What is a Data Structure?
A data structure is a way of organizing data so that the data can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized for specific tasks. For example, B-trees are particularly well-suited for the implementation of databases, while compiler implementations usually use hash tables to look up identifiers.

What are linear and non-linear data Structures?

- **Linear:** A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array. Linked List, Stacks and Queues.
- **Non-Linear:** A data structure is said to be non-linear if the traversal of nodes is nonlinear in nature. Example: Graph and Trees.

What are the various operations that can be performed on different Data Structures?

- **Insertion** ? Add a new data item in the given collection of data items.
- **Deletion** ? Delete an existing data item from the given collection of data items.
- **Traversal** ? Access each data item exactly once so that it can be processed.
- **Searching** ? Find out the location of the data item if it exists in the given collection of data items.
- **Sorting** ? Arranging the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.

How is an Array different from Linked List?

- The size of the arrays is fixed, and Linked Lists are Dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
- Random access is not allowed on Linked Listed.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have a better cache locality that can make a pretty big difference in performance.

What is Stack and where it can be used?
Stack is a linear data structure in which the order LIFO(Last In First Out) or FILO(First In Last Out) for accessing elements. Basic operations of the stack are: **Push, Pop, Peek**

Applications of Stack:

1. Infix to Postfix Conversion using Stack
2. Evaluation of Postfix Expression
3. Reverse a String using Stack
4. Implement two stacks in an array
5. Check for balanced parentheses in an expression

What is a Queue, how it is different from the stack and how is it implemented?
The queue is a linear structure that follows the order is **F**irst **I**n **F**irst **O**ut (FIFO) to access elements. Mainly the following are basic operations on queue: **Enqueue, Dequeue**, **Front, Rear**
The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added. Both Queues and Stacks can be implemented using Arrays and Linked Lists.

What are Infix, prefix, Postfix notations?

- **Infix notation:** X + Y – Operators are written in between their operands. This is the usual way we write expressions. An expression such as

A * (B + C) / D

- **Postfix notation (also known as "Reverse Polish notation"):** X Y + Operators are written after their operands. The infix expression given above is equivalent to

A B C + * D/

- **Prefix notation (also known as "Polish notation"):** + X Y Operators are written before their operands. The expressions given above are equivalent to

/ * A + B C D

What is a Linked List and What are its types?

A linked list is a linear data structure (like arrays) where each element is a separate object. Each element (that is node) of a list is comprised of two items – the data and a reference to the next node. Types of Linked List :

1. **Singly Linked List :** In this type of linked list, every node stores address or reference of the next node in the list and the last node has next address or reference as NULL. For example 1->2->3->4->NULL
2. **Doubly Linked List :** Here, here are two references associated with each node, One of the reference points to the next node and one to the previous node. Eg. NULL<-1<->2<->3->NULL
3. **Circular Linked List :** Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or a doubly circular linked list. Eg. 1->2->3->1 [The next pointer of the last node is pointing to the first]

Which data structures are used for BFS and DFS of a graph?

- Queue is used for BFS
- Stack is used for DFS.

Can doubly-linked be implemented using a single pointer variable in every node?
A doubly linked list can be implemented using a single pointer.

How to implement a stack using queue?
A stack can be implemented using two queues. Let stack to be implemented be 's' and queues used to implement be 'q1' and 'q2'. Stack 's' can be implemented in two ways:

- Method 1 (By making push operation costly)
- Method 2 (By making pop operation costly)

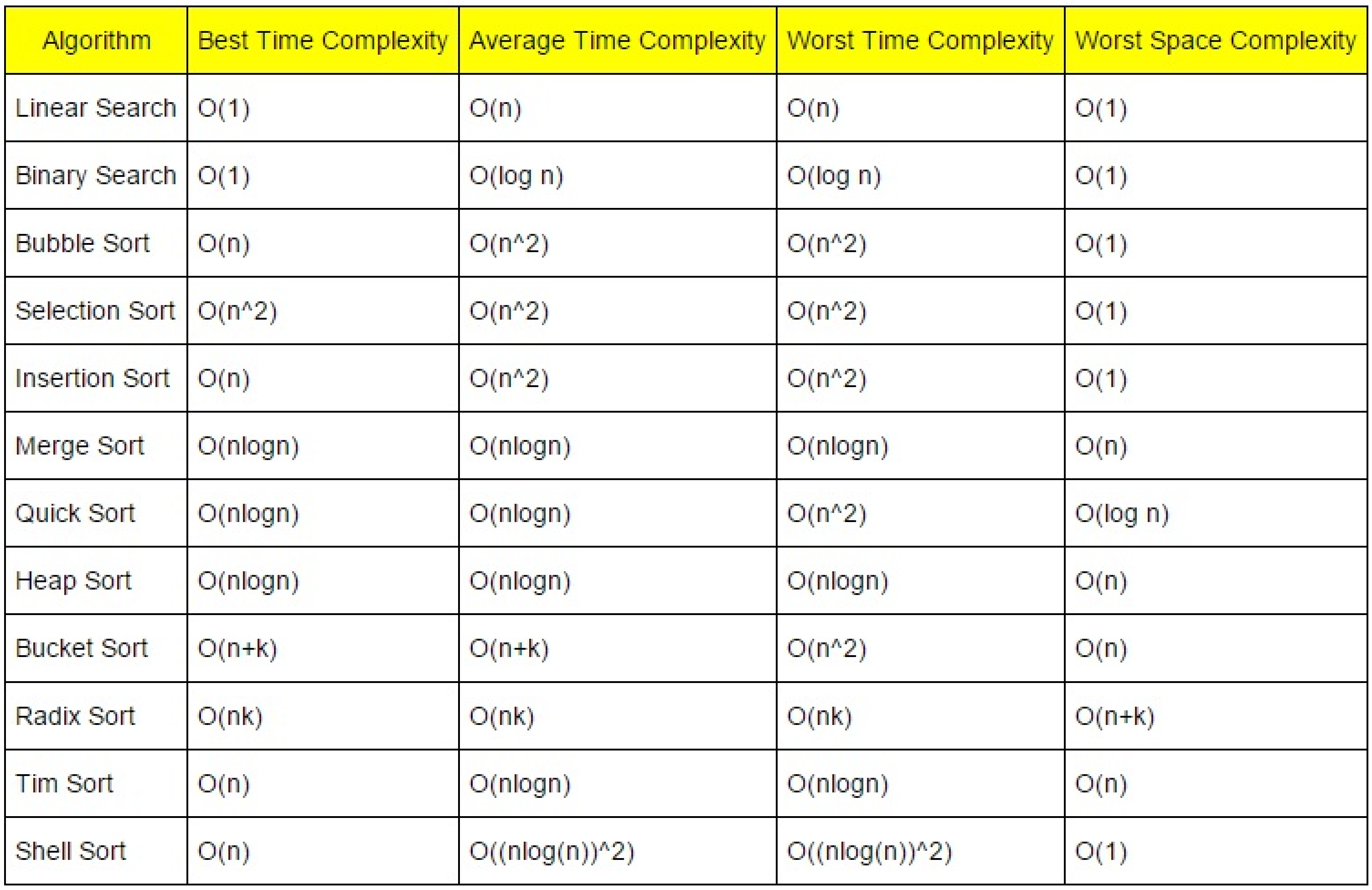
How to implement a queue using a stack?
A queue can be implemented using two stacks. Let the queue to be implemented be q and the stacks used to implement q be stack1 and stack2. q can be implemented in two ways:

- Method 1 (By making enQueue operation costly)
- Method 2 (By making deQueue operation costly)

Which Data Structure Should be used for implementing LRU cache?
We use two data structures to implement an LRU Cache.

1. **Queue** which is implemented using a doubly-linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near the rear end and the least recent pages will be near the front end.
2. **A Hash** with page number as key and address of the corresponding queue node as value.

How to check if a given Binary Tree is BST or not?
If inorder traversal of a binary tree is sorted, then the binary tree is BST. The idea is to simply do inorder traversal and while traversing keep track of previous key value. If current key value is greater, then continue, else return false.



Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

S. No.	**Parameters**	**BFS**	**DFS**
1.	**Stands for**	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	**Data Structure**	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	**Definition**	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
4.	**Technique**	BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
5.	**Conceptual Difference**	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
6.	**Approach used**	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
7.	**Suitable for**	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
8.	**Suitability for Decision-Trees**	BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.
9.	**Time Complexity**	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.

10.	**Visiting of Siblings/ Children**	Here, siblings are visited before the children.	Here, children are visited before the siblings.
11.	**Removal of Traversed Nodes**	Nodes that are traversed several times are deleted from the queue.	The visited nodes are added to the stack and then removed when there are no more nodes to visit.
12.	**Backtracking**	In BFS there is no concept of backtracking.	DFS algorithm is a recursive algorithm that uses the idea of backtracking
13.	**Applications**	BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and topological order etc.
14.	**Memory**	BFS requires more memory.	DFS requires less memory.
15.	**Optimality**	BFS is optimal for finding the shortest path.	DFS is not optimal for finding the shortest path.
16.	**Space complexity**	In BFS, the space complexity is more critical as compared to time complexity.	DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node.
17.	**Speed**	BFS is slow as compared to DFS.	DFS is fast as compared to BFS.
18,	**Tapping in loops**	In BFS, there is no problem of trapping into infinite loops.	In DFS, we may be trapped in infinite loops.
19.	**When to use?**	When the target is close to the source, BFS performs better.	When the target is far from the source, DFS is preferable.

ArrayList	**LinkedList**
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Suitable when you require fast random access and have a relatively stable or infrequent insertion and deletion of elements.	Suitable when frequent insertions and deletions are required, and random access is not a primary concern.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) Iterating over elements is faster, especially when accessing elements sequentially.	Iterating over elements is relatively slower compared to ArrayList.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.
8) Insertion slower compared to LikedList ,Time Complexity : O(N)	Insertion faster than ArrayList ,Time Complexity : O(1)
Same case as above for deletion.	Same case as above for deletion.

Vector v/s HashMap

1. **Purpose:**
   - **Vector:** Also known as a dynamic array, a vector is a resizable array that stores elements in a contiguous memory block. It is suitable for scenarios where you need fast access to elements by index, and the number of elements is expected to change frequently.
   - **HashMap:** A hash map, also known as an associative array or dictionary, is a data structure that allows you to store key-value pairs. It is designed for efficient retrieval of values based on keys.
2. **Access Time:**
   - **Vector:** Provides fast access to elements by index. Retrieving an element by index has a constant time complexity O(1).
   - **HashMap:** Provides fast access to values based on keys. On average, hash map lookups have an O(1) time complexity, but in the worst case, they can have O(n) complexity, depending on collisions and the efficiency of the

hash function.

3. **Memory Allocation:**
   - **Vector:** Uses a contiguous block of memory to store elements. The memory is allocated in a single chunk, and resizing may involve copying elements to a larger block of memory.
   - **HashMap:** Uses a combination of arrays and linked lists (or other collision resolution methods) to store key-value pairs. Memory is allocated dynamically, and resizing is generally less frequent compared to vectors.
4. **Flexibility:**
   - **Vector:** Well-suited for scenarios where the number of elements changes frequently, and you need efficient random access to elements.
   - **HashMap:** Ideal for scenarios where you need to associate keys with values and require fast lookup based on keys. It is especially useful when the number of elements is large, and you want to avoid linear searches.
5. **Duplicates:**
   - **Vector:** Allows duplicate elements, and you can have multiple occurrences of the same value at different indices.
   - **HashMap:** Typically does not allow duplicate keys. If you attempt to insert a key that already exists, the existing value may be overwritten.

ArrayList	**HashMap**
ArrayList implements the List interface.	HashMap implements the Map interface.
ArrayList stores element's value and maintains the indexes for each element.	HashMap stores elements **key** & value pair. For each value, there must be a key associated with HashMap.
ArrayList stores only a single object.	HashMap stores elements in Key and value pairs.
We get the element by specifying the index of it in ArrayList.	The elements are being fetched by the corresponding Key in HashMap.
The ArrayList maintains the order of the objects they are inserted.	HashMap does not provide a guarantee of the order in which they are inserted.
ArrayList allows duplicate elements.	HashMap allows duplicate values but does not allow duplicate keys.
The ArrayList always gives **O(1) performance** in best case or worst-case time complexity.	The HashMap get() method has **O(1)** time complexity in the best case and **O(n)** time complexity in worst case.
ArrayList has any number of null elements.	HashMap allows only one null Key and lots of null values.
ArrayList is the index-based data structure supported by the array.	While HashMap is a mapped data structure that works on hashing to obtain stored values.

The similarity between ArrayList and HashMap:

1. Both are not synchronized; We cannot use them in a multi-threading environment without any external synchronization.
2. Both ArrayList and HashMap Iterator are fail-fast. It throws ConcurrentModificationException and detects the structural changes when the iterator is created in **ArrayList** or **HashMap**.
3. Allows both ArrayList and HashMap null. HashMap allows null keys and values.
4. Both uses the **get()** method. The **ArrayList.get()** method works based on the index and **HashMap.get()** method takes a parameter key_element of object type and refers to the key which associated value is fetched. Therefore, they both provides constant-time performance.
5. An array supports ArrayList. Similarly, HashMap is also internally implemented by array.
6. We use **Iterator** for accessing the elements of ArrayList and HashMap.

Basic	**HashSet**	**HashMap**
Implements	Set interface	Map interface
Duplicates	No	Yes duplicates values are allowed but no duplicate key is allowed
Dummy values	Yes	No
Objects required during an add operation	1	2

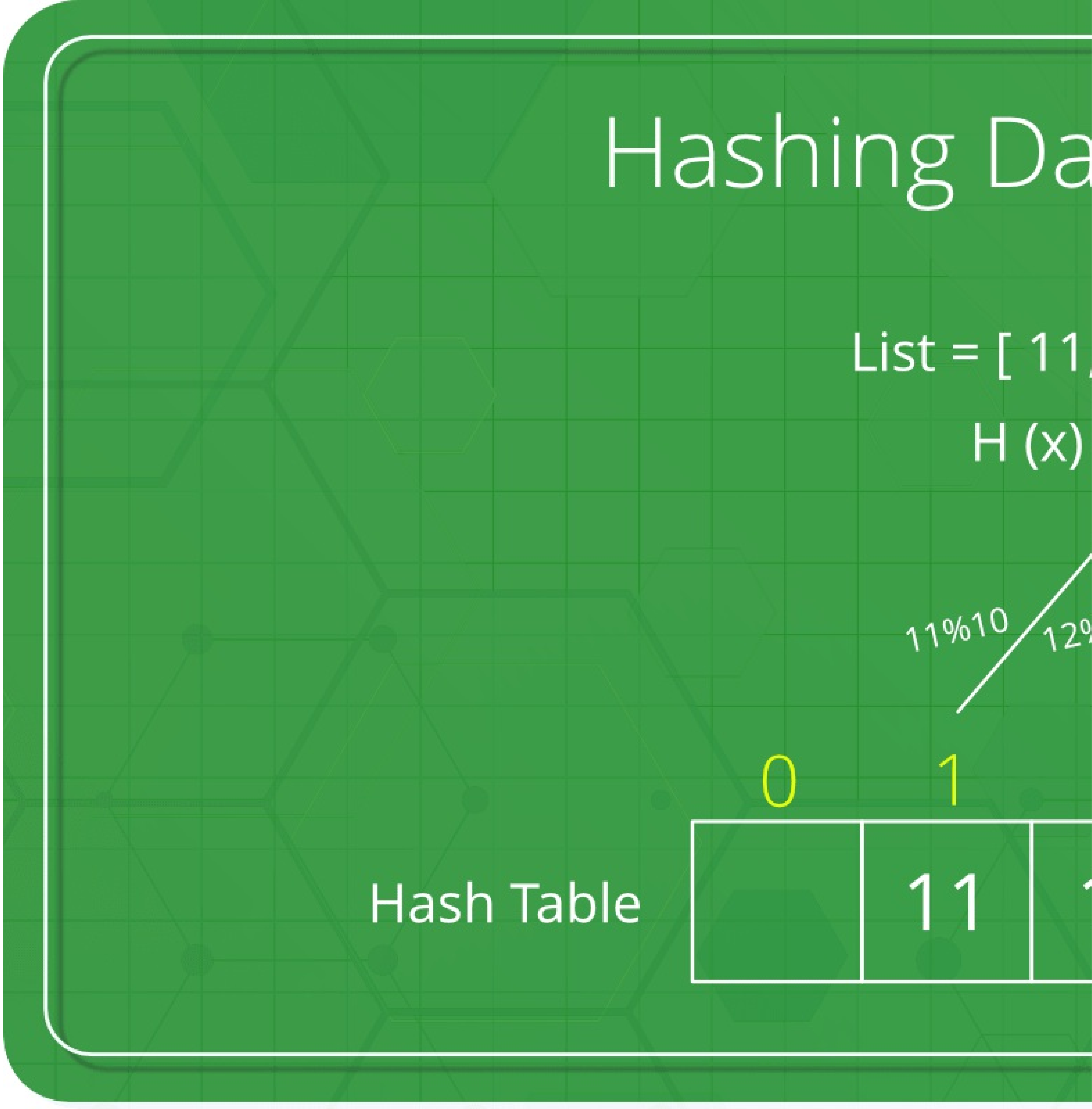
Adding and storing mechanism	HashMap object	Hashing technique
Speed	It is comparatively slower than HashMap	It is comparatively faster than HashSet because of hashing technique has been used here.
Null	Have a single null value	Single null key and any number of null values
Insertion Method	Only one value is required for the insertion process. Add() function is used for insertion	Two values are required for the insertion process. Put() function is used for insertion.
Data storage	The data is stored as objects.	The data is stored as key-value pair.
Complexity	O(n)	O(1)

What is Hashing?

Hashing is a technique used in data structures to quickly find or store data by using a key. It involves the following steps:

1. **Key**: Each data item has a unique identifier called a key.
2. **Hash Function**: A hash function takes a key and converts it into a hash code, typically an integer. This hash code determines the index where the data is stored in a hash table.
3. **Hash Table**: A hash table is an array-like structure that stores data at the index calculated by the hash function.

It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.



Hashing Data Structure

Hashing is a technique used in computer science to quickly organize and retrieve data. It involves a hash function, which takes input data (keys) and converts it into a fixed-size code (hash code). This hash code is used as an index to store or retrieve the corresponding data in a data structure called a hash table.

The goal is to efficiently locate information without going through every piece of data.

What is Hash Table?

A Hash table is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on the hashing concept, where each key is translated by a hash function into a distinct index in an array. The index functions as a storage location for the matching value. In simple words, it maps the keys with the value.

What is Load factor?

A hash table's load factor is determined by how many elements are kept there in relation to how big the table is. The table may be cluttered and have longer search times and collisions if the load factor is high. An ideal load factor can be maintained with the use of a good hash function and proper table resizing.

What is a Hash function?

A Function that translates keys to array indices is known as a hash function. The keys should be evenly distributed across the array via a decent hash function to reduce collisions and ensure quick lookup speeds.

The hash function takes input data, such as keys, and converts it into a fixed-size code called a hash code. This hash code is used as an index in the array to store or retrieve the associated data.

- **Integer universe assumption:** The keys are assumed to be integers within a certain range according to the integer universe assumption. This enables the use of basic hashing operations like division or multiplication hashing.
- **Hashing by division:** This straightforward hashing technique uses the key's remaining value after dividing it by the array's size as the index. When an array size is a prime number and the keys are evenly spaced out, it performs well.
- **Hashing by multiplication:** This straightforward hashing operation multiplies the key by a constant between 0 and 1 before taking the fractional portion of the outcome. After that, the index is determined by multiplying the fractional component by the array's size. Also, it functions effectively when the keys are scattered equally.

Collision resolution techniques**:**

Collisions happen when two or more keys point to the same array index. Chaining, open addressing, and double hashing are a few techniques for resolving collisions.

- **Open addressing:** collisions are handled by looking for the following empty space in the table. If the first slot is already taken, the hash function is applied to the subsequent slots until one is left empty. There are various ways to use this approach, including double hashing, linear probing, and quadratic probing.
- **Separate Chaining:** In separate chaining, a linked list of objects that hash to each slot in the hash table is present. Two keys are included in the linked list if they hash to the same slot. This method is rather simple to use and can manage several collisions.
- **Robin Hood hashing:** To reduce the length of the chain, collisions in Robin Hood hashing are addressed by switching off keys. The algorithm compares the distance between the slot and the occupied slot of the two keys if a new key hashes to an already-occupied slot. The existing key gets swapped out with the new one if it is closer to its ideal slot. This brings the existing key closer to its ideal slot. This method has a tendency to cut down on collisions and average chain length.

Array v/s Vector

1. **Memory Allocation:**
   - **Array:** In many programming languages, arrays have a fixed size determined at the time of declaration. Memory for the entire array is allocated at once, and the size cannot be changed during runtime. If you need a larger array, you often have to declare a new one and copy the elements from the old array.
   - **Vector:** Vectors are dynamic arrays that can resize themselves during runtime. They automatically manage memory allocation, making them more flexible than arrays when the number of elements changes.
2. **Size and Capacity:**
   - **Array:** The size of the array is fixed and determined at the time of declaration. The array's capacity remains constant throughout its lifetime.
   - **Vector:** Vectors can grow or shrink dynamically. They have both a size (the number of elements currently in the vector) and a capacity (the maximum number of elements the vector can hold without resizing).
3. **Resizing:**
   - **Array:** Resizing an array typically involves allocating a new block of memory, copying the existing elements, and deallocating the old memory.
   - **Vector:** Vectors automatically handle resizing. When the number of elements exceeds the current capacity, vectors allocate a larger memory block, copy the existing elements, and update their capacity.
4. **Ease of Use:**
   - **Array:** Arrays are simple and straightforward. They provide constant-time access to elements by index.
   - **Vector:** Vectors provide dynamic resizing and additional functionalities, such as push_back(), pop_back(), and

various utility functions, making them more convenient for dynamic scenarios.

5. **Standard Libraries:**
    - **Array:** Arrays are often part of the core syntax of programming languages and may have limited functionalities.
    - **Vector:** Vectors are typically part of standard libraries in many programming languages and provide a rich set of operations for manipulating and accessing elements.
6. **Static vs. Dynamic:**
    - **Array:** Arrays are considered static data structures due to their fixed size.
    - **Vector:** Vectors are dynamic data structures because they can grow or shrink as needed.

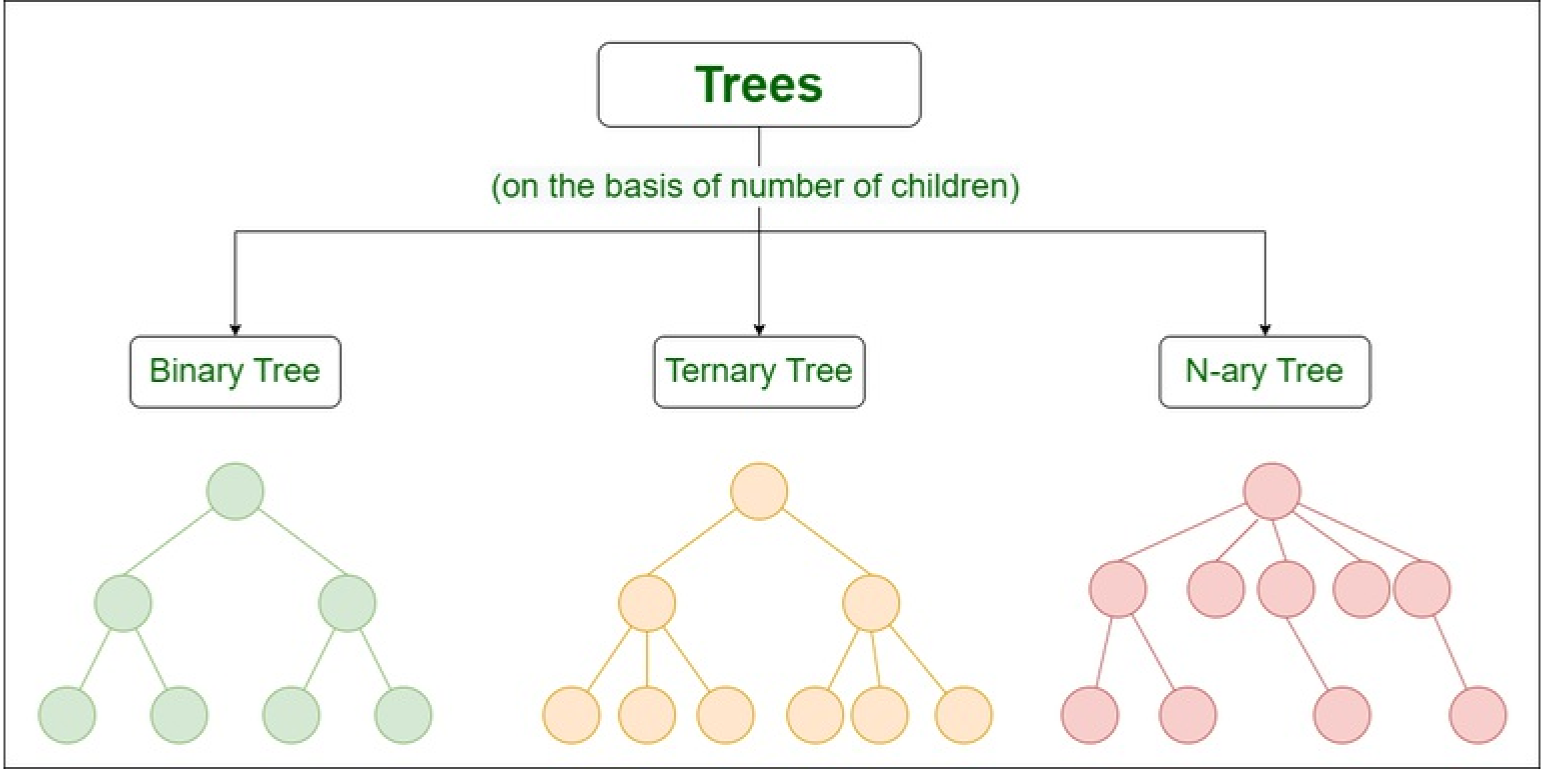
Basis	**Array**	**ArrayList**
Definition	An **array** is a dynamically-created object. It serves as a container that holds the constant number of values of the same type. It has a contiguous memory location.	The **ArrayList** is a class of Java **Collections** framework. It contains popular classes like **Vector, HashTable**, and **HashMap**.
Static/ Dynamic	Array is **static** in size.	ArrayList is **dynamic** in size.
Resizable	An array is a **fixed-length** data structure.	ArrayList is a **variable-length** data structure. It can be resized itself when needed.
Initialization	It is mandatory to provide the size of an array while initializing it directly or indirectly.	We can create an instance of ArrayList without specifying its size. Java creates ArrayList of default size.
Performance	It performs **fast** in comparison to ArrayList because of fixed size.	ArrayList is internally backed by the array in Java. The resize operation in ArrayList slows down the performance.
Primitive/ Generic type	An array can store both **objects** and **primitives** type.	We cannot store **primitive** type in ArrayList. It automatically converts primitive type to object.
Iterating Values	We use **for** loop or **for each** loop to iterate over an array.	We use an **iterator** to iterate over ArrayList.
Type-Safety	We cannot use generics along with array because it is not a convertible type of array.	ArrayList allows us to store only **generic/ type, that's why it is type-safe.**
Length	Array provides a **length** variable which denotes the length of an array.	ArrayList provides the **size()** method to determine the size of ArrayList.
Adding Elements	We can add elements in an array by using the **assignment** operator.	Java provides the **add()** method to add elements in the ArrayList.
Single/ Multi-Dimensional	Array can be **multi-dimensional**.	ArrayList is always **single-dimensional**.

What is a Tree?

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.

Types of Trees in Data Structure based on the number of children:



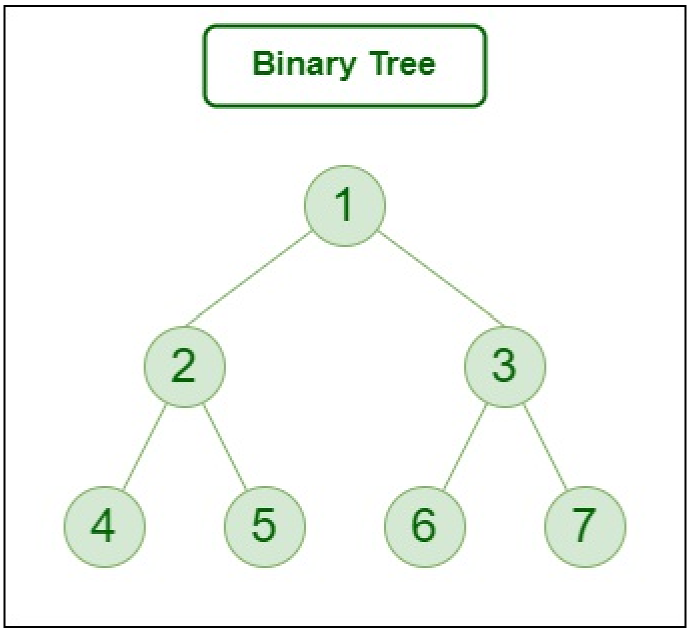
Types of Trees in Data Structure based on the number of children

Let us see about these trees one by one:

1. Binary Tree

A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Example:
Consider the tree below. Since each node of this tree has only 2 children, it can be said that this tree is a Binary Tree



Binary Tree

Types of Binary Tree:

- Binary Tree consists of following types **based on the number of children**:
  1. Full Binary Tree
  2. Degenerate Binary Tree
- **On the basis of completion of levels**, the binary tree can be divided into following types:
  1. Complete Binary Tree

2. Perfect Binary Tree
3. Balanced Binary Tree

Types of Binary Tree based on the number of children:

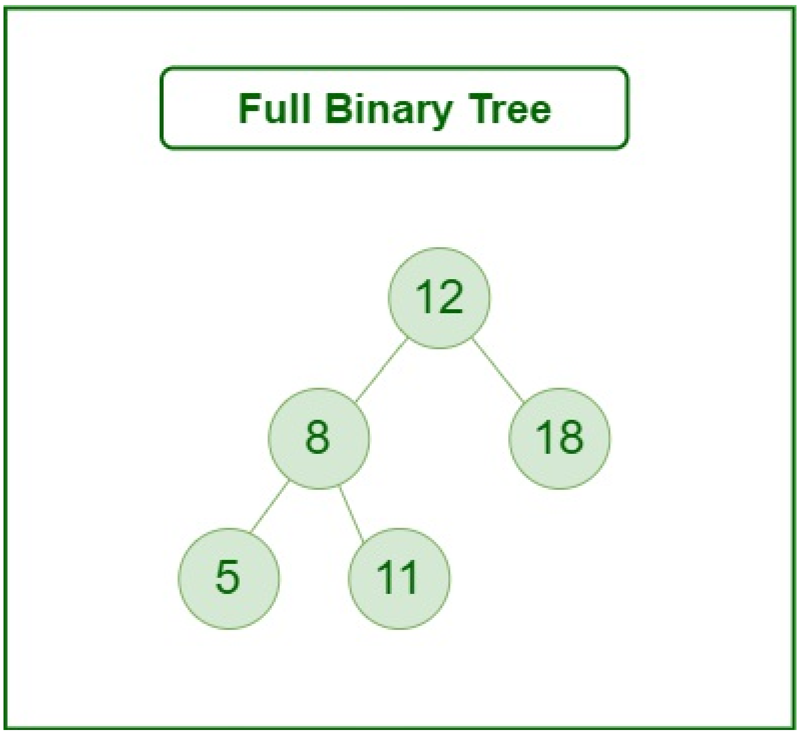
Following are the types of Binary Tree based on the number of children:

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.

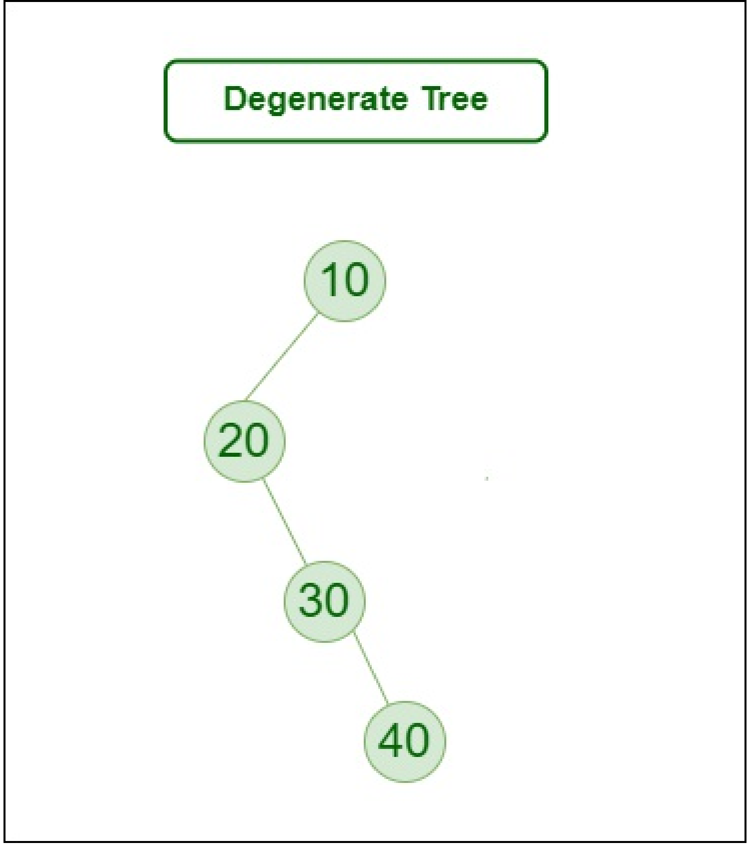
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. It is also known as a proper binary tree.



Full Binary Tree

2. Degenerate (or pathological) tree

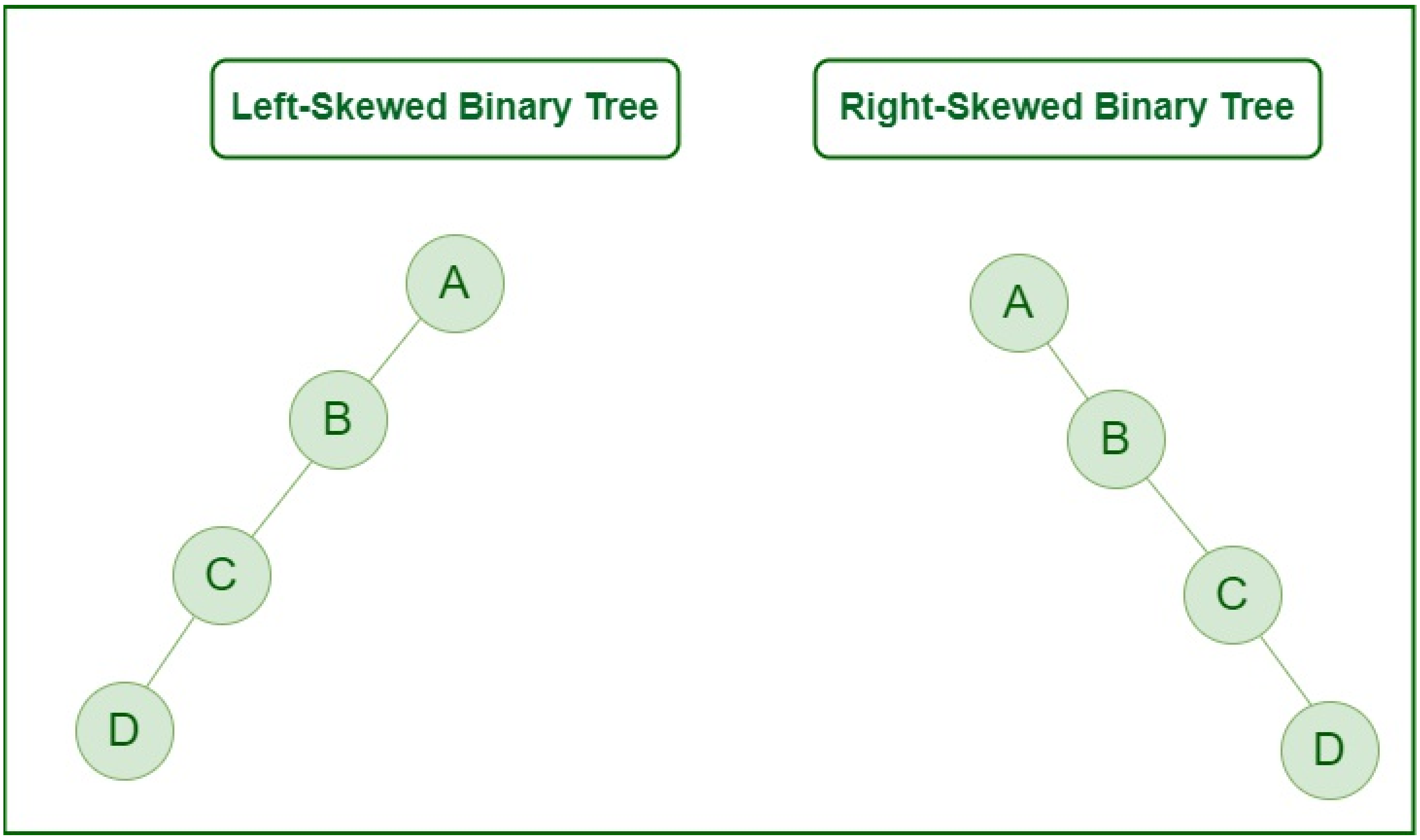
A Tree where every internal node has one child. Such trees are performance-wise same as linked list. A degenerate or pathological tree is a tree having a single child either left or right.



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

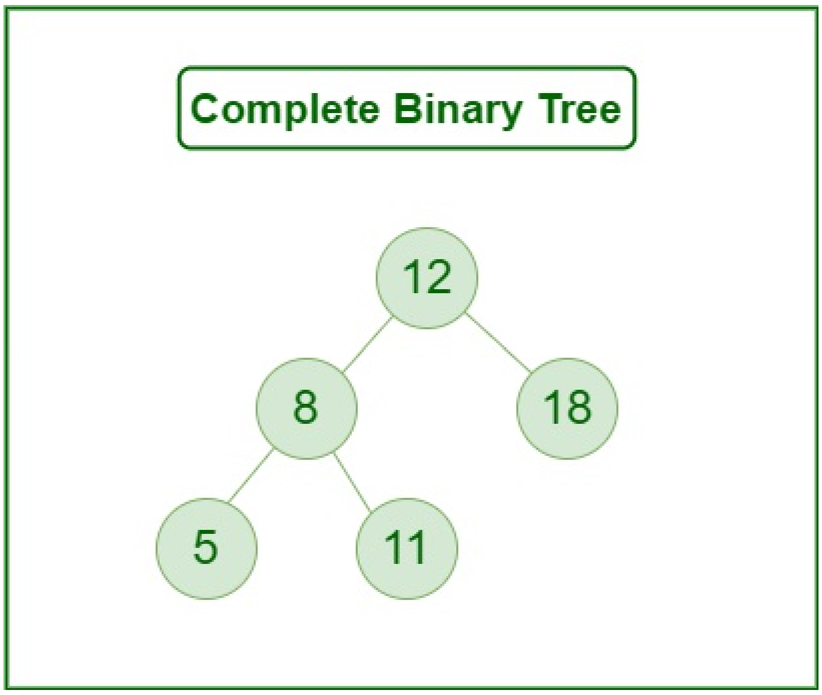
3. Balanced Binary Tree

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.

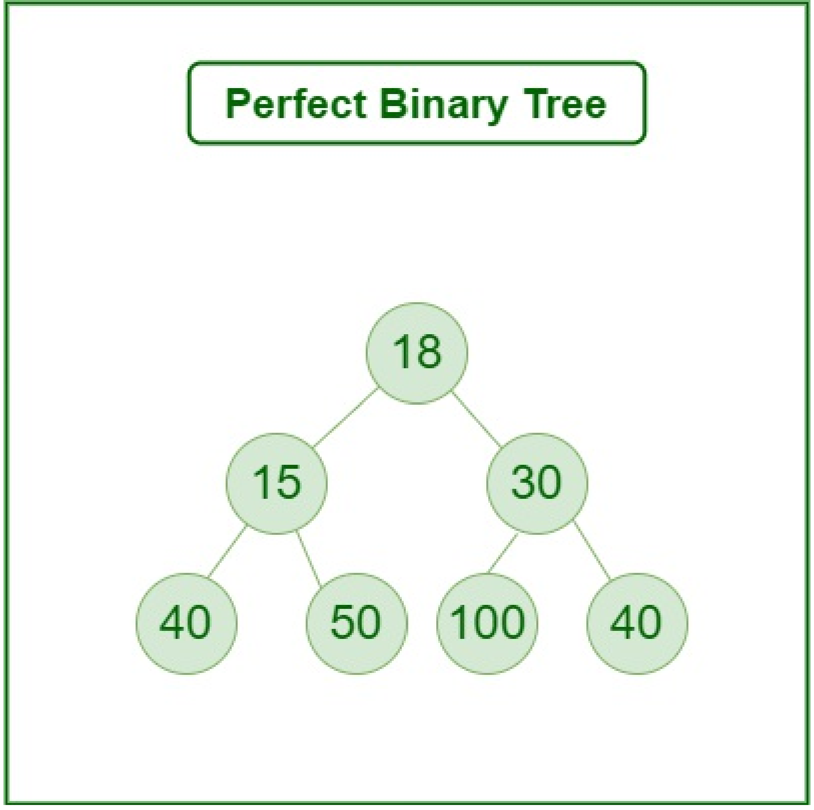


Complete Binary Tree

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

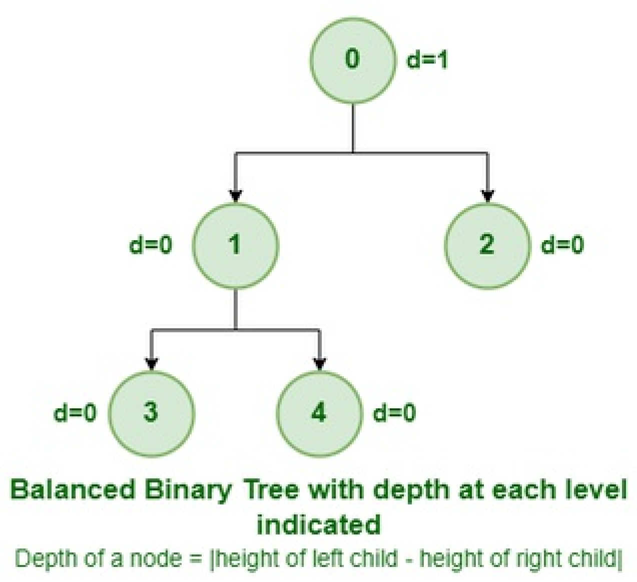
In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

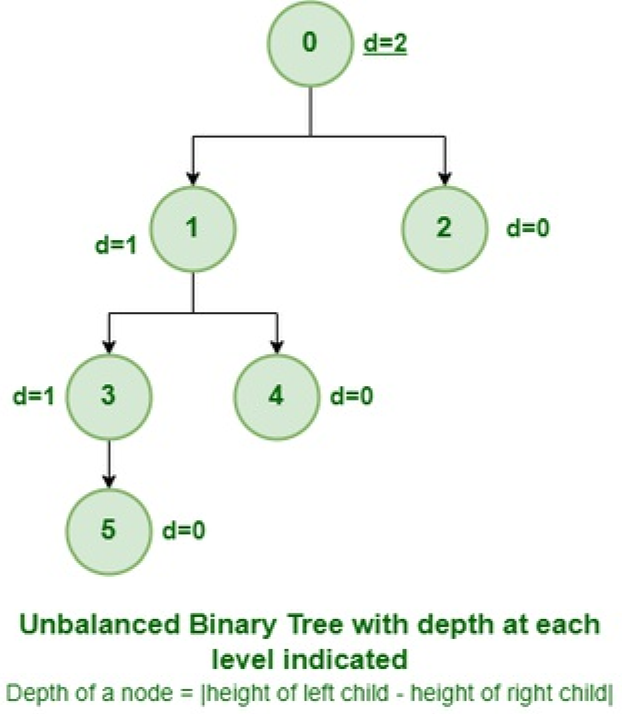
L = I + 1 Where L = Number of leaf nodes, I = Number of internal nodes.

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.


0
d=1
1
d=0
2
d=0
3
d=0
4
d=0
Balanced Binary Tree with depth at each level indicated
Depth of a node = |height of left child - height of right child|


0
d=2
1
d=1
2
d=0
3
d=1
4
d=0
5
d=0
Unbalanced Binary Tree with depth at each level indicated
Depth of a node = |height of left child - height of right child|

Example of Balanced and Unbalanced Binary Tree

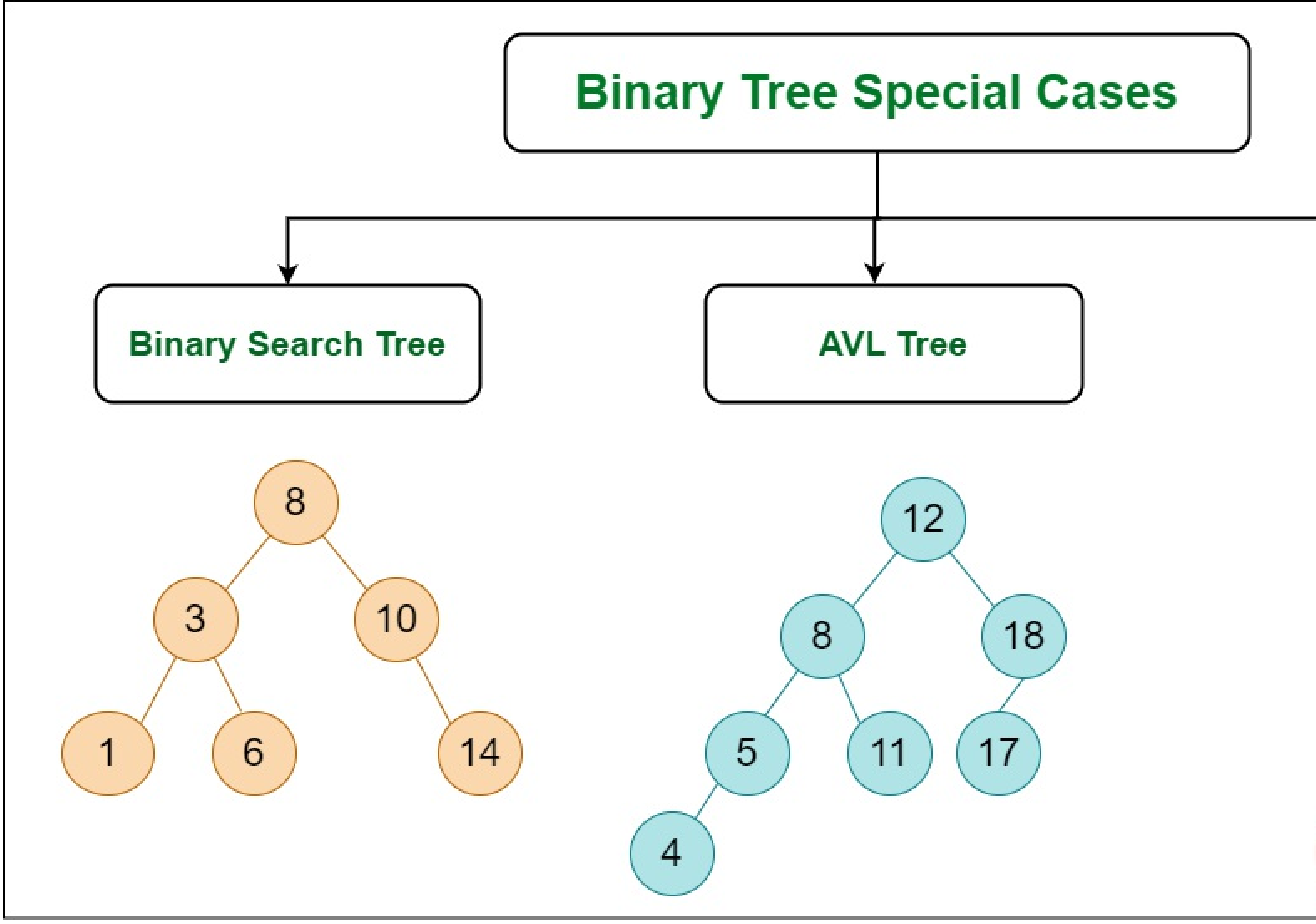
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1. In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

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:

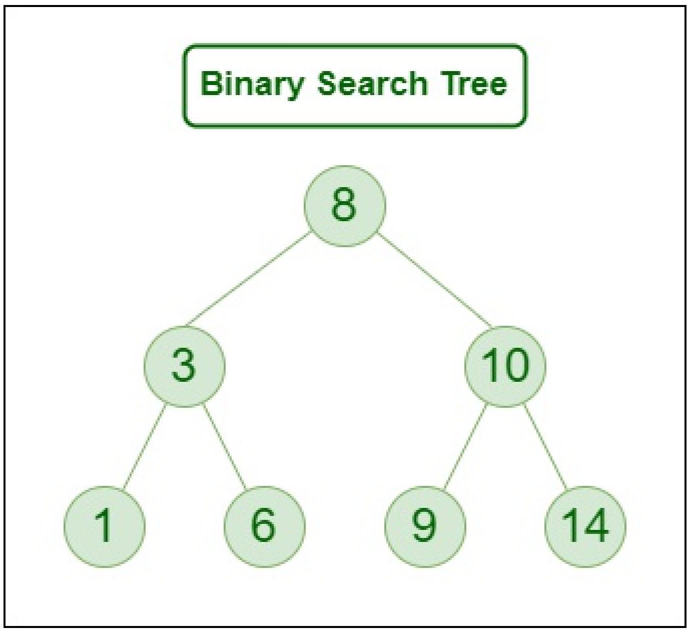


Binary Tree Special cases

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.

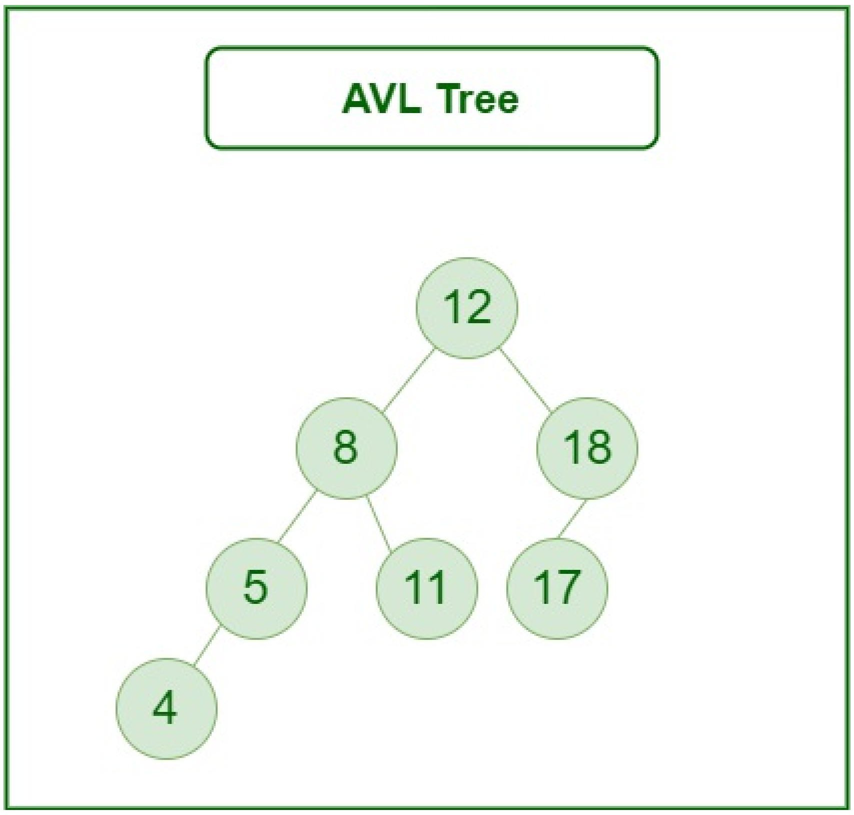


Binary Search Tree

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.

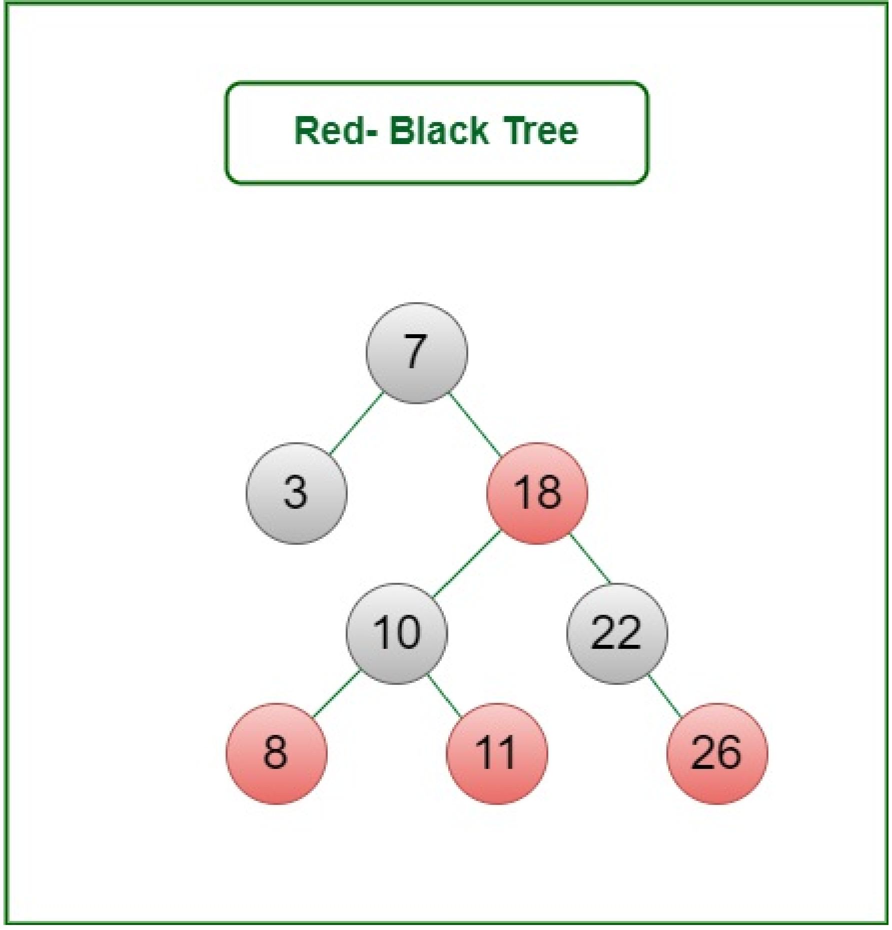
Example of AVL Tree shown below:
The below tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1



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.

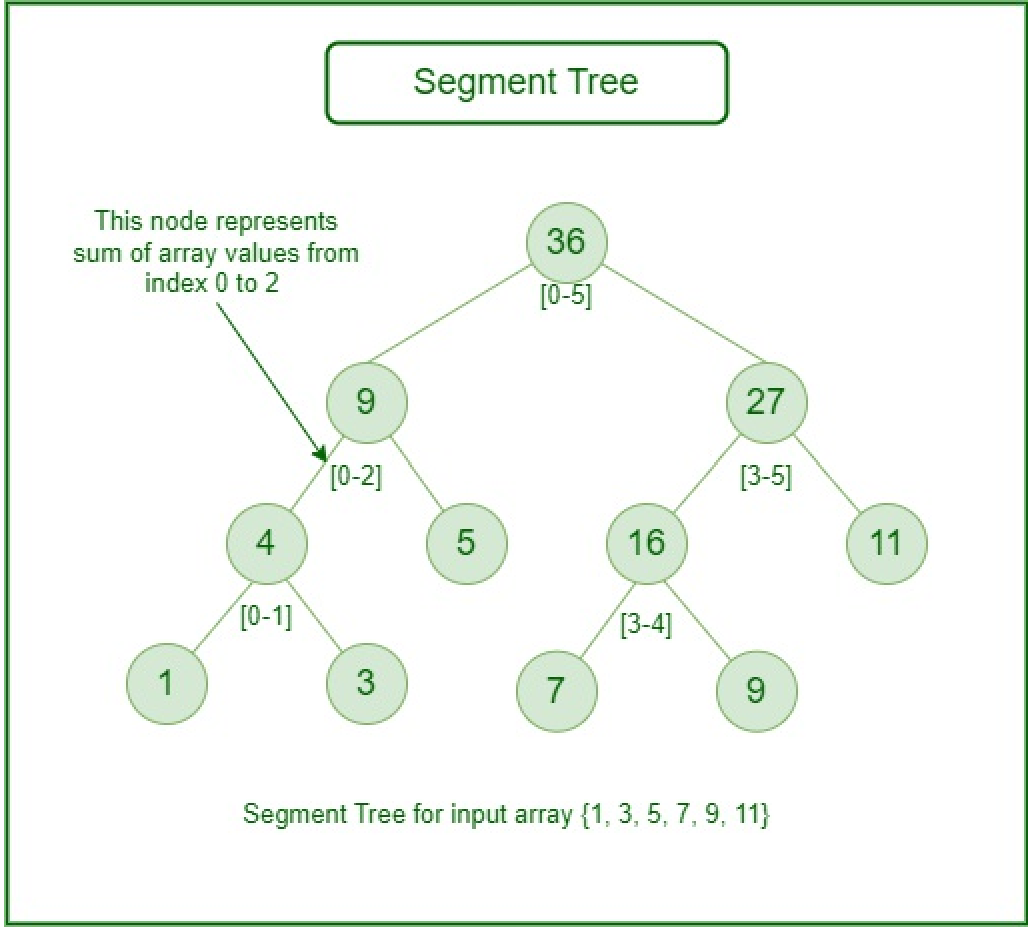
5. B+ Tree

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
This node represents sum of array values from index 0 to 2
36
[0-5]
9
27
[0-2]
[3-5]
4
5
16
11
[0-1]
[3-4]
1
3
7
9
Segment Tree for input array {1, 3, 5, 7, 9, 11}

Segment Tree

Searching Techniques:

1. **Linear Search:**
   - A simple method where each element in the array is checked until the target element is found.
   - Time Complexity: O(n)
2. **Binary Search:**
   - Applicable only to sorted arrays. It repeatedly divides the search interval in half.
   - Time Complexity: O(log n)
3. **Hashing:**
   - Utilizes a hash function to map keys to indices, providing constant time average search.
   - Common data structures include hash tables and hash maps.

https://www.geeksforgeeks.org/searching-algorithms/

Sorting Techniques:

1. **Bubble Sort:**
   - Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.
   - Time Complexity: O(n^2)
2. **Selection Sort:**
   - Divides the input list into two parts: the sorted and the unsorted. It repeatedly selects the smallest (or largest) element from the unsorted part and swaps it with the first element of the unsorted part.
   - Time Complexity: O(n^2)
3. **Insertion Sort:**
   - Builds the sorted array one item at a time by repeatedly taking the next element and inserting it into the already sorted part of the array.
   - Time Complexity: O(n^2)
4. **Merge Sort:**
   - A divide and conquer algorithm that divides the array into two halves, sorts them, and then merges them.
   - Time Complexity: O(n log n)
5. **Quick Sort:**

   - Another divide and conquer algorithm that picks an element as a pivot and partitions the array around the pivot.
   - Time Complexity: O(n log n) on average; O(n^2) worst case
6. **Heap Sort:**
   - Builds a binary heap and uses it to sort the elements.
   - Time Complexity: O(n log n)
7. **Radix Sort:**
   - A non-comparative sorting algorithm that works by distributing elements into buckets according to their individual digits.
   - Time Complexity: O(k * n), where k is the number of digits

https://www.geeksforgeeks.org/introduction-to-sorting-algorithm/

Linked List

A linked list is a data structure consisting of a sequence of elements, where each element (called a node) contains two parts:

1. Data: The value stored in the node.
2. Pointer: A reference (or link) to the next node in the sequence.

There are various types of linked lists:

- **Singly Linked List:** Each node points to the next node.
- **Doubly Linked List:** Each node points to both the next and the previous node.
- **Circular Linked List:** The last node points back to the first node, forming a circle.

Insertion:

- At the Beginning**:** Create a new node and point its next reference to the current head of the list. Update the head to this new node.
- At the End**:** Traverse the list to find the last node, then create a new node and set the last node's next reference to this new node.
- At a Given Position**:** Traverse the list to find the node after which the new node should be inserted, then update references accordingly.

Stack

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The element that is added last is the one to be removed first.

Insertion (Push):

- To insert an element in a stack, place it on the top. Update the top reference to this new element.

Real-Life Application:

- **Undo Mechanism in Text Editors:** The most recent action is undone first.
- **Browser History:** The most recently visited page is the first to be popped.

Technical Application:

- **Function Call Management in Recursion:** The stack keeps track of active function calls.

Queue

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. The element that is added first is the one to be removed first.

Insertion (Enqueue):

- To insert an element in a queue, add it to the end. Update the end reference to this new element.

Real-Life Application:

- **Line at a Ticket Counter:** The person who comes first is served first.
- **Print Queue:** The document that was sent to the printer first is printed first.

Technical Application:

- **Breadth-First Search (BFS) in Graphs:** Nodes are processed in the order they are discovered.
- **Task Scheduling:** Jobs are processed in the order they arrive.

Comparison and Summary

Linked List

- **Applications:**
  - Dynamic memory allocation
  - Implementation of other data structures like stacks, queues, and graphs
  - Manipulation of polynomials

Stack

- **Applications:**
  - Expression evaluation and syntax parsing
  - Depth-first search in graph algorithms
  - Reversing data

Queue

- **Applications:**
  - Managing tasks in operating systems (e.g., CPU scheduling)
  - Handling requests in web servers
  - Breadth-first search in graph algorithms