DSA 3

a avl tree implementaion - https://medium.com/cracking-the-coding-interview-in-ruby-python-and/avl-trees-explained-in-javascript-mastering-javascript-38bb37181c3e

- https://learnersbucket.com/tutorials/data-structures/avl-tree-in-javascript/
- https://dev.to/igorok/javascript-red-black-tree-4703

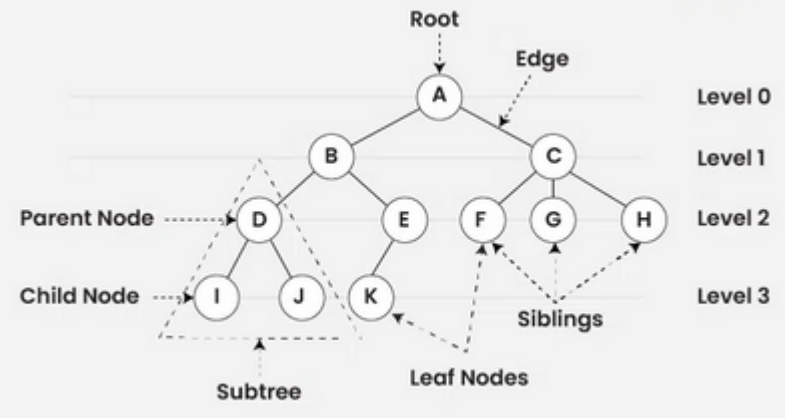
Tree

It is a hierarchical data structure that consists of nodeds connected by edges

Tree data structure is a **hierarchical structure** that is used to represent and organize data in the form of parent child relationship

- each node stores a data value
- non linear data structure so it has quicker and and easier access time than linear ones
- no loops or cycles
- used in file system , family tree , dom , chatbots , abstract syntax trees, html dom
  Basic operations on a stree DS

Terminologies



- **Parent Node**: The node which is an immediate 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. {I, J, K, F, G, H} 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 .
- **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.

Properties of Trees

- **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.

Advantages of Tree:

- **Efficient searching**: Trees are particularly efficient for searching and retrieving data. The time complexity of searching in a tree is $O(\ log\, n)$ in AVL and Red Black Trees. This is better than arrays and linked list but not as good as hashing, but the advantages the these trees provide are sorted data, search for floor and ceiling of data.
  - In binary search tree if it is balanced $O(logn)$(best case) if it is skewed $O(n)$(worst case)
  - balanced tree - log n
  - Trie - O(m) where m is the length of the searched key
  - heap - O(n)
- **Fast insertion and deletion**: Inserting and deleting nodes in a self balancing binary search trees like AVL and Red Black can be done in $O(log\, n)$ time. This is again better than arrays and linked

list not as good as hashing, but the advantages the these trees provide are sorted data, search for floor and ceiling of data.

    - binary search tree - insertion best case O(1) , insertino average case O(log n) (balanced tree), worst case n (skewed) , delteion best case (O(log n)), worst case O(n)(skewed)
    - heap - logn
    - trie - O(m)
- 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.
- Natural organization: Trees have a natural hierarchical organization that can be used to represent many types of relationships. This makes them particularly useful for representing things like file systems, organizational structures, and taxonomies.
- Flexible size: Unlike Arrays, trees can easily grow or shrink dynamically depending on the number of nodes that are added or removed. This makes them particularly useful for applications where the data size may change over time.

Disadvantages of Tree:

- **Memory overhead**: Trees can require a significant amount of memory to store, especially if they are very large. This can be a problem for applications that have limited memory resources.
- **Imbalanced trees**: If a tree is not balanced, it can result in uneven search times. This can be a problem in applications where speed is critical.
- **Complexity**: Unlike Arrays and Linked Lists, Trees can be complex data structures, and they can be difficult to understand and implement correctly. This can be a problem for developers who are not familiar with them.
- Search, Insert and Delete Times: If a problem requires only search, insert and delete, not other operations like sorted data traversal, floor, and ceiling, Hash Tables always beat Self Balancing Binary Search Trees.

Types of Trees based on the number of childs

- **Binary Tree** - In a binary tree, each node can have a maximum of two children linked to it.
  on the basis of number of childs binary trees are classified as
    - Full Binary Tree
    - Degenerate Binary Tree
      On the basis of completion of levels binary trees are classified as
    - Complete Binray Tree
    - Perfect Binary Tree
    - Balanced Binary Tree

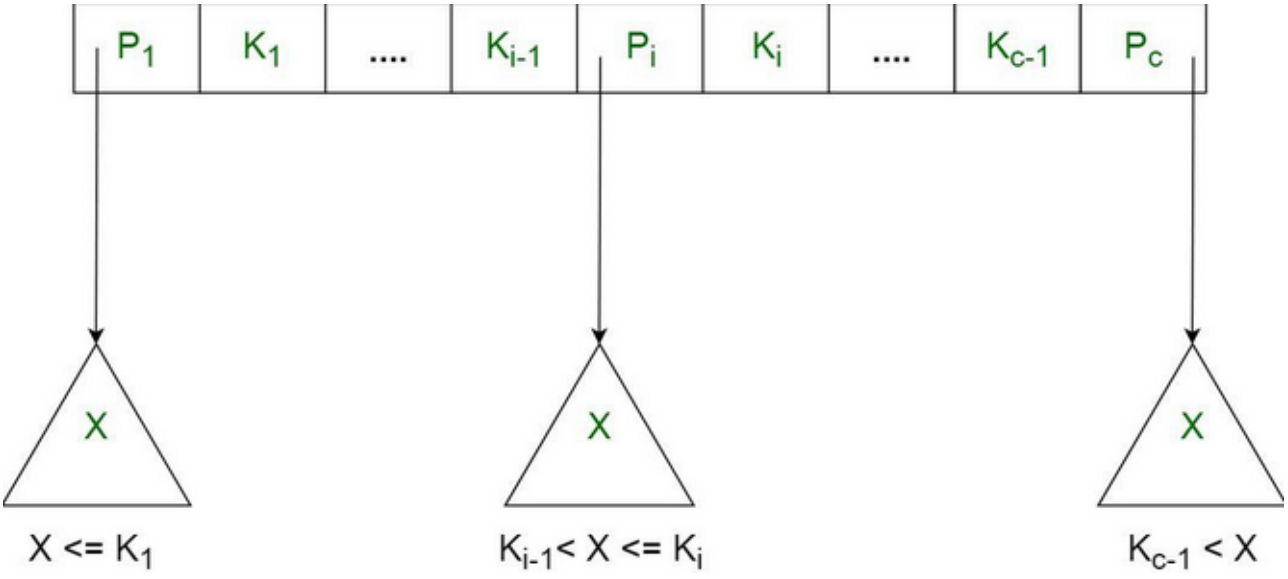
- **Ternary Tree**- A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”
    - Ternary Search Tree - A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.
- **N-ary Tree or Generic Tree**- 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.

Types of Trees based on the node's values

- **Binary Search Tree** - node based binary tree ds with 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.
- **AVL Tree** - AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.
- **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.
    - Every node has a color either red or black.
    - The root of the tree is always black.
    - There are no two adjacent red nodes (A red node cannot have a red parent or red child).
    - Every path from a node (including root) to any of its descendants’ NULL nodes has the same number of black nodes.
    - All leaf (NULL) nodes are black nodes.
- **B-Tree** - B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in the main memory.
    - All leaves are at the same level.
    - B-Tree is defined by the term minimum degree ‘t‘. The value of ‘t‘ depends upon disk block size.
    - Every node except the root must contain at least t-1 keys. The root may contain a minimum of 1 key.
    - All nodes (including root) may contain at most $(2 * t–1)$ keys.
    - The number of children of a node is equal to the number of keys in it plus 1.
    - All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
    - B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

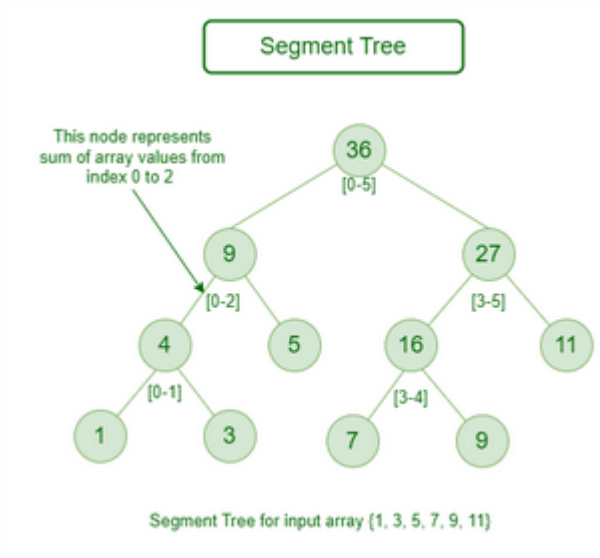
- Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
  Insertion of a Node in B-Tree happens only at Leaf Node.

- **B+ tree** - B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree.
  - Each internal node is of the form: $< P_1, K_1, P_2, K_2, ..., P_{c-1}, K_{c-1}, P_c >$ where $c <= a$ and each P_i is a tree pointer (i.e points to another node of the tree) and, each Ki is a key-value (see diagram-I for reference).
  - Every internal node has : $K_1 < K_2 < .... < K_c - 1$
    For each search field values 'X' in the sub-tree pointed at by Pi, the following condition holds: $K_i - 1 < X <= K_i$, for $1 < i < c$ and, $K_i - 1 < X$, for $i = c$ (See diagram I for reference)
  - Each internal node has at most 'a' tree pointers.
  - The root node has, at least two tree pointers, while the other internal nodes have at least \ceil $(\frac{a}{2})$ tree pointers each.
  - If an internal node has 'c' pointers, $c <= a$, then it has 'c–1' key values.



- **Segment Tree** - 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.



Tree Traversal algorithms

- **Depth-First Search** (DFS): Visits nodes depth-wise (Pre-order, In-order, Post-order), using a stack.
  - Explores as deep as possible, useful in topological sorting and cycle detection
- **Breadth-First Search** (BFS): Visits nodes level by level, using a queue.
  - Explores all neighbors first (shortest path in unweighted graphs).

To Check

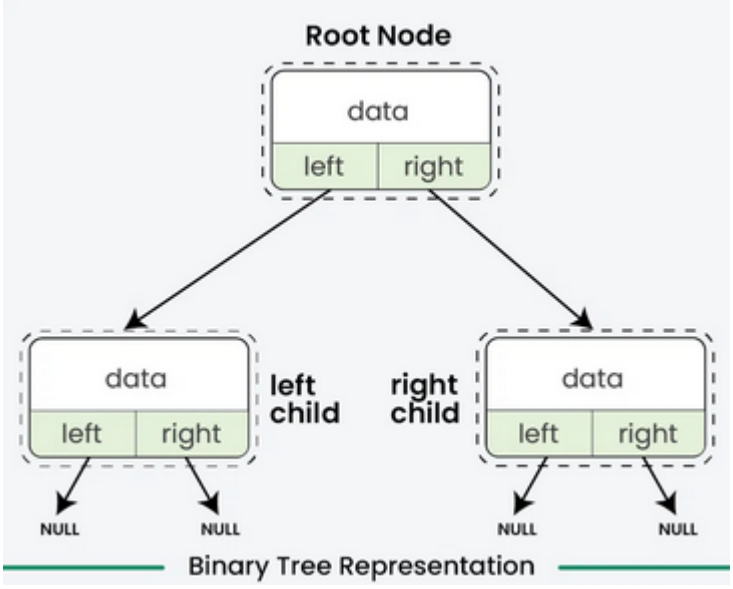
- spanning tree
- balanced tree
- self balancing tree
- tree traversal algorithms
- comparitson of bfs and dfs
- minimum spanning tree
- level order traversal

Practicals

- [x] Implement a tree structure with multiple children per node.

Binary Tree

A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child.



Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child

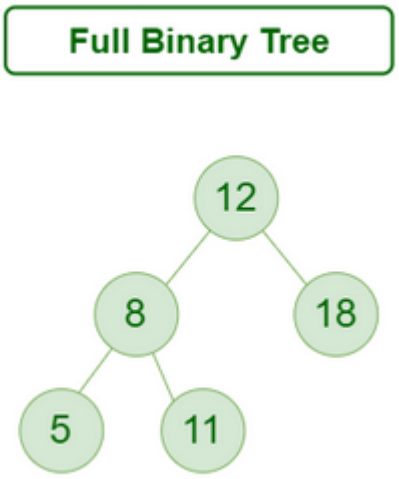
Properties of Binary tree

- The maximum number of nodes at level L of a binary tree is 2^L
- The maximum number of nodes in a binary tree of height H is $2^{H+1}–1$
- minimum number of nodes in a binary tree of height H is $H - 1$
- Total number of leaf nodes in a binary tree = total number of nodes with 2 children + 1
- In a Binary Tree with N nodes, the minimum possible height or the minimum number of levels is $Log_2(N + 1)$
- A Binary Tree with L leaves has at least $\lfloor Log2L \rfloor + 1$ levels

Types of Binary Tree based on the number of children

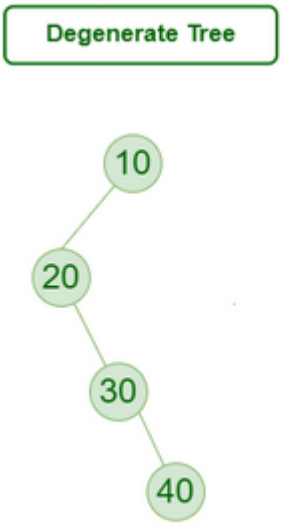
- **Full/Proper/Strict Binary Tree** - A Binary Tree is a full binary tree if every node has **0 or 2 children**. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two

children.



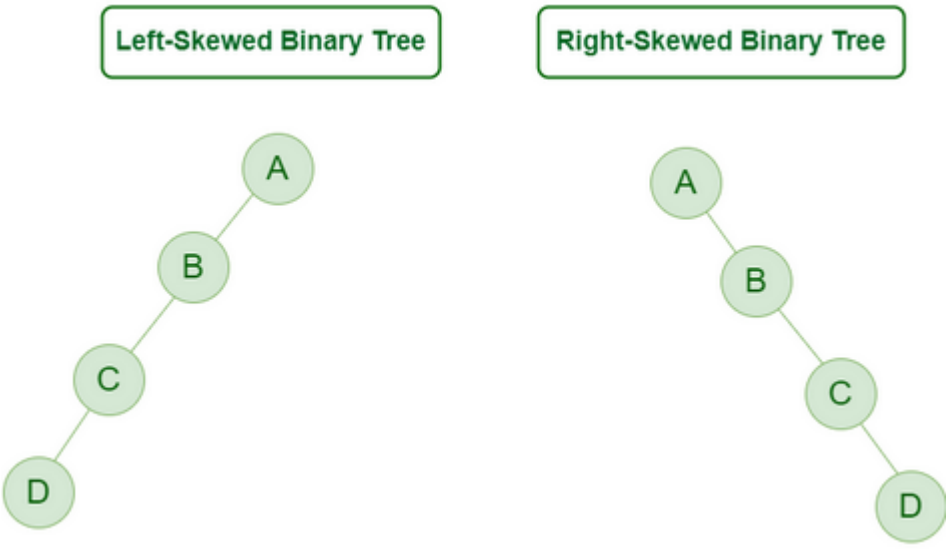
  - no of leaf node = no of internal node + 1
  - minimum number of nodes for height h =- $2h + 1$

- **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.



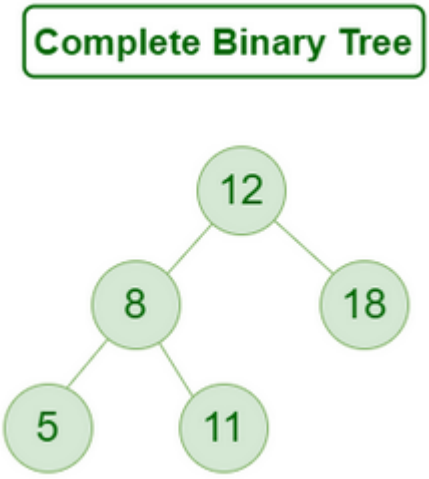
- **Skewed Binary Tree**- A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary

tree: left-skewed binary tree and right-skewed binary tree.

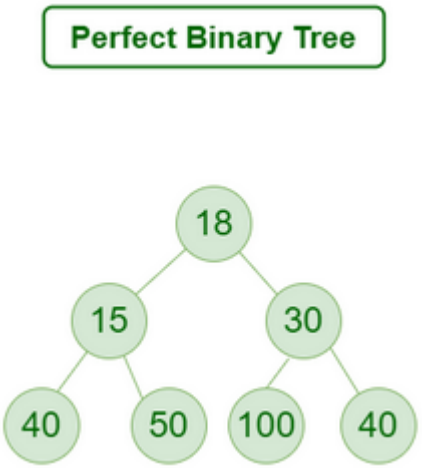


Types of Binary Tree On the basis of the completion of levels

- **Full Binary Tree**: A binary tree in which every node has either 0 or 2 children.
- **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.



  - 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.
  - minimum number of nodes at height H is 2^H
- **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.



- Perfect bniary tree is full and complete
  In a Perfect Binary Tree, the number of leaf nodes is the number of internal nodes plus 1

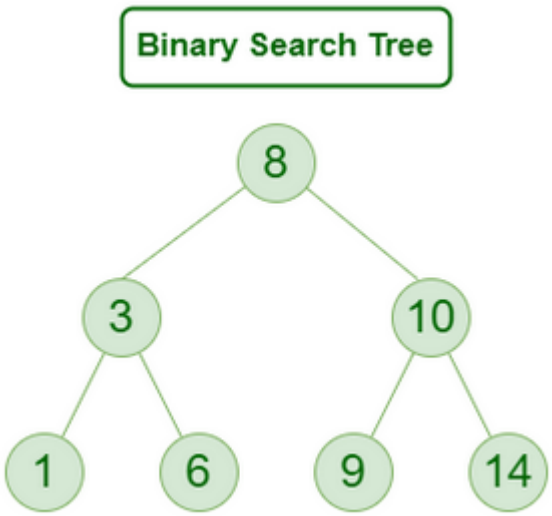
$L = I + 1$ Where L = Number of leaf nodes, I = Number of internal nodes.

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 $2^{h+1}-1$ 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.

- **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.
- A **self-balancing binary tree** is a binary search tree (BST) that automatically maintains a balanced structure to ensure efficient operations like insertion, deletion, and search. The main idea is to keep the tree height close to log(n) to prevent performance degradation.

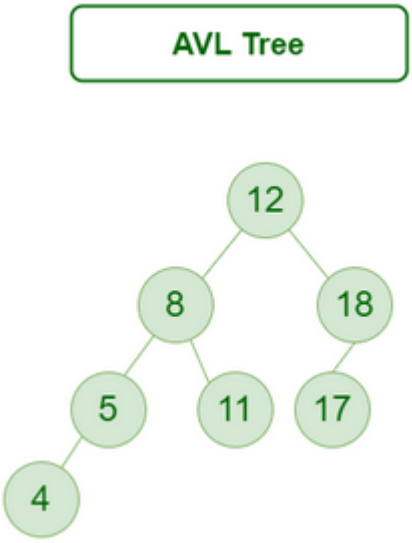
Types of Binary Tree On the basis of Node values

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



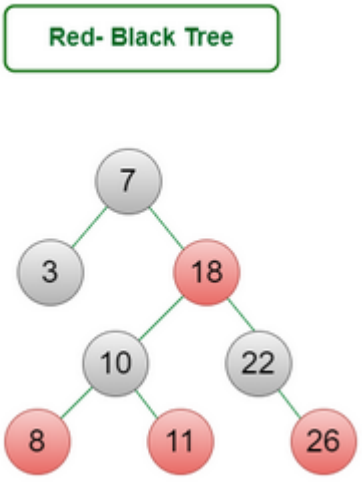
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

- **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.



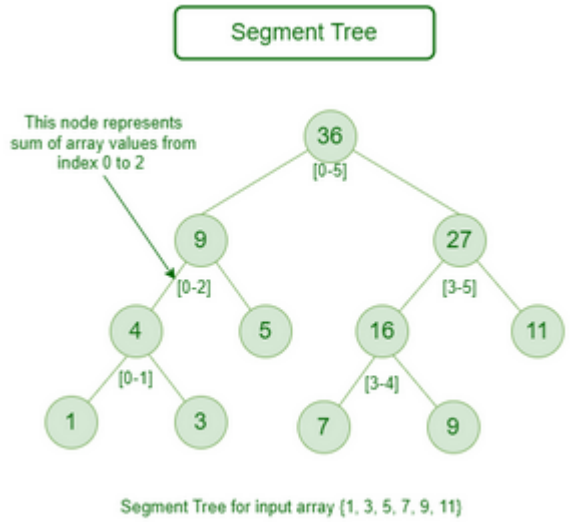
- **Red Black Tree** - A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions. Although the balance 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.
  - only maximum two roations are needed to make it balanced in avl it may take so may rotations to make it balanced
  - insertion and deletion is faster in red black tree thatn the avl tree
  - every leaf which is nil is black and the root is also black
  - if a node is red then its children are black
  - Every avl tree is red black and every red black tree is not avl

- Every perfect binary tree which only contain black node is a red black tree



- **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.
- **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.
- **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 for input array {1, 3, 5, 7, 9, 11}

Traversal in binary tree

- Depth-First Search (DFS) algorithms
  - **Preorder Traversal** (current-left-right): Visits the node first, then left subtree, then right subtree.
  - **Inorder Traversal** (left-current-right): Visits left subtree, then the node, then the right subtree.
  - **Postorder Traversal** (left-right-current): Visits left subtree, then right subtree, then the node
- Breadth-First Search (BFS) algorithms: BFS explores all nodes at the present depth before moving on to nodes at the next depth level. It is typically implemented using a queue. BFS in a binary tree is commonly referred to as Level Order Traversal.

Advantages of Binary Tree

- Efficient Search: Binary Search Trees (a variation of Binary Tree) are efficient when searching for a specific element, as each node has at most two child nodes when compared to linked list and arrays
- Memory Efficient: Binary trees require lesser memory as compared to other tree data structures, therefore memory-efficient.
- Binary trees are relatively easy to implement and understand as each node has at most two children, left child and right child.

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 like arrays and linked list. This is because each node requires two child references or 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 or skewed, meaning that each node has only one child. In this case, search operations in Binary Search Tree (a variation of Binary Tree) can degrade to O(n) time complexity, where n is the number of nodes in the tree.

Applications of Binary Tree

- Binary Tree can be used to represent hierarchical data.
- 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.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.

Binary tree implementation using array

- First method- adding element at each level to the array(make sure the tree is complete)
  - left child of ith element = $(2 \times i) + 1$ th position
  - right child of ith element = $(2 \times i) + 2$ th position
  - Parent is at $\left[\frac{i-1}{2}\right]$ th position

Binary Search Tree

A Binary Search Tree (or BST) is a data structure used in computer science for organizing and storing data in a sorted manner. Each node in a Binary Search Tree has at most two children, a left child and a right child, with the left child containing values less than the parent node and the right child containing values greater than the parent node.

Binary search tree is a binary tree in which for every node the left subtee contains the values less than the node and the right subtree contains the values greater than the node

- number of arrangements of n numbers in the bst - $\frac{(2n)!}{n!(n+1)!}$

Advantages of Binary Search Tree (BST):

- Efficient searching: O(log n) time complexity for searching with a self balancing BST
- Ordered structure: Elements are stored in sorted order, making it easy to find the next or previous element
- Dynamic insertion and deletion: Elements can be added or removed efficiently
- Balanced structure: Balanced BSTs maintain a logarithmic height, ensuring efficient operations
- Doubly Ended Priority Queue: In BSTs, we can maintain both maximum and minimum efficiently

Disadvantages of Binary Search Tree (BST):

- Not self-balancing: Unbalanced BSTs can lead to poor performance
- Worst-case time complexity: In the worst case, BSTs can have a linear time complexity for searching and insertion
- Memory overhead: BSTs require additional memory to store pointers to child nodes
- Not suitable for large datasets: BSTs can become inefficient for very large datasets
- Limited functionality: BSTs only support searching, insertion, and deletion operations

Applications of Binary Search Tree

- A Self-Balancing Binary Search Tree is used to maintain sorted stream of data. For example, suppose we are getting online orders placed and we want to maintain the live data (in RAM) in sorted order of prices. For example, we wish to know number of items purchased at cost below a given cost at any moment. Or we wish to know number of items purchased at higher cost than given cost.
- A Self-Balancing Binary Search Tree is used to implement doubly ended priority queue. With a Binary Heap, we can either implement a priority queue with support of extractMin() or with extractMax(). If we wish to support both the operations, we use a Self-Balancing Binary Search Tree to do both in O(Log n)
- There are many more algorithm problems where a Self-Balancing BST is the best suited data structure, like count smaller elements on right, Smallest Greater Element on Right Side, etc.
- A BST can be used to sort a large dataset. By inserting the elements of the dataset into a BST and then performing an in-order traversal, the elements will be returned in sorted order. When compared to normal sorting algorithms, the advantage here is, we can later insert / delete items in O(Log n) time.
- Variations of BST like B Tree and B+ Tree are used in Database indexing.
- TreeMap and TreeSet in Java, and set and map in C++ are internally implemented using self-balancing BSTs, more formally a Red-Black Tree.

Avl Tree

AVL tree is a binary search tree with an added feature — it automatically ensures that it remains balanced during insertions and deletions.

To Check

- complexity of BST search
- tree vs BST

Practicals

- [x] Insert node in a BST
    - Best case -O(log n) - when the tree is balanced

    - worst case - O(n) - when the tree is skewed
- [x] search node in BST
    - Best case -O(log n) - when the tree is balanced
    - worst case - O(n) - when the tree is skewed
- [x] DFS
    - O(n)T
- [x] BFS
    - O(n)
- [x] count the number of nodes
- [x] find min and max, secondmax , secondmin , thirdmax thirdmin in BST
    - worst case - h- height of the tree - when the tree is skewed
- [x] height of the tree and nodes , height of left subtree and right subtree
    - O(n)T
- [x] find the depth of the binary tree
    - average - O(log n)
    - words n (skewed)
- [x] check bst or not - leetcode
- [x] Check two trees are identical - leetcode
- [x] check tree is a subtree of another
- [x] Find the closest value in BST
- [x] Range sum of BST
- [x] delete node from BST
- [ ] Distance between two nodes in BST

//lt

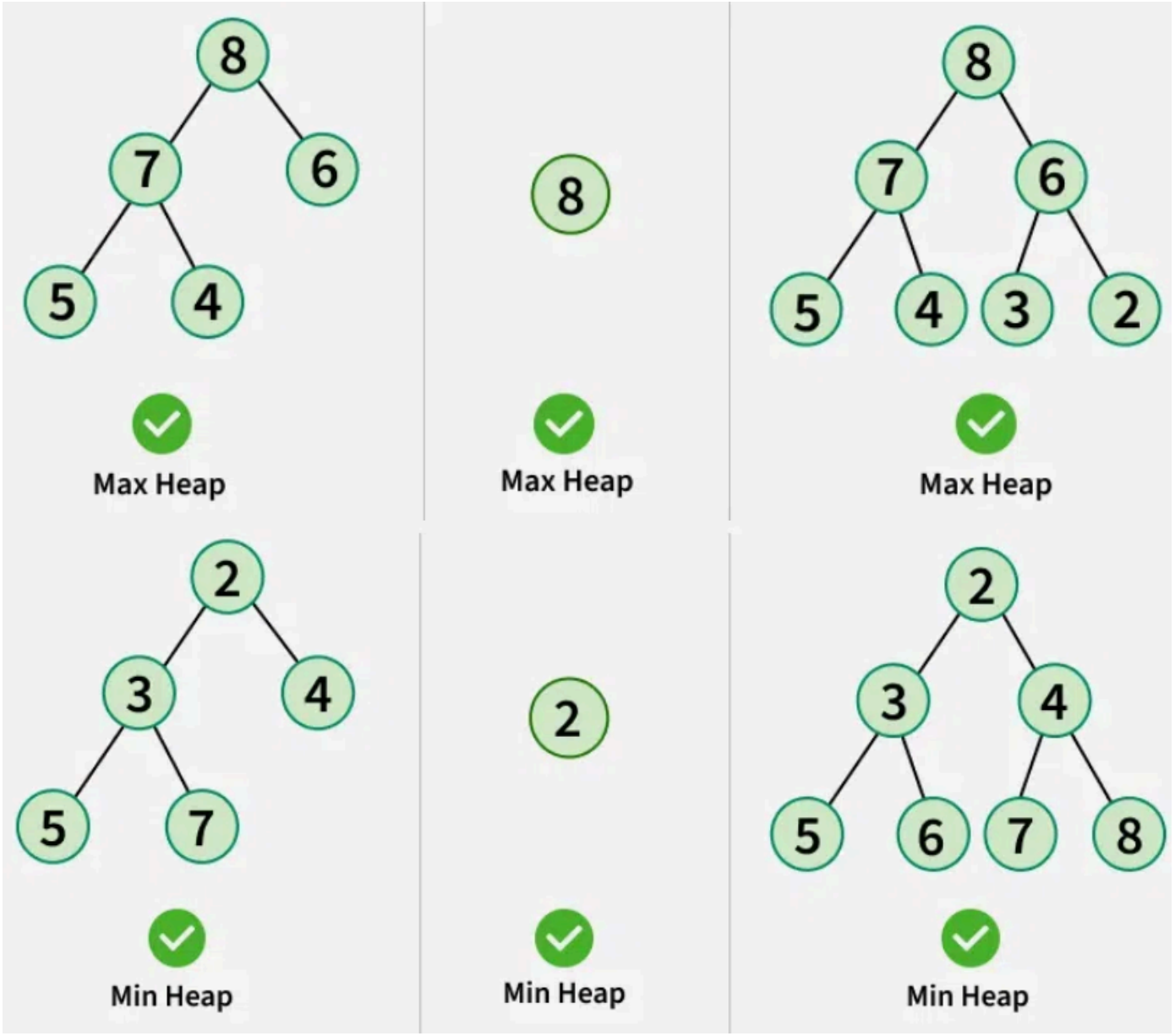
- [ ] check if bst is balanced
- [ ] leetcode question=kth smallest in bst
- [ ] construct binary tree from preOrder postOrder and inorder Traversals

Heap

Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property**.

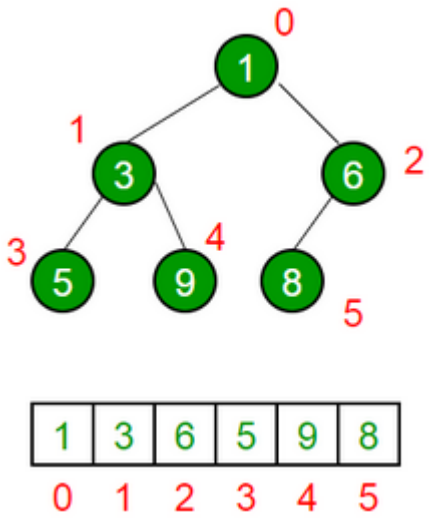
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property**.



A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

- The root element will be at arr[0].
- The below table shows indices of other nodes for the ith node, i.e., arr[i]:

Expression	Description
`arr[(i-1)/2]`	Returns the parent node
`arr[(2*i)+1]`	Returns the left child node
`arr[(2*i)+2]`	Returns the right child node



Operations on Heap:

Below are some standard operations on min heap:

- **getMin()**: It returns the root element of Min Heap. The time Complexity of this operation is O(1). In case of a maxheap it would be getMax().
- **extractMin()**: Removes the minimum element from MinHeap. The time Complexity of this Operation is O(log N) as this operation needs to maintain the heap property (by calling heapify()) after removing the root.
- **decreaseKey()**: Decreases the value of the key. The time complexity of this operation is O(log N). If the decreased key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- **insert()**: Inserting a new key takes O(log N) time. We add a new key at the end of the tree. If the new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- **delete()**: Deleting a key also takes O(log N) time. We replace the key to be deleted with the minimum infinite by calling decreaseKey(). After decreaseKey(), the minus infinite value must reach root, so we call extractMin() to remove the key.
- **heapify()** - Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

Applications of Heaps:

- Heap Sort: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
- Priority Queue: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time. Binomial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
- Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.

- Many problems can be efficiently solved using Heaps. See following for example. a) K’th Largest Element in an array. b) Sort an almost sorted array/ c) Merge K Sorted Arrays.

Advantages of Heap Data Structure

- Time Efficient: Heaps have an average time complexity of O(log n) for inserting and deleting elements, making them efficient for large datasets. We can convert any array to a heap in O(n) time. The most important thing is, we can get the min or max in O(1) time
- Space Efficient : A Heap tree is a complete binary tree, therefore can be stored in an array without wastage of space.
- Dynamic: Heaps can be dynamically resized as elements are inserted or deleted, making them suitable for dynamic applications that require adding or removing elements in real-time.
- Priority-based: Heaps allow elements to be processed based on priority, making them suitable for real-time applications, such as load balancing, medical applications, and stock market analysis.
- In-place: Most of the applications of heap require in-place rearrangements of elements. For example HeapSort.

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.
- 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.

Heap Sort

- Rearrange array elements so that they form a Max Heap.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.
    - Heapify the remaining elements of the heap.
- Heap sort is an in-place algorithm.

- Its typical implementation is not stable but can be made stable
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Complexity Analysis of Heap Sort

- Time Complexity: O(n log n)
- Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

Advantages of Heap Sort

- Efficient Time Complexity: Heap Sort has a time complexity of O(n log n) in all cases. This makes it efficient for sorting large datasets. The log n factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.
- Memory Usage: Memory usage can be minimal (by writing an iterative heapify() instead of a recursive one). So apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- Simplicity: It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

Disadvantages of Heap Sort

- Costly: Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is O(n Log n) for both.
- Unstable: Heap sort is unstable. It might rearrange the relative order.
- Inefficient: Heap Sort is not very efficient because of the high constants in the time complexity.

To check

- Advantage of priority queue using heap
- hepify complexity

Priority queue

It is an abastract data type that operates similar to qqueue except that each element has a certain priority. The pritority of the elements i the priority queue determines the order in which elements are removed from the PQ

Applications

- used in certain implementaion of dijkstra's shortest path algorithms
- used in huffman codign - a lossless dta compression algorithm
- Best first search algorithms
- used in minimum spanning tree

Practicals

- [x] implement heap data structure
- [x] decrease and increase key in heap
- [x] heapify an array
- [x] Heap sort
- [x] priority queues in heap
- [x] Top K Frequent Elements(use Heap)
    - create a frequencey map
    - insert that elements and frequency to an maxHeap
    - extract largest k elements from the heap
- [ ] Find the longest non-repeating substring in a string.
- [ ] find unique characters from given string

Trie

The Trie data structure is a tree-like data structure used for storing a dynamic set of strings. It is commonly used for efficient retrieval and storage of keys in a large dataset.

A Trie data structure consists of nodes connected by edges. Each node represents a character or a part of a string. The root node, the starting point of the Trie, represents an empty string. Each edge emanating from a node signifies a specific character. The path from the root to a node represents the prefix of a string stored in the Trie.

- Trie supports operations such as insertion, search, and deletion of keys,
- The root node of a Trie (Prefix Tree) is the starting point of the data structure. It does not store any character itself but acts as an entry point to store words or prefixes.

Prefix Trie

A Prefix Trie (or simply "Trie") is a tree data structure used to store a set of words or prefixes efficiently.

- insertion takes O(n) Time and O(1) Space
- Searching takes – O(n) Time and O(1) Space
- autocompletion takes - O(n) Time

Uses

- Autocompeletion
- spell checking
- dictionary impelementation
- IP Routing

Suffix Trie

A Suffix Trie stores all suffixes of a given string, allowing for fast substring searches.

- insertion takes O(n^2) time
- search takes O(m) time - m is the length of the seaching string

Uses of Suffix Trie:

- Substring Search
- Pattern Matching (DNA sequences, Text Searching, etc.)
- Plagiarism Detection
- Bioinformatics (Genome Analysis)

Advantages of Trie

- Efficient Prefix search
- No hash collission
- Lexicographical sorting
- Useful for DNA & Text Search

Comparison Table: Trie vs Other Data Structures

Feature	Trie (Prefix Tree)	Hash Table	BST
Search Time	O(n)	O(1) (avg), O(n) (worst)	O(log n)
Prefix Search	✅ Fast	❌ Not Supported	❌ Not Efficient
Space Efficiency	✅ Shared Prefixes	❌ High (Collision Handling)	✅ Efficient
Lexicographical Order	✅ Yes	❌ No	✅ Yes
Collision Handling	❌ Not Needed	❌ Needed	❌ Not Needed

compressed Tree

A Compressed Trie is an optimized version of a regular Trie that reduces the number of nodes by compressing A Compressed Trie is an optimized version of a standard Trie where long chains of single-child nodes are merged into a single node. This reduces memory usage and improves search efficiency.

Time complexiteis

Operation	Average Case	Worst Case
Insertion	(O(N))	(O(N))

Operation	Average Case	Worst Case
Search	(O(N))	(O(N))
Starts With	(O(N))	(O(N))
Deletion	(O(N))	(O(N))
Auto-suggest	(O(N + M))	(O(N + M))
Longest Prefix	(O(N))	(O(N))

Where:

- (N) is the length of the word being inserted, searched, or deleted.
- (M) is the number of words that share a common prefix (for auto-suggest).

Breakdown of Each Operation

1. Insertion: (O(N))

- We traverse the word character by character, inserting nodes as needed.
- In the worst case, we insert all (N) characters (if the word is new).

2. Search: (O(N))

- We traverse the word character by character.
- If all characters exist, the word is found in (O(N)).
- If a character is missing, we return `false` early.

3. Starts With: (O(N))

- Similar to search, but we only check if a prefix exists, not the end of the word.

4. Deletion: (O(N))

- We traverse the word to find the last character.
- If it's the only child, we remove nodes while backtracking.
- Worst case: the word is fully removed, taking (O(N)).

5. Auto-Suggest: (O(N + M))

- First, we traverse the prefix ((O(N))).
- Then, we perform a BFS/DFS to gather all words starting with the prefix ((O(M))).
- (M) is the number of words sharing the prefix.

6. Longest Common Prefix: (O(N))

- We traverse down while there is only one child.
- The longest possible prefix length is (O(N)), where (N) is the shortest word.

Trie vs. Other Data Structures

Operation	Trie ((O(N)))	HashMap ((O(1)))	Binary Search Tree (BST) ((O(\log M)))
Insert	(O(N))	(O(1)) (Hashing)	(O(\log M)) (Balanced BST)
Search	(O(N))	(O(1)) (Hashing)	(O(\log M)) (Balanced BST)
Prefix	(O(N))	(O(N)) (Scanning Keys)	(O(\log M))
Suggest	(O(N + M))	(O(M)) (Iterate All Keys)	(O(M \log M)) (Sorting)

- **Trie is best for prefix-based operations** like auto-suggestions and longest common prefix.
- **HashMaps are best for exact word lookups** due to (O(1)) hashing.
- **BSTs are good for sorted order operations**, but prefix searches are inefficient.

Practicals

- [x] Insert Word in Trie
- [x] Search Word in Trie
- [x] trie prefix search implementation
- [x] Delete a Word using Trie
- [x] Replace a word by Another Word using Trie
- [x] Count number of words using Trie
- [x] autosuggestion using trie
- [ ] Print all words
- [x] find the lognest prefix
- [x] find unique characters from given string

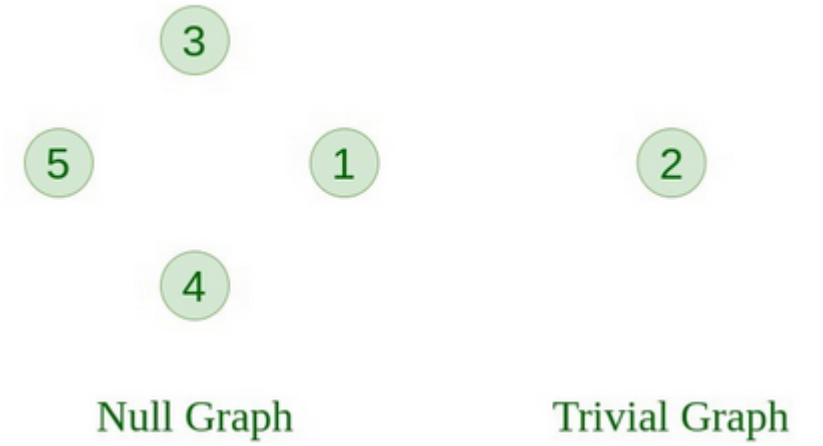
Graph

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(V, E).

- **Vertices**: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- **Edges**: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

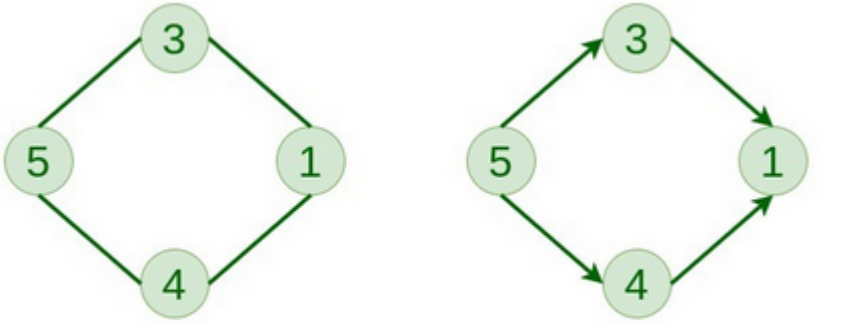
Types of Graphs

- **Null Graph**- A graph is known as a null graph if there are no edges in the graph.
- **Trivial Graph**- Graph having only a single vertex, it is also the smallest graph possible.



Null Graph Trivial Graph

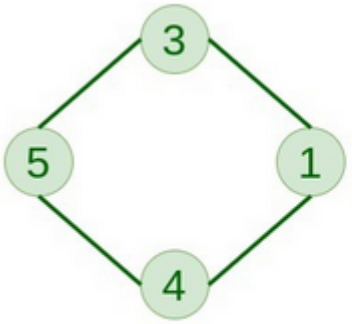
- **Undirected Graph** - A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.
- **Directed Graph** - A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



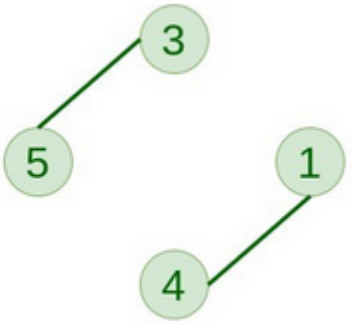
Undirected Graph Directed Graph

- **Connected Graph**- The graph in which from one node we can visit any other node in the graph is known as a connected graph
- **Disconnected Graph**- The graph in which at least one node is not reachable from a node is known as a disconnected graph.
- **Strongly connected**: In directed graphs, if there is a directed path between every pair of vertices, the graph is strongly connected.

- **Weakly connected**: A directed graph is weakly connected if its undirected version is connected.

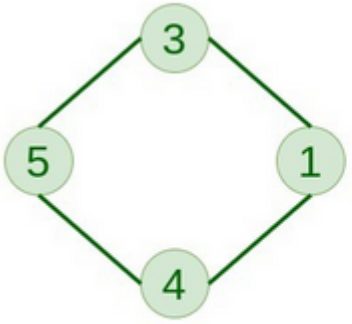


Connected Graph

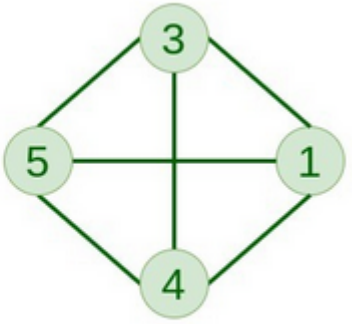


Disconnected Graph

- **Regular Graph**- The graph in which the degree of every vertex is equal to K is called K regular graph.
- **Complete Graph**- The graph in which from each node there is an edge to each other node.

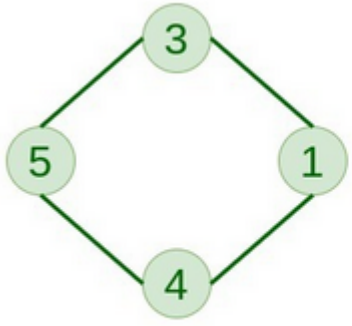


2-Regular

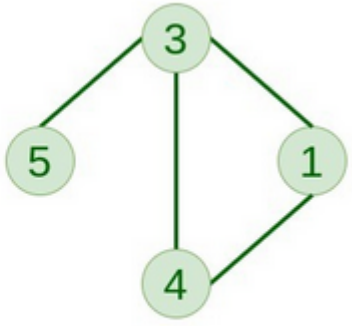


Complete Graph

- **Cycle Graph**- The graph in which the graph is a cycle in itself, the minimum value of degree of each vertex is 2.
- **Cyclic Graph**- A graph containing at least one cycle is known as a Cyclic graph

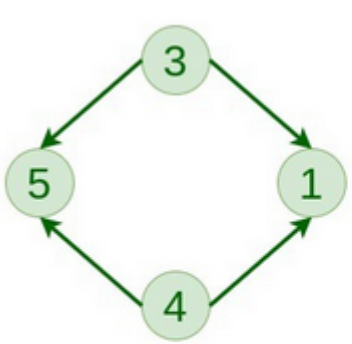


Cycle Graph

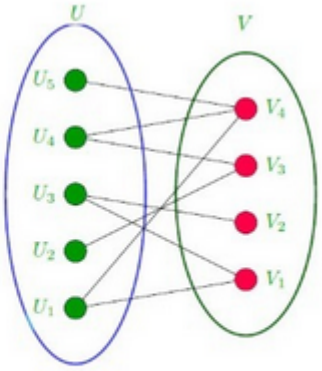


Cyclic Graph

- **Directed Acyclic Graph**- A Directed Graph that does not contain any cycle.
- **Bi Partite Graph**- A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



Directed Acyclic Graph



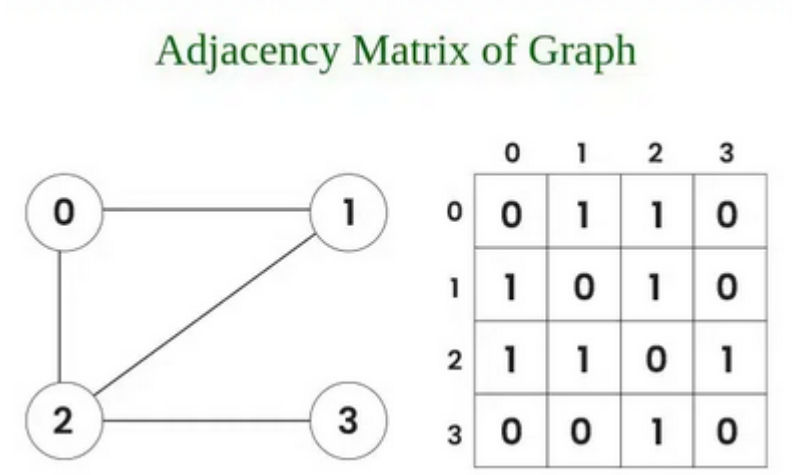
Bipartite Graph

- **Weighted Graph** - A graph in which the edges are already specified with suitable weight is known as a weighted graph.
  Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.
- **Simple graph**: A graph without loops or multiple edges.
- **Multigraph**: A graph that may have multiple edges between the same pair of vertices.
- **isolated vertex** - vertex with degree zero (no edges)
- **Path**: A sequence of vertices such that each vertex is connected to the next by an edge.
- **Cycle**: A path that starts and ends at the same vertex, and it does not repeat any edge or vertex except for the starting/ending vertex.
- **Eulerian path** - An Eulerian path (or Eulerian trail) in a graph is a path that visits every edge of the graph exactly once. The path may pass through vertices multiple times, but each edge is traversed only once.
- **Hamiltonian Path/Circuit**: A path or circuit that visits each vertex exactly once.
- **Graph indexing** is a technique used to speed up the process of querying and accessing specific parts of a graph. It involves creating data structures that store information about the graph in such a way that specific queries (like finding neighbors, paths, or subgraphs) can be answered more efficiently than by directly traversing the graph each time
- The **degree of a vertex** is the number of edges connected to it.
- **In-degree**: The number of edges directed towards a vertex (for directed graphs).
- **Out-degree**: The number of edges directed away from a vertex (for directed graphs).

Representation of graph data structure

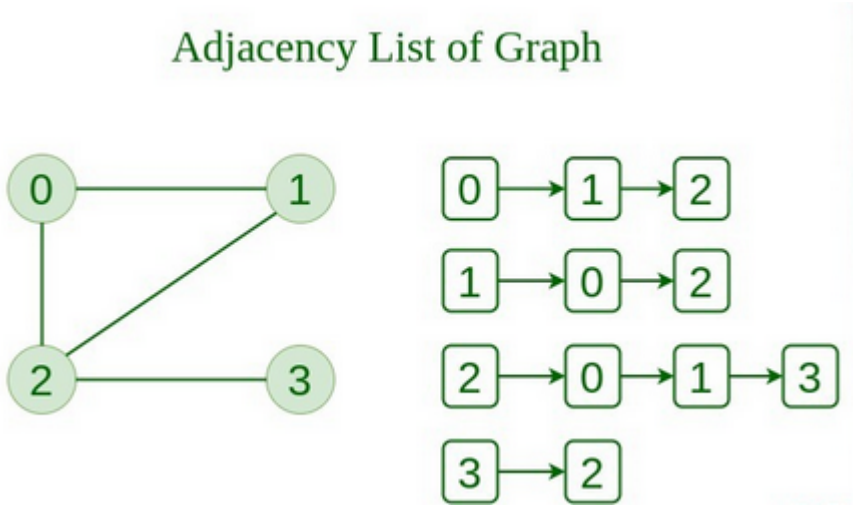
- **Adjacency Matrix Representation of Graph Data Structure**:- In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices

$.O(n^2)S$,



- good when the graph is dense

- **Adjacency List Representation of Graph**:- This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex. $O(n^2 + 2e)S$



- good when the graph is sparse

  When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as Prim's and Dijkstra adjacency matrix is used to have less complexity.

Types of edges after DFS in directed graph

- **tree edges** - the member of dfs traversal
- forware edge - $e(x, y)$ in which y a appears after x and there is a path from x to y
- back edge - $e(x, y)$ in which x appers before y and there is a pathfrom y to x
- cross edge - $e(x, y)$ in which there is no path from y to x

Real-Life Applications of Graph Data Structure

- Used heavily in social networks. Everyone on the network is a vertex (or node) of the graph and if connected, then there is an edge. Now imagine all the features that you see, mutual friends, people that follow you, etc can seen as graph problems.

- Used to represent the topology of computer networks, such as the connections between routers and switches.
- Used to represent the connections between different places in a transportation network, such as roads and airports.
- Neural Networks: Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
- Compilers: Graph Data Structure is used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
- Robot planning: Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.
- Dependencies in a software project (or any other type of project) can be seen as graph and generating a sequence to solve all tasks before dependents is a standard graph topological sorting algorithm.
- For optimizing the cost of connecting all locations of a network. For example, minimizing wire length in a wired network to make sure all devices are connected is a standard Graph problem called Minimum Spanning Tree.

Advantages of Graph Data Structure:

- Graph Data Structure used to represent a wide range of relationships as we do not have any restrictions like previous data structures (Tree cannot have loops and have to be hierarchical. Arrays, Linked List, etc are linear)
- They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
- Any real world problem where we certain set of items and relations between them can be easily modeled as a graph and a lot of standard graph algorithms like BFS, DFS, Spanning Tree, Shortest Path, Topological Sorting and Strongly Connected
- Graph Data Structure can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

Disadvantages of Graph Data Structure:

- Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.
- Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.
- Graph Data Structure can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

Application of weighted graphs

- shortest path problems
- minimum spanning treee
- Weighted graphs are used in optimization problems where resources need to be allocated to tasks or schedules based on certain constraints.
- In social networks, the weight of an edge might represent the strength or frequency of interaction between two individuals or entities.
- In telecommunications, a network can be represented as a weighted graph, where each edge has a weight that represents the cost, bandwidth, or latency associated with communication between two nodes.
- Graphs are widely used in biology and genetics to model relationships and networks, such as metabolic pathways, protein interactions, or gene regulation networks.
- In electrical engineering, weighted graphs are used to model networks of resistors, capacitors, or transistors, where the weights represent electrical characteristics such as resistance or capacitance.

Spanning Tree

A Spanning Tree of a graph is a subset of the graph that:

- Includes all vertices of the original graph.
- Is a tree (i.e., it has no cycles).
- Has exactly (V - 1) edges, where V is the number of vertices in the graph.

Properties

- a graph can have more than one spanning tree
- spanning tree should not contain any cycles
- spanning tree should be connnected
- removing one edge from the spanning tree make it disconnected and adding one edge will create a cycle
- a complete undirected graph can have n^{n-2} spanning treees
- every connected and undirected graph must have atleast one spanning tree
- from completed graph removing maximum of $(e - n + 1)$ edges , we can construct a spannign tree

Minimum Spanning Tree

spanning tree with minimum edge weight sum

Kruskal's algorithm for minimum spanning tree

- remove all cycles
- remove parallel edges with higher weight from the graph

- connect the graph from minimum weight to the higher weights without forming any cycles

Prim's Algorightm for minimum spanning tree

- remove all cycles
- remove all paraller edges with higher weight from the graph
- choose any vertex as the root node
- choose the edge with minium weight of the node
- from the two vertex choose the one edge which has the lower weight and add it
- continue with the two edge vertexes

Cycle detection using DFS

depth first search starting from a node and for each traversal note down the parent node and also track the visited nodes if there is a stage occures in which a node has a visited neighbour which is not the parent then there is a cycle detected

To Check

- tarjan's algorithm
- djkistras algorithm
- prims algorithm, Kruskal algorithm.

Practicals

- [x] implement using adjacency list
- [x] implementation of DFS
- [x] implementation of BFS
- [x] check two nodes have a path
- [ ] implement using adjacency matrix
- [ ] Find the number of cycle in a graph
- [ ] Delete a cycle in a directed graph
- [ ] Solve problems to find the shortest path in a graph.
- [x] clone graph - leetcode
- [ ] number islands- graph - leetcode
  https://www.geeksforgeeks.org/explore/?category[]=

Time complexities

1. Tree (General)

Operation	Time Complexity
Insertion	O(1) (if known location), O(n) (if searching)
Deletion	O(1) (if known location), O(n) (if searching)
Traversal (DFS/BFS)	O(n)
Search	O(n) (worst case)

> **Note:** General trees don’t have a fixed structure like BST or Trie.

---

2. Binary Search Tree (BST)

Operation	Average Case	Worst Case (Skewed Tree)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Search	O(log n)	O(n)
Traversal (Inorder, Preorder, Postorder)	O(n)	O(n)

> **Balanced BSTs (e.g., AVL, Red-Black Trees)** always maintain **O(log n)** operations.

---

3. Heap (Min/Max Heap)

Operation	Time Complexity
Insertion	O(log n)
Deletion (Extract Min/Max)	O(log n)
Search	O(n) (no direct search)
Get Min/Max	O(1)
Heapify	O(n) (building a heap)

> **Binary Heaps are used in Priority Queues, Dijkstra’s Algorithm, etc.**

---

4. Trie (Prefix Tree)

Operation	Time Complexity
Insertion	O(L) (L = length of word)

Operation	Time Complexity
Deletion	O(L)
Search	O(L)
Prefix Search (Autocomplete)	O(L)

> **Trie is efficient for dictionary-based problems like autocomplete.**

---

5. Graph (Adjacency List vs. Adjacency Matrix)

Operation	Adjacency List	Adjacency Matrix
Add Vertex	O(1)	$O(V^2)$
Add Edge	O(1)	O(1)
Remove Edge	O(V)	O(1)
Remove Vertex	O(V + E)	$O(V^2)$
Search (BFS/DFS)	O(V + E)	$O(V^2)$

> **Graph Traversal (BFS/DFS)** complexity depends on the number of **vertices (V)** and **edges (E).**
> **Dijkstra's Algorithm (using Heap/Priority Queue): O((V + E) log V)**

Applications

Application of tree -filesystems , Dom ,databases , network routing
Applications of bst - databases , implementing dictionaries , garbage colection algorithms
Applications of heap - priority queue implementation , graph shortest path algorithms , heap sort , sheduling ,memory management
Applications of trie - autocomplete suggestions , spell checking , ip routing , genome sequencing
Graph - maps and navigation , social networks , neural networks, web crawlera