Full Binary Tree****

 A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.



*Full Binary Tree*

### ****2. Degenerate (or pathological) tree****

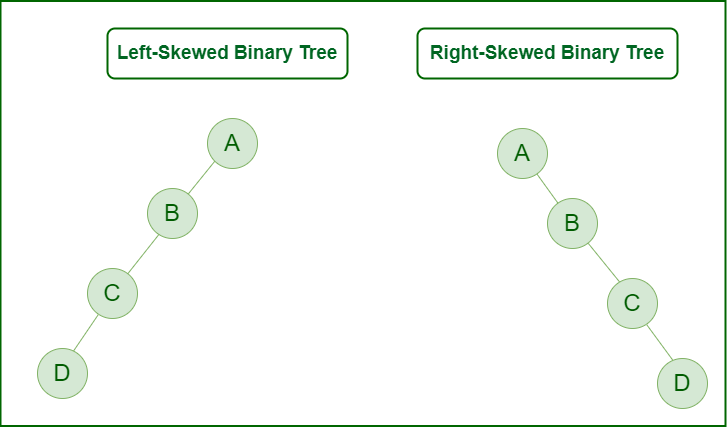
A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.



*Degenerate (or pathological) tree*

### ****3. Skewed Binary Tree****

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



*Skewed Binary Tree*

## Types of Binary Tree ****On the basis of the completion of levels****:

1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

### ****1. Complete Binary Tree****

 A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.

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

* Every level except the last level must be completely filled.
* All the leaf elements must lean towards the left.
* The last leaf element might not have a right sibling i.e. a complete binary tree doesn’t have to be a full binary tree.

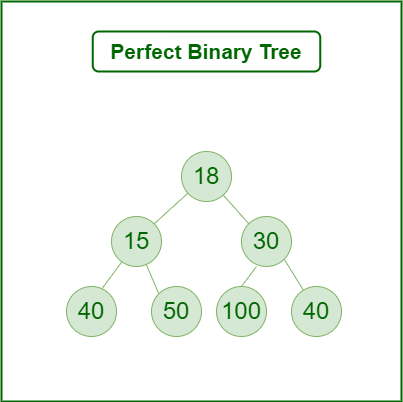


*Complete Binary Tree*

### ****2. Perfect Binary Tree****

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.   
The following are examples of Perfect Binary Trees.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



*Perfect Binary Tree*

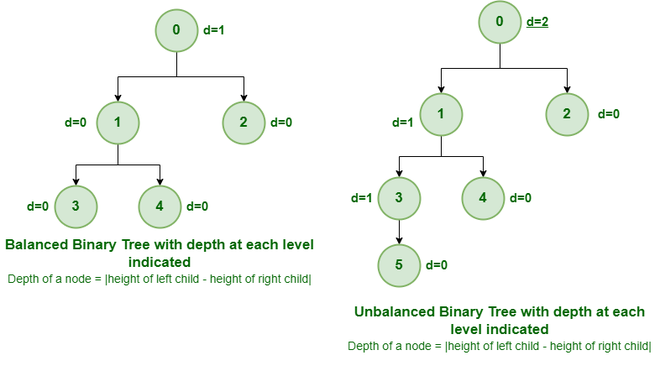
In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

 L = I + 1 Where L = Number of leaf nodes, I = Number of internal nodes.

A Perfect Binary Tree of height h (where the height of the binary tree is the number of edges in the longest path from the root node to any leaf node in the tree, height of root node is 0) has 2h+1 – 1 node.   
An example of a Perfect binary tree is ancestors in the family. Keep a person at root, parents as children, parents of parents as their children.

### ****3. Balanced Binary Tree****

A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths is the same and that there are no adjacent red nodes. Balanced Binary Search trees are performance-wise good as they provide O(log n) time for search, insert and delete.



*Example of Balanced and Unbalanced Binary Tree*

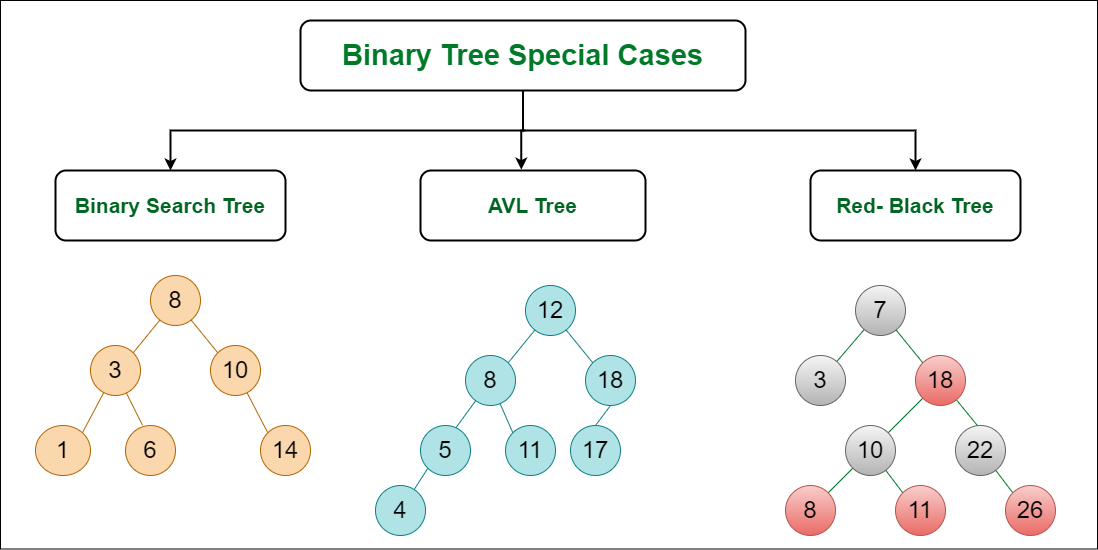
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1. In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

## Some Special Types of Trees:

**On the basis of node values**, the Binary Tree can be classified into the following special types:

1. Binary Search Tree
2. AVL Tree
3. Red Black Tree
4. B Tree
5. B+ Tree
6. Segment Tree

Below Image Shows Important Special cases of binary Trees:

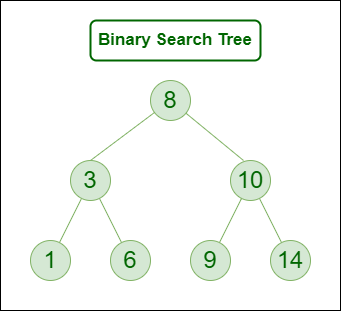


*Binary Tree Special cases*

### ****1. Binary Search Tree****

**Binary Search Tree** is a node-based binary tree data structure that has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree.

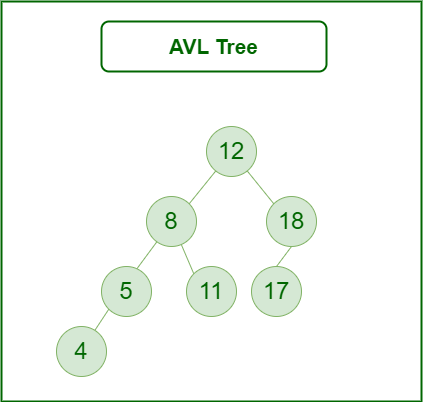


*Binary Search Tree*

### ****2. AVL Tree****

AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

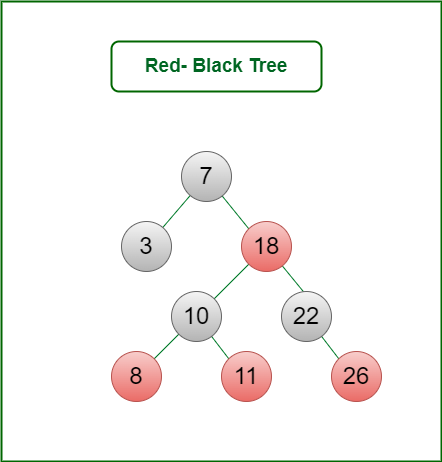
**Example of AVL Tree shown below:**  
The below tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1



*AVL Tree*

### 3. Red Black Tree

A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.



*Red Black Tree*

### 4. B – Tree

A B-tree is a type of self-balancing tree data structure that allows efficient access, insertion, and deletion of data items. B-trees are commonly used in databases and file systems, where they can efficiently store and retrieve large amounts of data. A B-tree is characterized by a fixed maximum degree (or order), which determines the maximum number of child nodes that a parent node can have. Each node in a B-tree can have multiple child nodes and multiple keys, and the keys are used to index and locate data items.

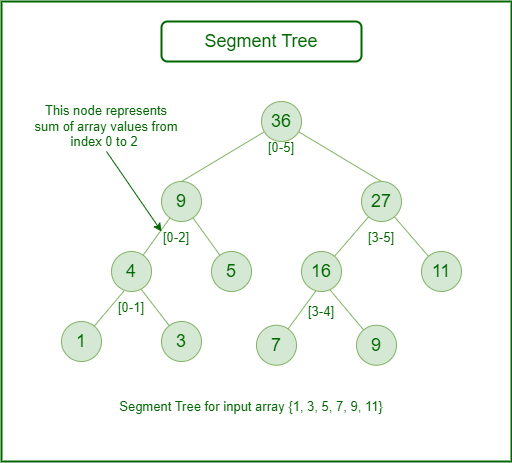
### 5. B+ Tree

A B+ tree is a variation of the B-tree that is optimized for use in file systems and databases. Like a B-tree, a B+ tree also has a fixed maximum degree and allows efficient access, insertion, and deletion of data items. However, in a B+ tree, all data items are stored in the leaf nodes, while the internal nodes only contain keys for indexing and locating the data items. This design allows for faster searches and sequential access of the data items, as all the leaf nodes are linked together in a linked list.

### 6. Segment Tree

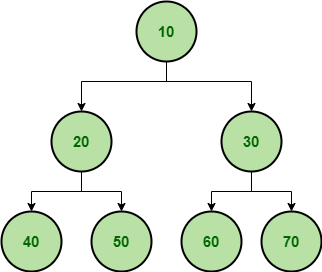
In computer science, a **Segment Tree**, also known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, it’s a structure that cannot be modified once it’s built. A similar data structure is the interval tree.

A **segment tree** for a set I of n intervals uses O(n log n) storage and can be built in O(n log n) time. Segment trees support searching for all the intervals that contain a query point in time O(log n + k), k being the number of retrieved intervals or segments.



*Segment Tree*

A [binary tree](https://www.geeksforgeeks.org/binary-tree-data-structure/) is a tree that has at most two children for any of its nodes. There are several types of binary trees. To learn more about them please refer to the article on “[**Types of binary tree**](https://www.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/)“



*Sample Binary Tree*

**Application of Binary Trees:**

* **Search algorithms:**Binary search algorithms use the structure of binary trees to efficiently search for a specific element. The search can be performed in O(log n) time complexity, where n is the number of nodes in the tree.
* **Sorting algorithms:**Binary trees can be used to implement efficient sorting algorithms, such as binary search tree sort and heap sort.
* **Database systems:** Binary trees can be used to store data in a database system, with each node representing a record. This allows for efficient search operations and enables the database system to handle large amounts of data.
* **File systems:**Binary trees can be used to implement file systems, where each node represents a directory or file. This allows for efficient navigation and searching of the file system.
* **Compression algorithms:** Binary trees can be used to implement Huffman coding, a compression algorithm that assigns variable-length codes to characters based on their frequency of occurrence in the input data.
* **Decision trees:** Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.
* **Game AI**: Binary trees can be used to implement game AI, where each node represents a possible move in the game. The AI algorithm can search the tree to find the best possible move.

**Real-time applications of Binary Trees:**

* DOM in HTML.
* File explorer.
* Used as the basic data structure in Microsoft Excel and spreadsheets.
* Editor tool: Microsoft Excel and spreadsheets.
* Evaluate an expression
* Routing Algorithms

**Advantages of Binary Tree:**

* **Efficient searching**: Binary trees are particularly efficient when searching for a specific element, as each node has at most two child nodes, allowing for binary search algorithms to be used. This means that search operations can be performed in O(log n) time complexity.
* **Ordered traversal:**The structure of binary trees enables them to be traversed in a specific order, such as in-order, pre-order, and post-order. This allows for operations to be performed on the nodes in a specific order, such as printing the nodes in sorted order.
* **Memory efficient**: Compared to other tree structures, binary trees are relatively memory-efficient because they only require two child pointers per node. This means that they can be used to store large amounts of data in memory while still maintaining efficient search operations.
* **Fast insertion and deletion:**Binary trees can be used to perform insertions and deletions in O(log n) time complexity. This makes them a good choice for applications that require dynamic data structures, such as database systems.
* **Easy to implement:** Binary trees are relatively easy to implement and understand, making them a popular choice for a wide range of applications.
* **Useful for sorting:** Binary trees can be used to implement efficient sorting algorithms, such as heap sort and binary search tree sort.

**Disadvantages of Binary Tree:**

* **Limited structure:**Binary trees are limited to two child nodes per node, which can limit their usefulness in certain applications. For example, if a tree requires more than two child nodes per node, a different tree structure may be more suitable.
* **Unbalanced trees:**Unbalanced binary trees, where one subtree is significantly larger than the other, can lead to inefficient search operations. This can occur if the tree is not properly balanced or if data is inserted in a non-random order.
* **Space inefficiency:**Binary trees can be space inefficient when compared to other data structures. This is because each node requires two child pointers, which can be a significant amount of memory overhead for large trees.
* **Slow performance in worst-case scenarios:** In the worst-case scenario, a binary tree can become degenerate, meaning that each node has only one child. In this case, search operations can degrade to O(n) time complexity, where n is the number of nodes in the tree.
* **Complex balancing algorithms:** To ensure that binary trees remain balanced, various balancing algorithms can be used, such as AVL trees and red-black trees. These algorithms can be complex to implement and require additional overhead, making them less suitable for some applications.

**What is Heap Data Structure?**

A **heap** is a binary tree-based data structure that satisfies the heap property: the value of each node is greater than or equal to the value of its children. This property makes sure that the root node contains the **maximum** or **minimum** value (depending on the type of heap), and the values decrease or increase as you move down the tree.

**Types of Heaps**

There are two main types of heaps:

* **Max Heap:** The root node contains the maximum value, and the values decrease as you move down the tree.
* **Min Heap:** The root node contains the minimum value, and the values increase as you move down the tree.

**Heap Operations**

Common heap operations are:

* **Insert**: Adds a new element to the heap while maintaining the heap property.
* **Extract Max/Min:**Removes the maximum or minimum element from the heap and returns it.
* **Heapify**: Converts an arbitrary binary tree into a heap.

**Heap Data Structure Applications**

Heaps have various applications, like:

* Heaps are commonly used to implement priority queues, where elements are retrieved based on their priority (maximum or minimum value).
* Heapsort is a sorting algorithm that uses a heap to sort an array in ascending or descending order.
* Heaps are used in graph algorithms like **Dijkstra’s algorithm**and**Prim’s algorithm** for finding the shortest paths and minimum spanning trees.

## Properties of Heap:

#### Heap has the following Properties:

* **Complete Binary Tree:**A heap tree is a complete binary tree, meaning all levels of the tree are fully filled except possibly the last level, which is filled from left to right. This property ensures that the tree is efficiently represented using an array.
* **Heap Property:**This property ensures that the minimum (or maximum) element is always at the root of the tree according to the heap type.
* **Parent-Child Relationship:** The relationship between a parent node at index **‘i’** and its children is given by the formulas: left child at index**2i+1** and right child at index**2i+2**for 0-based indexing of node numbers.
* **Efficient Insertion and Removal:**Insertion and removal operations in heap trees are efficient. New elements are inserted at the next available position in the bottom-rightmost level, and the heap property is restored by comparing the element with its parent and swapping if necessary. Removal of the root element involves replacing it with the last element and heapifying down.
* **Efficient Access to Extremal Elements:**The minimum or maximum element is always at the root of the heap, allowing constant-time access.

## Operations Supported by Heap:

*Operations supported by****min – heap****and****max – heap****are same. The difference is just that min-heap contains minimum element at root of the tree and max – heap contains maximum element at the root of the tree.*

### ****Heapify:****

It is the process to rearrange the elements to maintain the property of heap data structure. It is done when a certain node creates an imbalance in the heap due to some operations on that node. It takes **O(log N)**to balance the tree.

* For **max-heap,**it**balances** in such a way that the maximum element is the root of that binary tree and
* For **min-heap,**it balances in such a way that the minimum element is the root of that binary tree.

### Insertion:

* If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the **heapify** operation so that it maintains the property of the heap.

This operation also takes **O(logN)** time.

**Examples:**

*Assume initially heap(taking****max-heap****) is as follows*

*8  
        /   \  
     4     5  
   / \  
1   2*

*Now if we insert 10 into the heap  
             8  
        /      \  
      4       5  
   /  \      /  
1    2  10*

*After heapify operation final heap will be look like this  
           10  
         /    \  
      4      8  
   /  \     /  
1    2  5*

### Deletion:

* If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.
* Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operations so that it maintains the property of the heap.

It takes **O(logN)** time.

**Example:**

*Assume initially heap(taking max-heap) is as follows  
           15  
         /   \  
      5     7  
   /  \  
2     3*

*Now if we delete 15 into the heap it will be replaced by leaf node of the tree for temporary.  
           3  
        /   \  
     5     7  
   /      
2*

*After heapify operation final heap will be look like this  
           7  
        /   \  
     5     3  
   /     
2*

### getMax (For max-heap) or getMin (For min-heap):

It finds the maximum element or minimum element for **max-heap** and **min-heap** respectively and as we know minimum and maximum elements will always be the root node itself for min-heap and max-heap respectively. It takes **O(1)** time.

### removeMin or removeMax:

This operation returns and deletes the maximum element and minimum element from the max-heap and min-heap respectively. In short, it deletes the root element of the heap binary tree.

**Implementation of Heap Data Structure:-**

The following code shows the implementation of a **max-heap**.

Let’s understand the **maxHeapify** function in detail:-

**maxHeapify**is the function responsible for restoring the property of the Max Heap. It arranges the node **i**, and its subtrees accordingly so that the heap property is maintained.

1. Suppose we are given an array, **arr[]** representing the complete binary tree. The left and the right child of **ith** node are in indices **2\*i+1** and **2\*i+2**.
2. We set the index of the current element, **i**, as the ‘MAXIMUM’.
3. If **arr[2 \* i + 1] > arr[i]**, i.e., the left child is larger than the current value, it is set as ‘MAXIMUM’.
4. Similarly if **arr[2 \* i + 2] > arr[i]**, i.e., the right child is larger than the current value, it is set as ‘MAXIMUM’.
5. Swap the ‘MAXIMUM’ with the current element.
6. Repeat steps **2 to 5** till the property of the heap is restored.

**Operations of Heap:**

* **Heapify**: a process of creating a heap from an array.
* [**Insertion**](https://www.geeksforgeeks.org/insertion-and-deletion-in-heaps/): process to insert an element in existing heap time complexity O(log N).
* **Deletion**: deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
* **Peek:** to check or find the most prior element in the heap, (max or min element for max and min heap).

**Application of Heap Data Structure:**

* **Priority queues:** The heap data structure is commonly used to implement priority queues, where elements are stored in a heap and ordered based on their priority. This allows constant-time access to the highest-priority element, making it an efficient data structure for managing tasks or events that require prioritization.
* **Heapsort algorithm:**The heap data structure is the basis for the heapsort algorithm, which is an efficient sorting algorithm with a worst-case time complexity of O(n log n). The heapsort algorithm is used in various applications, including database indexing and numerical analysis.
* **Memory management:** The heap data structure is used in memory management systems to allocate and deallocate memory dynamically. The heap is used to store the memory blocks, and the heap data structure is used to efficiently manage the memory blocks and allocate them to programs as needed.
* **Graph algorithms:** The heap data structure is used in various graph algorithms, including Dijkstra’s algorithm, Prim’s algorithm, and Kruskal’s algorithm. These algorithms require efficient priority queue implementation, which can be achieved using the heap data structure.
* **Job scheduling**: The heap data structure is used in job scheduling algorithms, where tasks are scheduled based on their priority or deadline. The heap data structure allows efficient access to the highest-priority task, making it a useful data structure for job scheduling applications.

**Real-Time Application of Heap:**

* Patient treatment: In a hospital, an emergency patient, or the patient with more injury is treated first. Here the priority is the degree of injury.
* Systems concerned with security use heap sort, like the Linux kernel.

**Advantages of Heap Data Structure:**

* **Efficient insertion and deletion:**The heap data structure allows efficient insertion and deletion of elements. When a new element is added to the heap, it is placed at the bottom of the heap and moved up to its correct position using the heapify operation. Similarly, when an element is removed from the heap, it is replaced by the bottom element, and the heap is restructured using the heapify operation.
* **Efficient priority queue:** The heap data structure is commonly used to implement a priority queue, where the highest priority element is always at the top of the heap. The heap allows constant-time access to the highest priority element, making it an efficient data structure for implementing priority queues.
* **Guaranteed access to the maximum or minimum element**: In a max-heap, the top element is always the maximum element, and in a min-heap, the top element is always the minimum element. This provides guaranteed access to the maximum or minimum element in the heap, making it useful in algorithms that require access to the extreme values.
* **Space efficiency:**The heap data structure requires less memory compared to other data structures, such as linked lists or arrays, as it stores elements in a complete binary tree structure.
* **Heap-sort algorithm**: The heap data structure forms the basis for the heap-sort algorithm, which is an efficient sorting algorithm that has a worst-case time complexity of O(n log n).

**Disadvantages of Heap Data Structure:**

* **Lack of flexibility:** The heap data structure is not very flexible, as it is designed to maintain a specific order of elements. This means that it may not be suitable for some applications that require more flexible data structures.
* **Not ideal for searching:**While the heap data structure allows efficient access to the top element, it is not ideal for searching for a specific element in the heap. Searching for an element in a heap requires traversing the entire tree, which has a time complexity of O(n).
* **Not a stable data structure:**The heap data structure is not a stable data structure, which means that the relative order of equal elements may not be preserved when the heap is constructed or modified.
* **Memory management:**The heap data structure requires dynamic memory allocation, which can be a challenge in some systems with limited memory. In addition, managing the memory allocated to the heap can be complex and error-prone.
* **Complexity:** While the heap data structure allows efficient insertion, deletion, and priority queue implementation, it has a worst-case time complexity of O(n log n), which may not be optimal for some applications that require faster algorithms.

### ****Why and when to use heap?****

Heaps are used in a variety of algorithms and data structures to efficiently manage and retrieve elements based on their priority. Some of the main use cases of heaps include:

1. **Priority Queues:** Heaps can be used to implement priority queues, where elements with higher priority are retrieved before elements with lower priority.
2. **Sorting**: Heapsort is a comparison-based sorting algorithm that can sort an array in O(n log n) time.
3. Graph algorithms: Heaps are used in graph algorithms such as Dijkstra’s shortest path algorithm to efficiently find the node with the smallest distance from the source.
4. **Median Maintenance**: Heaps can be used to efficiently maintain the median of a dynamic set of numbers.
5. **Scheduling Tasks:** Heaps can be used to schedule tasks based on their priority in real-time operating systems.
6. **Memory Management**: Heaps are used in memory management to allocate and deallocate memory efficiently.

**What is Trie Data Structure?**

**Trie data structure** is defined as a Tree based data structure that is used for storing some collection of strings and performing efficient search operations on them. The word Trie is derived from re**TRIE**val, which means finding something or obtaining it.

**Trie data structure** follows some property that If two strings have a common prefix then they will have the same ancestor in the trie. A**trie data structure** can be used to sort a collection of strings alphabetically as well as search whether a string with a given prefix is present in the **trie**or not.

**What is need of Trie Data Structure?**

A **Trie data structure** is used for **storing**and **retrieval**of data and the same operations could be done using another data structure which is **Hash Table**but Trie data structure can perform these operations more efficiently than a Hash Table. Moreover, Trie has its own advantage over the Hash table. A Trie data structure can be used for **prefix-based** searching whereas a Hash table can’t be used in the same way.

**Advantages of Trie Data Structure over a Hash Table:**

The A trie data structure has the following advantages over a hash table:

* We can efficiently do **prefix search** (or auto-complete) with Trie.
* We can easily print all words in alphabetical order which is not easily possible with hashing.
* There is no overhead of Hash functions in a Trie data structure.
* Searching for a String even in the large collection of strings in a Trie data structure can be done in **O(L)** Time complexity, Where L is the number of words in the query string. This searching time could be even less than O(L) if the query string does not exist in the trie.

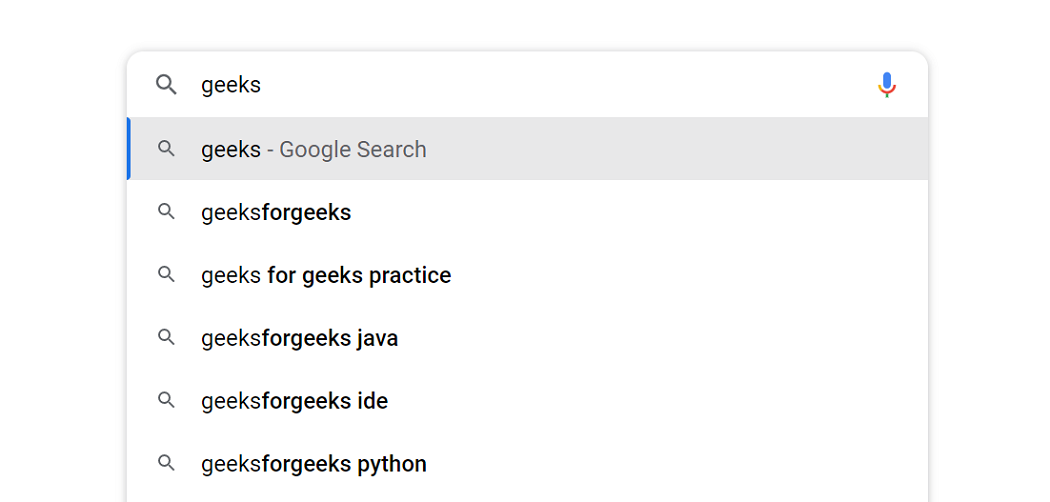
**Properties of a Trie Data Structure**

Below are some important properties of the Trie data structure:

* Each Trie has an empty root node, with links (or references) to other nodes
* Each node of a Trie represents a string and each edge represents a character.
* Every node consists of hashmapsor**an array of pointers**, with each index representing a character and a flag to indicate if any string ends at the current node.
* Trie data structure can contain any number of characters including **alphabets**, **numbers**, and **special characters**. But for this article, we will discuss strings with characters a-z. Therefore, only 26 pointers need for every node, where the**0th** index represents**‘a’** and the **25th** index represents**‘z’** characters.
* Each path from the root to any node represents a word or string.

## [Applications of Trie data structure](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-trie/):

**1. Autocomplete Feature:** Autocomplete provides suggestions based on what you type in the search box. Trie data structure is used to implement autocomplete functionality.



*Autocomplete feature of Trie Data Structure*

**2. Spell Checkers:**If the word typed does not appear in the dictionary, then it shows suggestions based on what you typed.  
**It is a 3-step process that includes** :

1. Checking for the word in the data dictionary.
2. Generating potential suggestions.
3. Sorting the suggestions with higher priority on top.

Trie stores the data dictionary and makes it easier to build an algorithm for searching the word from the dictionary and provides the list of valid words for the suggestion.

**3. Longest Prefix Matching Algorithm(Maximum Prefix Length Match):** This algorithm is used in networking by the routing devices in IP networking. Optimization of network routes requires contiguous masking that bound the complexity of lookup a time to O(n), where n is the length of the URL address in bits.

To speed up the lookup process, Multiple Bit trie schemes were developed that perform the lookups of multiple bits faster.

## [Advantages of Trie data structure](https://www.geeksforgeeks.org/advantages-trie-data-structure/):

* Trie allows us to input and finds strings in O(l) time, where l is the length of a single word. It is faster as compared to both hash tables and binary search trees.
* It provides alphabetical filtering of entries by the key of the node and hence makes it easier to print all words in alphabetical order.
* Trie takes less space when compared to BST because the keys are not explicitly saved instead each key requires just an amortized fixed amount of space to be stored.
* Prefix search/Longest prefix matching can be efficiently done with the help of trie data structure.
* Since trie doesn’t need any hash function for its implementation so they are generally faster than hash tables for small keys like integers and pointers.
* Tries support ordered iteration whereas iteration in a hash table will result in pseudorandom order given by the hash function which is usually more cumbersome.
* Deletion is also a straightforward algorithm with O(l) as its time complexity, where l is the length of the word to be deleted.

## Disadvantages of Trie data structure:

* The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the no of characters in the worst case.
* An efficiently constructed hash table(i.e. a good hash function and a reasonable load factor) has O(1) as lookup time which is way faster than O(l) in the case of a trie, where l is the length of the string.

Pending

Tree – 3 sample workout

Binary tree

Binary search tree- insertion, deletion, three traversal, preorder, postorder, inorder

Find the closest value to given number

Validate a given tree is BST or not

Heap

Create minheap, maxheap with build, insert, remove

Heap sort

Dijkstra’s algorithm, Prim’s algorithm, and Kruskal’s algorithm.

Trie

Concept and sample workouts

Graph

3 sample workout

Graph traversal

Learn all of applications

Extra

Tree types- binary tree, ternary tree, n-ary tree

Completed

Bfs inset, remove, search, traversal – inorder, postorede, preored, level order,

Find height,depth,size,closest value,is bst or not

Min max heap, heap sort insert,remove, traverse subtrees(left,right)