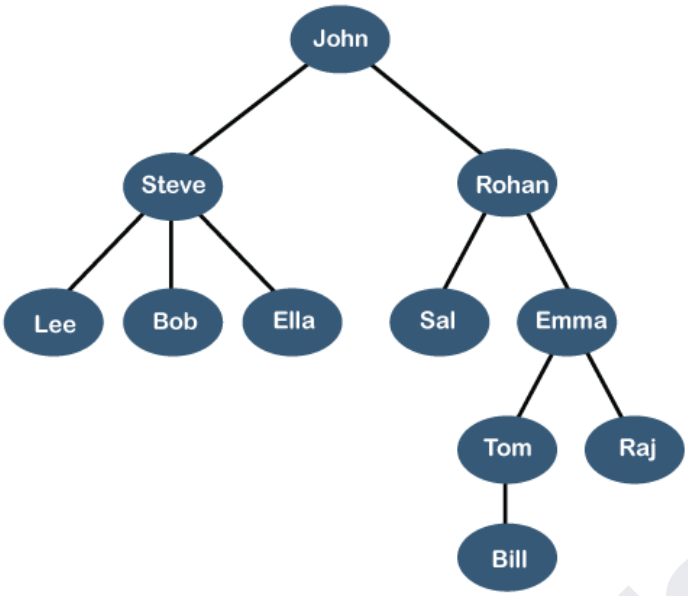
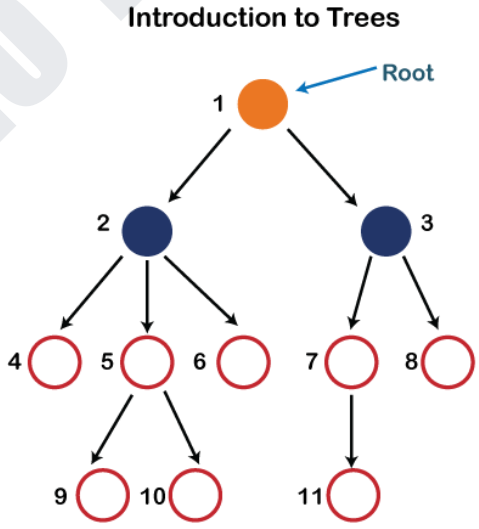
Tree

A tree is also one of the data structures that represent hierarchical data.



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



- **Node:** A node is a fundamental element of a tree. It represents a single element or entity in the tree. Each node can have zero or more child nodes.

- **Root**: The root node is the topmost node in the tree hierarchy. In other words the root node is the one that doesn't have any parent. In the above structure node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a **parent-child relationship**.

- **Siblings**: Nodes that have the same parent are called siblings. They share the same immediate parent node.

- **Leaf Node**: A leaf node, also known as a terminal node, is a node that does not have any child nodes. In other words it is at the bottom of the tree and does not have any descendants.

- **Internal Node**: An internal node is a node that has one or more child nodes. It is not a leaf node and is located between the root and the leaf nodes.

- **Path**: A path in a tree is a sequence of nodes that are connected through parent-child relationships. It represents the route from one node to another.

- **Depth and Height**: The depth of a node in a tree is the number of edges from the root node to that node. The height of a tree is the maximum depth among all its nodes.

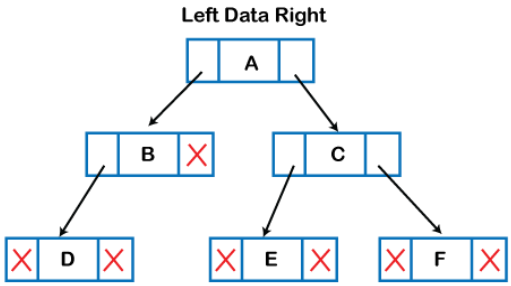
- **Number of edges**: If there are n nodes, then there would be n-1 edges. Each arrow in the structure represents the link known as an edge. There would be one link for the parent - child relationship.

- **Depth of node x**: The depth of node x can be defined as the length of the path from the root to the node x. The root node has 0 depth.

- **Height of node x**: The height of node x can be defined as the longest path from the node x to the leaf node.

- **Degree**: The number of child nodes.

Implementation of tree



The tree data structure can be created by creating the help of the pointers. The tree in the memory can be represented as shown in the above figure. The above figure shows the representation of the node containing three fields. The second field stores the data, the first field stores the address of the left child, and the third field stores the address of the right child.

```
Class Node{
        late int data;
        Node? left, right;
        Node(this.data);
}
```

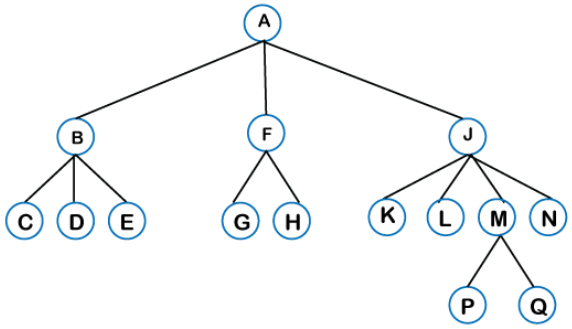
The above structure can only be defined for the **binary trees** because the binary tree can have utmost two children, and **generic trees** can have more than two children. The structure of the node for generic trees would be different as compared to the binary tree.

Applications

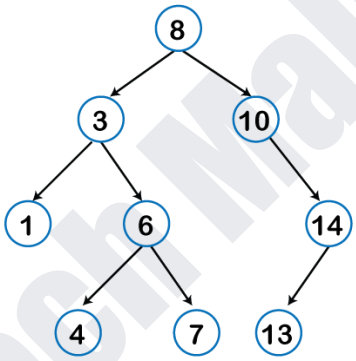
- ☐ File system: Tree data structures are widely used to represent file systems. Each directory is represented as a node, and the hierarchical organization of directories and files is reflected in the tree structures.
- ☐ Spell Checkers: Tree data structures, such as trie (prefix tree), are utilized in spell checkers and autocomplete systems. Tries allow for efficient searching and suggestions by storing and organizing words based on their prefixes.
- ☐ Database System: Tree data structures, such as B-tree and B+ trees, are used extensively in database systems to index and organize data. These trees provide efficient search, insertion and deletion operations, making them suitable for large-scale data storage and retrieval.
- ☐ Compiler Design: In compiler design, abstract syntax trees(ASTs) are used to represent the structure of programming language statements and expressions. ASTs are generated during the parsing phase and serve as a basis for subsequent compilation steps such as optimization and code generation.

Types of Tree Data structure

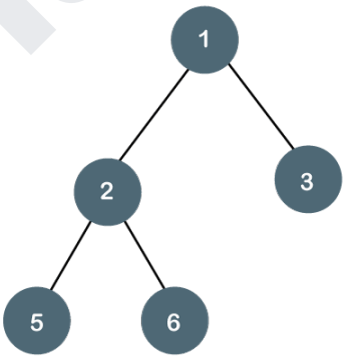
General Tree: The general tree is one of the types of tree data structure. In the general tree, a node can have either 0 or maximum n number of nodes.



Binary tree: A binary tree is a tree in which each node can have at most two child nodes, referred to as the left child and the right child. Binary trees are commonly used in many algorithms and data structures. Here at most means whether the node has 0 and 1 or 2 nodes.



The binary tree means that the node can have a maximum of two children.

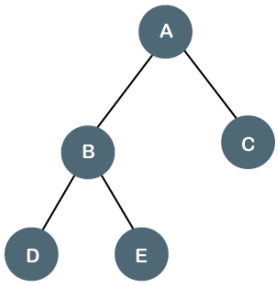


Height of binary tree = node - 1/ h=n-1;

Types of Binary Tree

1. **Full/ proper/ strict Binary Tree (0 or 2 Children)**

   The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree in which each node must contain 2 children except the leaf nodes.

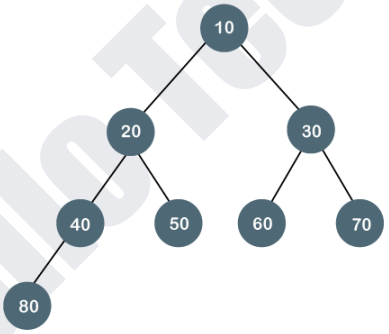


   The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.

   The minimum number of nodes in the full binary tree is 2*h-1.
   The minimum height of the full binary tree is $\log_2(n+1) - 1$.

2. **Complete Binary Tree (Left to right filled)**

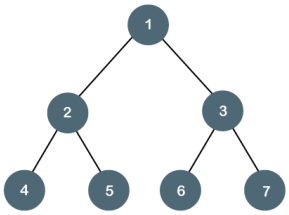
   All nodes are completely filled except the last level, all the nodes must be as left as possible.

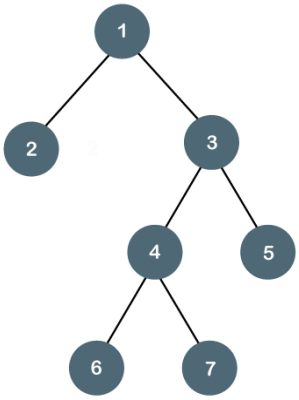


   The above tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

3. **Perfect Binary Tree (All levels have 2 child)**

   All the nodes have 2 children, and all the leaf nodes are at the same level.

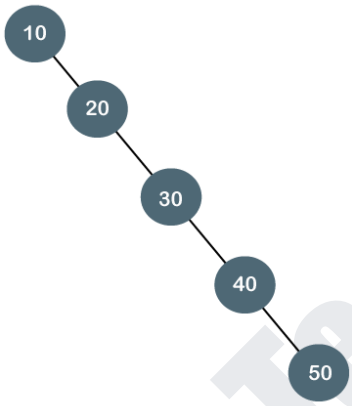




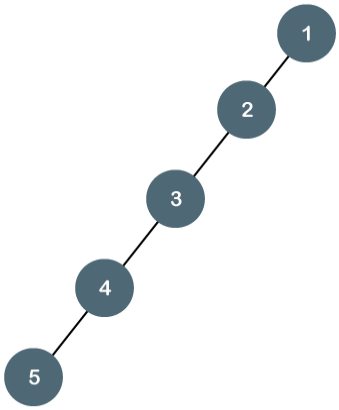
Not a perfect Tree.

4. **Degenerate Binary Tree**

The degenerate binary tree is a tree in which all the internal nodes have only one child.



A right-skewed tree as all the nodes have a right child only.

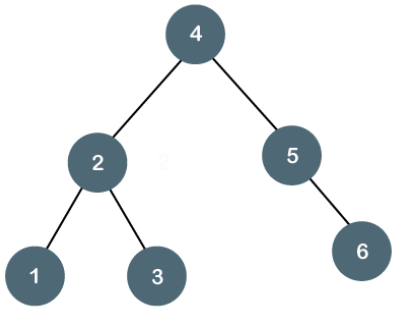


A left-skewed tree as all the nodes have a left child only.

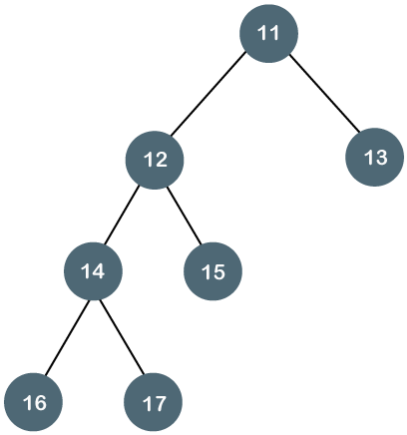
5. **Balanced Binary Tree ()**

   The balanced binary tree is a tree in which both left and right trees differ by utmost 1.

   The tree is a balanced binary tree because the difference b/w the left subtree and right subtree is zero.



   Balanced tree ^



   Unbalanced tree ^

   The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

Binary Search Tree: A BST is a binary tree in which the left child of a node contains value less than or equal to the node's value, and the right child contains a value greater than the node's value. BSTs are often used to efficiently store and retrieve data in a sorted manner.

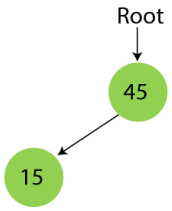
- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.

- As compared to array and linked lists, insertion and deletion operations are faster in BST

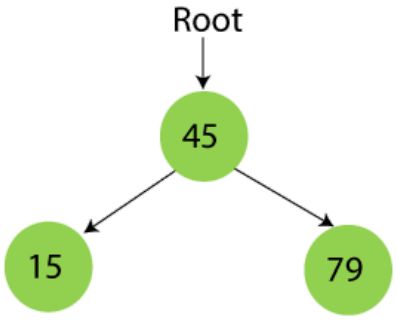
Elements - **45, 15, 79, 90, 10, 55, 12, 20, 50**

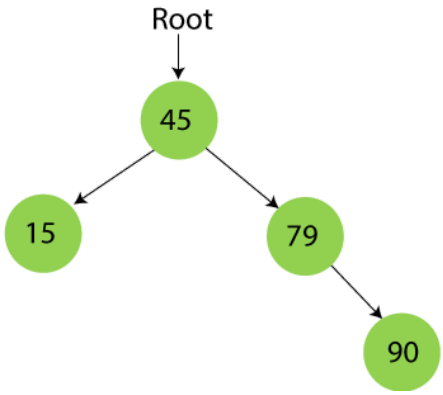
- First we have to insert 45 into the tree as the root of the tree.
- Then read the next element, if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

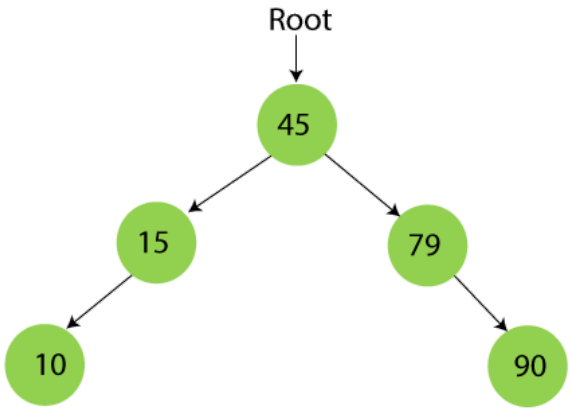
Insert 45

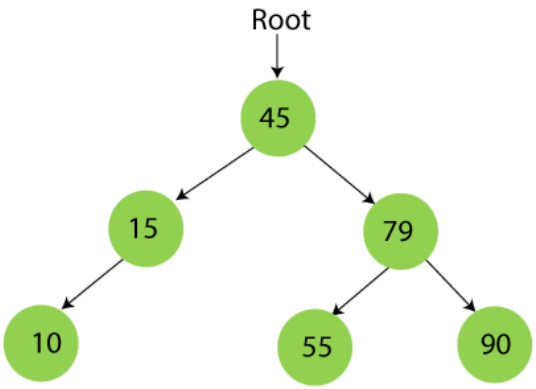


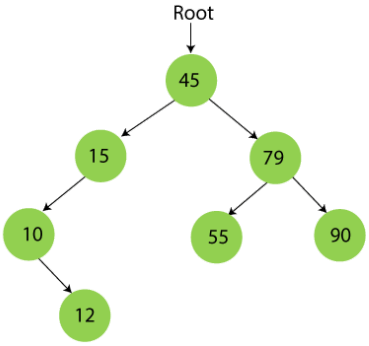
As 15 is smaller than 45, so insert it as the root node of the left subtree.

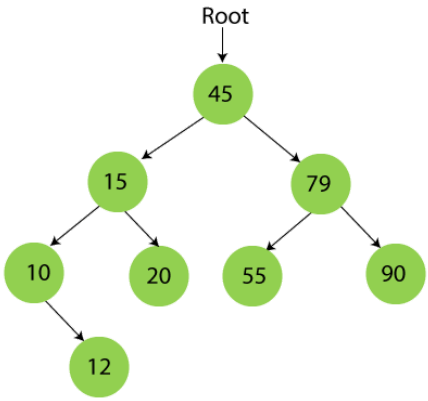


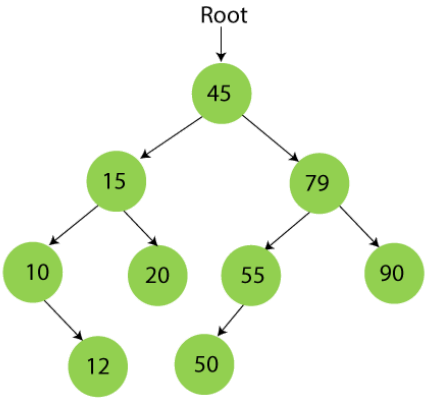








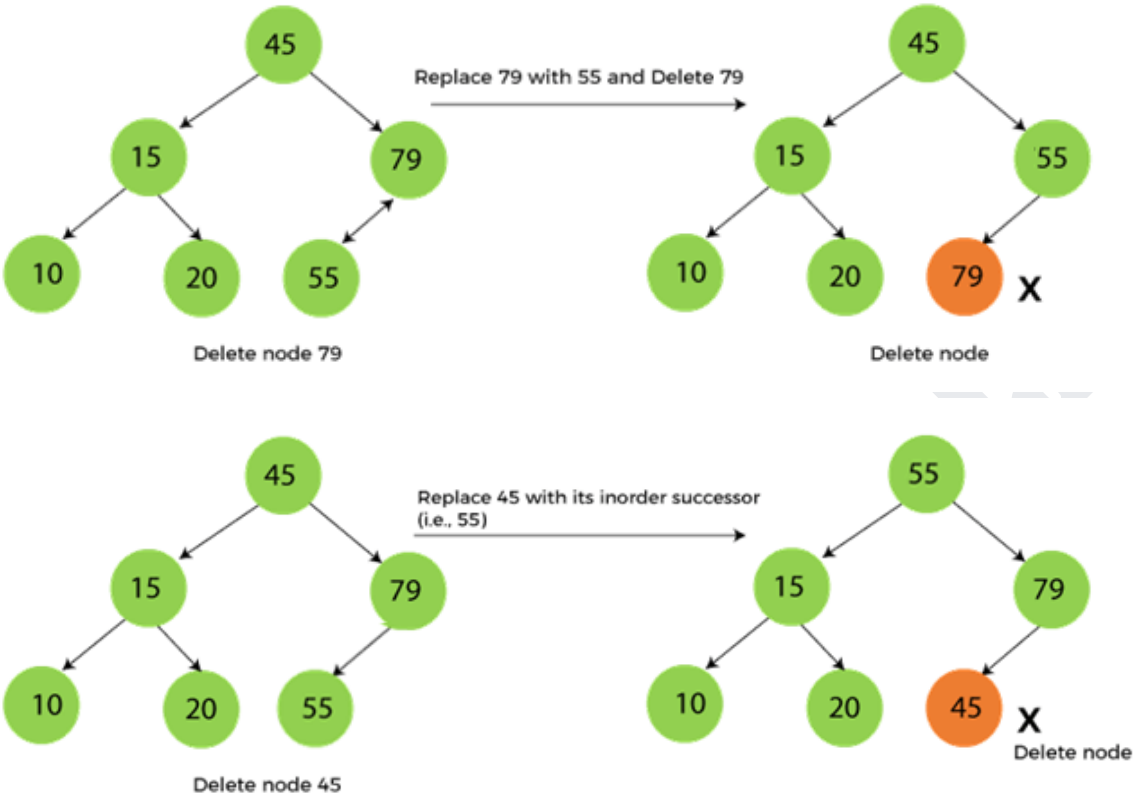




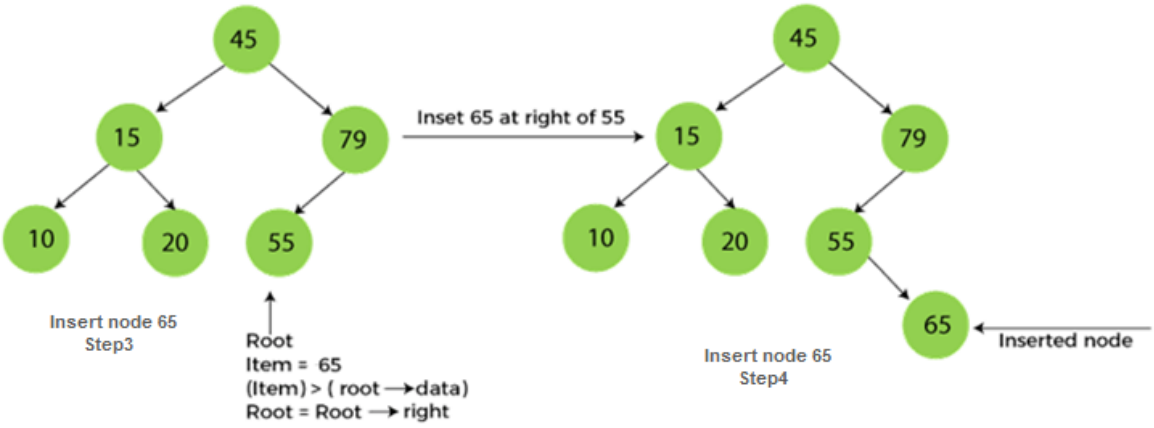
Now the creation of a binary search tree is completed. After that lets move towards the operations that can be performed on BST.

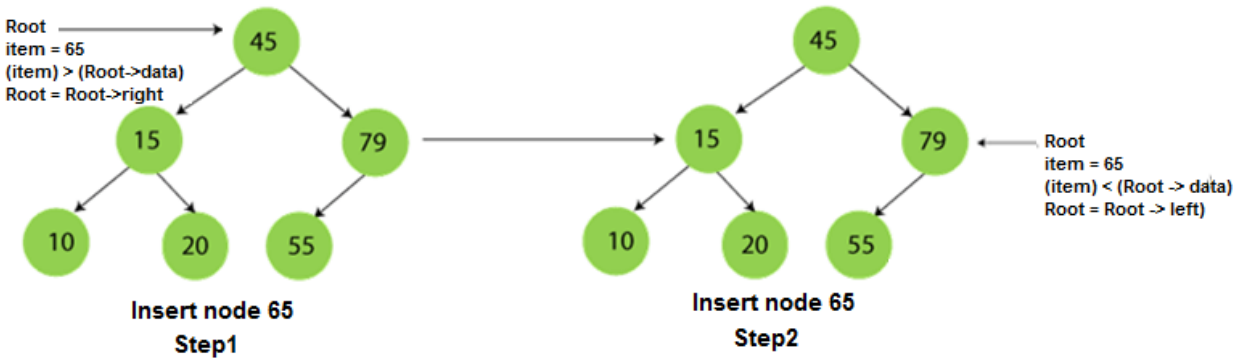
Deletion : In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So we simply have to replace the child

node with NULL and free up the allocated space.



Insertion





1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)

```
import 'dart:io';

class Node {
  late int data;
  Node? left, right;
  Node(this.data);
}

class BinarySearchTree {
  Node? root;
  void insert(int data) {
    Node? newNode = Node(data);
    if (root == null) {
      root = newNode;
```

```
    return;
  }
  Node? currentNode = root;
  while (true) {
    if (data < currentNode!.data) {
      if (currentNode.left == null) {
        currentNode.left = newNode;
        break;
      } else {
        currentNode = currentNode.left;
      }
    } else {
      if (currentNode.right == null) {
        currentNode.right = newNode;
        break;
      } else {
        currentNode = currentNode.right;
      }
    }
  }
}

bool contains(int data) {
  if (root != null) {
    Node? currentNode = root;
    if (currentNode!.data == data) {
      return true;
    }
    while (currentNode != null) {
      if (data < currentNode.data) {
        currentNode = currentNode.left;
      } else if (data > currentNode.data) {
        currentNode = currentNode.right;
      } else {
        return true;
      }
    }
  }
  return false;
}

void remove(int data) {
  _removeHelper(data, null, root);
}
```

```
void _removeHelper(int target, Node? parentNode, Node? currentNode) {
  while (currentNode != null) {
    if (target < currentNode.data) {
      parentNode = currentNode;
      currentNode = currentNode.left;
    } else if (target > currentNode.data) {
      parentNode = currentNode;
      currentNode = currentNode.right;
    } else {
      if (currentNode.left == null && currentNode.right == null) {
        if (parentNode == null) {
          root = null;
          return;
        } else if (parentNode.left == currentNode) {
          parentNode.left = null;
          return;
        } else if (parentNode.right == currentNode) {
          parentNode.right = null;
          return;
        }
      } else if (currentNode.right == null) {
        currentNode.data = _findLargest(currentNode.left);
        _removeHelper(currentNode.data, currentNode, currentNode.right);
      } else {
        currentNode.data = _findSmallest(currentNode.right);
        _removeHelper(currentNode.data, currentNode, currentNode.left);
      }
    }
  }
}

int _findLargest(Node? currentNode) {
  if (currentNode?.right == null) {
    return currentNode!.data;
  } else {
    return _findLargest(currentNode!.right);
  }
}

int _findSmallest(Node? currentNode) {
  if (currentNode?.left == null) {
    return currentNode!.data;
  } else {
```

```
    return _findSmallest(currentNode!.left);
  }
}

void inOrder() {
  print('Printed the inOrder');
  _inOrderHelper(root);
  print('');
}

void _inOrderHelper(Node? temp) {
  if (temp != null) {
    _inOrderHelper(temp.left);
    stdout.write('${temp.data} --> ');
    _inOrderHelper(temp.right);
  }
}

void preOrder() {
  print('Printed the preOrder');
  _preOrderHelper(root);
  print('');
}

void _preOrderHelper(Node? temp) {
  if (temp != null) {
    stdout.write('${temp.data} --> ');
    _preOrderHelper(temp.left);
    _preOrderHelper(temp.right);
  }
}

void postOrder() {
  print('Printed the postOrder');
  _postOrderHelper(root);
  print('');
}

void _postOrderHelper(Node? temp) {
  if (temp != null) {
    _postOrderHelper(temp.left);
    _postOrderHelper(temp.right);
    stdout.write('${temp.data} --> ');
  }
```

```
}

void closest(int data) {
  if (root == null) {
    print('Tree is empty');
    return;
  }
  int close = root!.data;
  Node? currentNode = root;
  while (currentNode != null) {
    if (data == currentNode.data) {
      print('Closeset value is ${currentNode.data}');
      return;
    } else if (data < currentNode.data) {
      if ((currentNode.data - data).abs() < (close - data).abs()) {
        close = currentNode.data;
      }
      currentNode = currentNode.left;
    } else {
      if ((currentNode.data - data).abs() < (close - data).abs()) {
        close = currentNode.data;
      }
      currentNode = currentNode.right;
    }
  }
  print('The closest value is $close');
  return;
}

void prime(Node? temp) {
  if (temp != null) {
    prime(temp.left);
    if (primeNo(temp.data)) {
      print(temp.data);
    }
    prime(temp.right);
  }
}

bool primeNo(int data) {
  for (int i = 2; i < data; i++) {
    if (data % i == 0) {
      return false;
    }
```

```
    }
    return true;
  }

  void proveDisplay() {
    _proveTheIntNumber(root);
  }

  void _proveTheIntNumber(Node? temp) {
    if (temp != null) {
      _proveTheIntNumber(temp.left);
      print(the_number_is_int(temp.data));
      _proveTheIntNumber(temp.right);
    }
  }

  bool the_number_is_int(var data) {
    if (data is int) {
      return true;
    }
    return false;
  }
}

void main() {
  BinarySearchTree tree = BinarySearchTree();
  tree.insert(10);
  tree.insert(8);
  tree.insert(7);
  tree.insert(11);
  tree.insert(13);
  tree.insert(12);
  print('');
  print(tree.contains(13));

  print('');
  tree.inOrder();
  tree.preOrder();
  tree.postOrder();

  print('');
  tree.closest(15);
  print('');
  tree.proveDisplay();
```

}

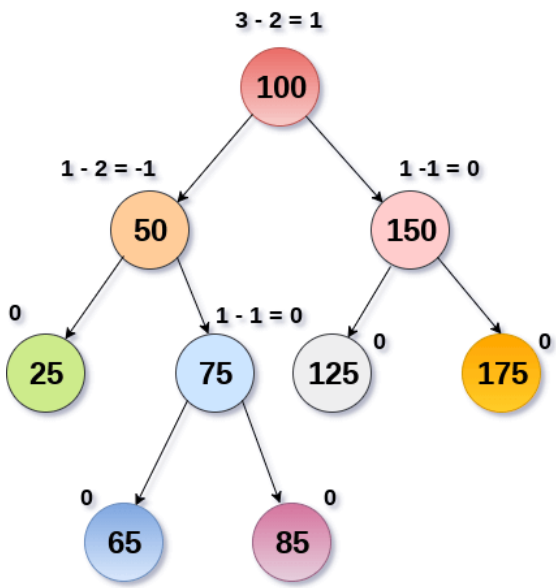
AVL Tree: AVL tree can be defined as a height balanced binary search tree in which each node is associated with a balance factor which is calculated by substringing the height of its right subtree from the left subtree.
Tree is said to be balanced if the balance factor of each node is in between -1 to 0 and 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

If the balance factor of any node is 1, it means that the left sub-tree is one level higher than the right subtree.

If the balance factor of any node is 0, it means that the left subtree and right subtree contain equal height.

If the balance factor of any node is -1, it means that the left sub-tree is one level lower than the right subtree.



AVL Tree

Algorithm	Average case	Worst case

Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

- Rotation: AVL trees use rotation operations to maintain balance during insertions and deletions. There are four types of rotations: left rotation, right rotation, left-right rotation (LR rotation), and right-left rotation (RL rotation). These rotations adjust the tree structure while preserving the order of the elements.

- Insertion: When inserting a new node into an AVL tree, the tree may become unbalanced. To restore balance, rotations are performed on affected nodes, starting from the newly inserted node and moving up the tree towards the root. These rotations ensure that the balance factor of each node remains within the acceptable range.

- Deletion: Similar to insertion, deletion in an AVL tree can cause imbalances. After deleting a node, rotations are performed to maintain the AVL tree's balanced property.

- Searching: AVL trees support efficient searching operations similar to regular binary search trees. The tree's balanced structure allows for faster search times compared to unbalanced trees like binary search trees.

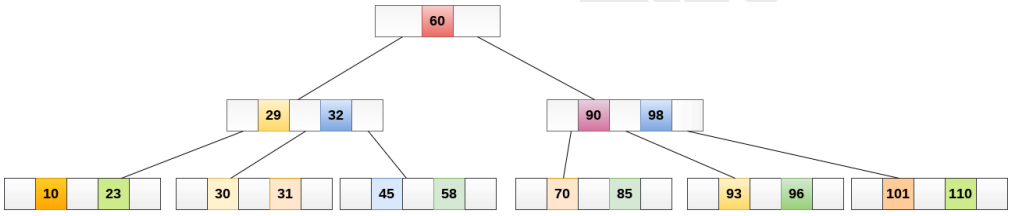
Red-Black tree: is a self-balancing binary search tree with the property that each node has an extra bit called its color, which can be either red or black. AVL tree is also a height balancing binary search tree then **why do we require a Red-Black tree**. In the AVL tree, we do not know how many rotations would be required to balance the tree, but in the Red-black tree, a maximum of 2 rotations are required to balance the tree. It contains one extra bit that represents either the red or black color of a node to ensure the balancing of the tree.

Splay tree :The splay tree data structure is also a binary search tree in which a recently accessed element is placed at the root position of the tree by performing some rotation

operations. Here, splaying means the recently accessed node. It is a self-balancing binary search tree having no explicit balance condition like AVL tree.Splay tree is a balanced tree but it cannot be considered as a height balanced tree because after each operation, rotation is performed which leads to a balanced tree.

B Tree: It is a specialized m-way tree that can be widely used for disk access. A B- Tree of order m can have at most m-1 keys and m children.One of the main reasons for using B tree is its capability to store a large number of keys in a single node and large key values by keeping the height of the tree relatively small.

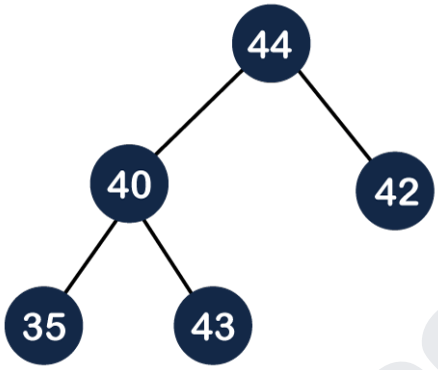
1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.



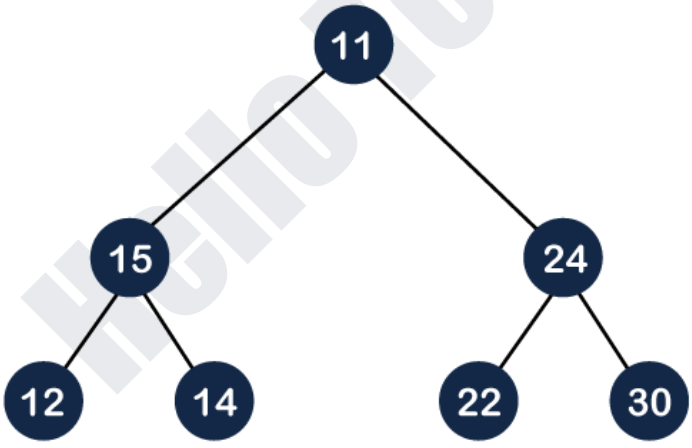
B+ Tree :It is an extension of B Tree which allows efficient insertion, deletion and search operations. It is suitable for efficient disk-based or external memory storage systems.
B+ trees are commonly used in database management systems to index large datasets efficiently. They provide efficient support for operations like insertion, deletion, and retrieval, making them well-suited for applications that require frequent updates and queries on large datasets, such as file systems and database indexes.

Heap

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e, leaf node, should be completely filled, and all the nodes should be left-justified.



Max heap



Min Heap

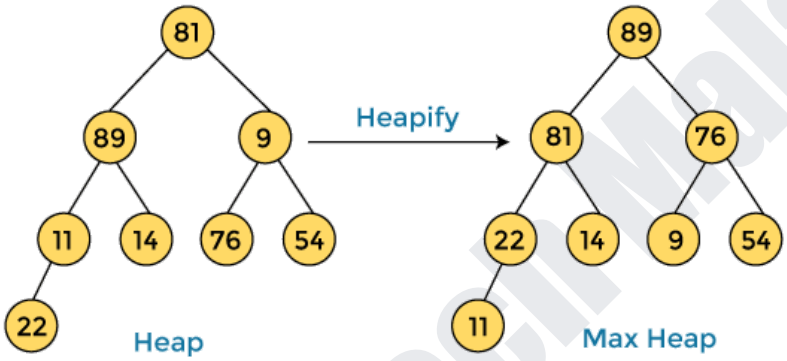
Heap Sort : Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.
Heapsort is the in-place sorting algorithm.

Heapsort algorithm:

1. The first step includes the creation of a heap by adjusting the elements of the array.
2. After the creation of the heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

81	89	9	11	14	76	54	22

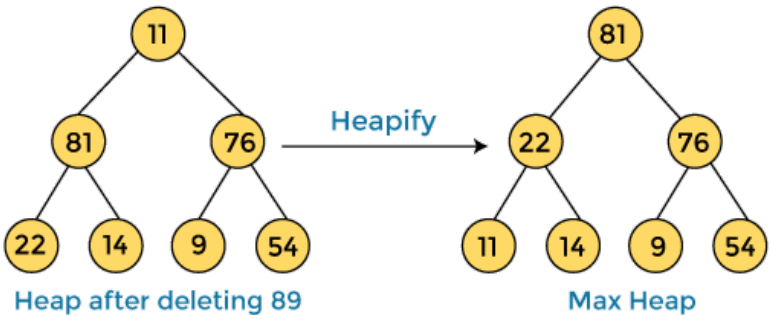
First, we have to construct a heap from the given array and convert it into max heap.



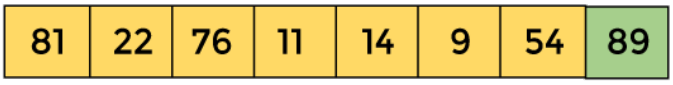
After converting

89	81	76	22	14	9	54	11

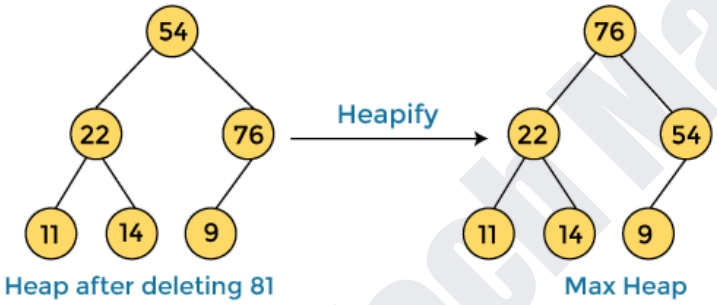
Next, we have to delete the root element(89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it into max heap.

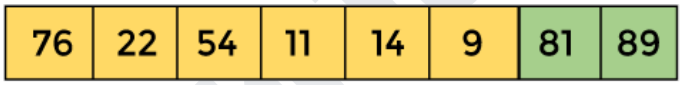


After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are

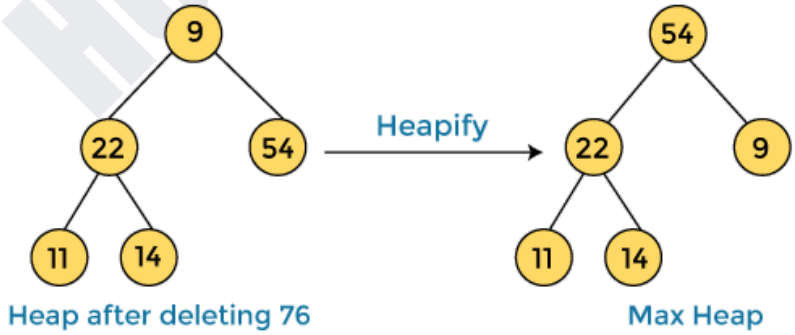


Swap 89 to 54

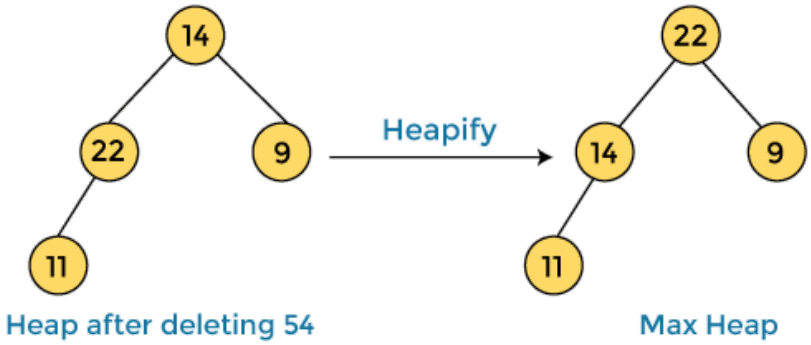




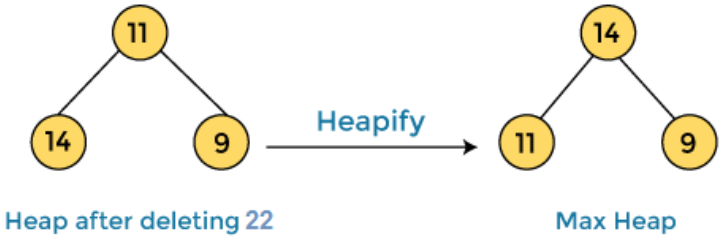
swap 76 to 9

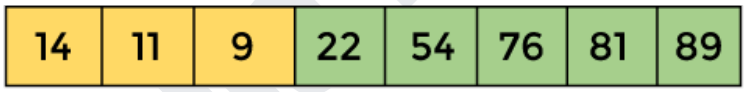


Swap 54 to 14

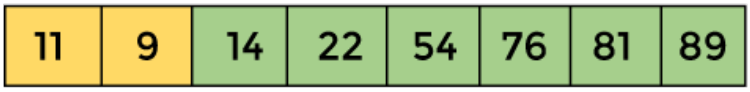


Swap 22 to 11

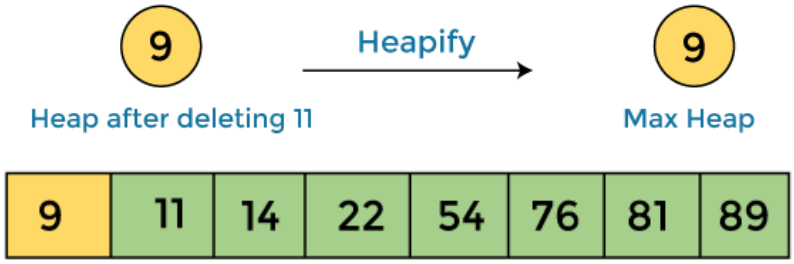




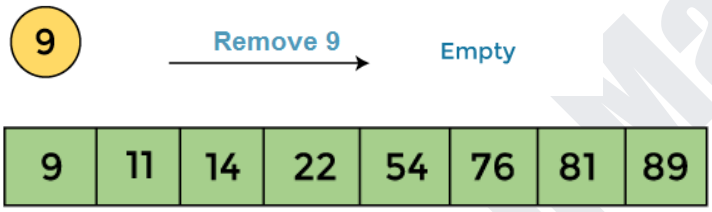
Swap 14 to 9



Swap 11 to 9



Now only one element left. After deleting it heap will be empty.



Case	Time Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)

Space Complexity	O(1)
Stable	N0

Max Heap

```
class MaxHeap {
  List maxHeap = [];

  int parent(int i) {
    return (i - 1) ~/ 2;
  }

  void insert(int index) {
    maxHeap.add(index);
    shiftUp(maxHeap.length - 1);
  }

  void shiftUp(int index) {
    int parentIndex = parent(index);
    while (index > 0 && maxHeap[parentIndex] < maxHeap[index]) {
      swap(maxHeap, parentIndex, index);
      index = parentIndex;
      parentIndex = parent(index);
    }
  }

  void swap(List arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  void display() {
    maxHeap.forEach((element) => print(element));
  }
}

void main() {
  List arr = [10, 80, 40, 30, 90];
  MaxHeap heap = MaxHeap();
  arr.forEach((element) => heap.insert(element));
  heap.display();
}
```

Result:
90

80
40
10
30

Min Heap

```
class MinHeap {
  List minHeap = [];
  void insert(int index) {
    minHeap.add(index);
    shiftUp(minHeap.length - 1);
  }

  int parentIdx(int i) {
    return (i - 1) ~/ 2;
  }

  void display() {
    minHeap.forEach((element) => print(element));
  }

  void shiftUp(int index) {
    int parentIndex = parentIdx(index);
    while (index > 0 && minHeap[parentIndex] > minHeap[index]) {
      swap(minHeap, parentIndex, index);
      index = parentIndex;
      parentIndex = parentIdx(index);
    }
  }

  void swap(List arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }
}

void main() {
  List arr = [70, 10, 30, 5, 50];
  MinHeap heap = MinHeap();
  arr.forEach((element) => heap.insert(element));
  heap.display();
}
```

Result:

```
5
10
30
70
50
```

Deletion:

```
class MaxHeap {
  List maxHeap = [];

  insert(int data) {
    maxHeap.add(data);
    shiftUp(maxHeap.length - 1);
  }

  int parent(int i) {
    return (i - 1) ~/ 2;
  }

  int leftIdx(int i) {
    return (i * 2) + 1;
  }

  int rightIdx(int i) {
    return (i * 2) + 2;
  }

  void shiftUp(int index) {
    int parentIndex = parent(index);
    while (index > 0 && maxHeap[parentIndex] < maxHeap[index]) {
      swap(maxHeap, parentIndex, index);
      index = parentIndex;
      parentIndex = parent(index);
    }
  }

  void remove() {
    swap(maxHeap, 0, maxHeap.length - 1);
    maxHeap.removeLast();
    shiftDown(0);
```

```
  }

  void shiftDown(int index) {
    int leftChildIndex = leftIdx(index);
    int endIndex = maxHeap.length - 1;
    while (leftChildIndex <= endIndex) {
      int rightChildIndex = rightIdx(index);
      int shift;
      if (rightChildIndex < endIndex &&
          maxHeap[rightChildIndex] > maxHeap[leftChildIndex]) {
        shift = rightChildIndex;
      } else {
        shift = leftChildIndex;
      }
      if (maxHeap[shift] > maxHeap[index]) {
        swap(maxHeap, shift, index);
        shift = index;
        leftChildIndex = leftIdx(index);
      } else {
        return;
      }
    }
  }

  void swap(List arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  void display() {
    maxHeap.forEach((element) => print(element));
  }
}

void main() {
  List arr = [25, 12, 55, 8, 64, 5];
  MaxHeap heap = new MaxHeap();
  arr.forEach((element) => heap.insert(element));
  heap.display();
  heap.remove();
  print('');
  heap.display();
}
```

Sort:

```
void main() {
  List arr = [10, 20, 12, 40, 50, 60];
  int size = arr.length;
  for (int i = (size ~/ 2) - 1; i >= 0; i--) {
    heapify(arr, size, i);
  }
  print(arr);
  for (int i = arr.length - 1; i >= 0; i--) {
    swap(arr, 0, i);
    heapify(arr, i, 0);
  }
  print(arr);
}

void heapify(List heap, int size, int i) {
  int largest = i;
  int leftChild = (2 * i) + 1;
  int rightChild = (2 * i) + 2;

  if (leftChild < size && heap[leftChild] > heap[largest]) {
    largest = leftChild;
  }
  if (rightChild < size && heap[rightChild] > heap[largest]) {
    largest = rightChild;
  }
  if (largest != i) {
    swap(heap, i, largest);
    heapify(heap, size, largest);
  }
}

void swap(List heap, int i, int j) {
  int temp = heap[i];
  heap[i] = heap[j];
  heap[j] = temp;
}
```

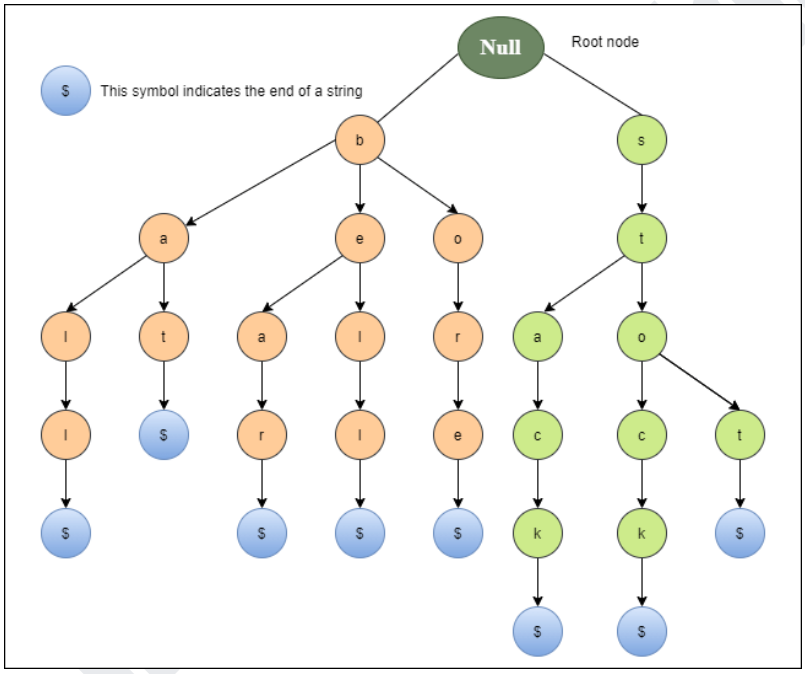
Trie data structure (O(length)**)**

A trie also known as a prefix tree or a digital tree is a tree based data structure that is primarily used for efficient string searching. It provides a way to store and retrieve strings with optimal time complexity.

In a Trie, each node represents a character or a part of a string. The root node represents an empty string, and each child node represents a character in the string. The edges connecting the nodes represent the characters themselves.

It has the number of pointers equal to the number of characters of the alphabet in each node. It can search a word in the dictionary with the help of the word's prefix. For example, if the English alphabet, each trie node can have a maximum of 26 points.

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



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

1. Every letter of the input key(words) is inserted as an individual in the Tre_node, Note that children point to the next level of Trie Nodes.
2. The key character array acts as an index of children.

3. If the present node already has a reference to the present letter, set the present node to that referenced node. Otherwise, create a new node, set the letter to be equal to the present letter, and even start the present node with this new node.
4. The character length determines the depth of the trie.

Deletion:

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

Contains: The value is in the trie ? Yes or No / like searching.

```
import 'dart:collection';

class TrieNode {
  Map<String, TrieNode> children = HashMap();
}

class Trie {
  TrieNode root = TrieNode();
  String endSymbol = "**";

  void insert(String str) {
    TrieNode node = root;
    for (int i = 0; i < str.length; i++) {
      String letter = str[i];
      if (!node.children.containsKey(letter)) {
        TrieNode newNode = TrieNode();
        node.children[letter] = newNode;
      }
      node = node.children[letter]!;
    }
    node.children[endSymbol] = TrieNode();
  }

  bool contains(String str) {
    TrieNode node = root;
    for (int i = 0; i < str.length; i++) {
      String letter = str[i];
      if (!node.children.containsKey(letter)) {
        return false;
      }
      node = node.children[letter]!;
    }
```

```
    return node.children.containsKey(endSymbol);
  }

  bool delete(String str) {
    return _deletHelper(str, root, 0);
  }

  bool _deletHelper(String str, TrieNode node, int index) {
    if (index == str.length) {
      if (!root.children.containsKey(endSymbol)) {
        return false;
      }
      node.children.remove(endSymbol);
      return node.children.isEmpty;
    }
    String letter = str[index];
    TrieNode childDelete = node.children[letter]!;
    bool shouldDelete = _deletHelper(str, childDelete, index + 1);
    if (shouldDelete) {
      node.children.remove(letter);
      return node.children.isEmpty;
    }
    return false;
  }
}

void main() {
  Trie trie = Trie();
  trie.insert('joyal');
  print(trie.contains('joyal'));
}
```

Applications:

1. Spell checker : Spell checking is a three - step process. First look for that word in a dictionary, generate possible suggestions, and then sort the suggestion words with the desired word at the top.

   Trie is used to store the word om dictionaries. The spell checker can easily be applied in the most efficient way by searching for words on a data structure. Using trie not only makes it easy to see the word in the dictionary, but it is also simple suggestions.

2. Auto-Complete : This functionality is widely used on text editors, mobile applications, and the internet. It provides a simple way to find an alternative word to complete the word for the following reasons.

- It provides an alphabetical filter of entries by the key of the node.
- We trace pointers only to get the node that represents the string entered by the user.
- As soon as you start typing, it tries to complete your input.

3. Browser history: It is also used to complete the URL in the browser. The browser keeps a history of the URLs of the websites you've visited.

Advantages of Trie

- It can insert faster and search the string than hash tables and binary search trees.
- It provides an alphabetical filter of entries by the key of the node.

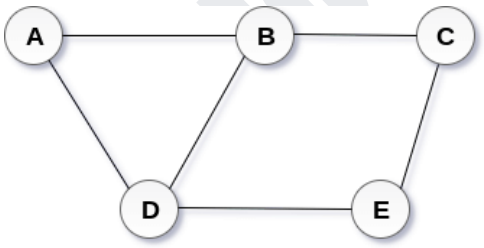
Disadvantages of Trie

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

Graph

A graph can be defined as a group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices(Nodes) maintain any complex relationship among them inserted of parent-child relationship.

A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

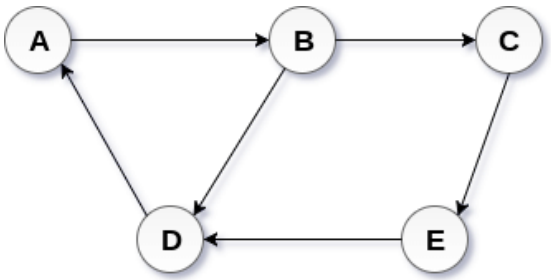


Undirected Graph

A graph G(V, E) with 5 vertices (A,B,C,D,E) and six edges ((A,B), (B,C),(C,E),(E,D),(D,B),(D,A))

Directed and Undirected Graph

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with tem. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.



Directed Graph

In a directed graph, edges from an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called the initial node while node B is called terminal node.

Path: A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed path: A path will be called a closed path if the initial node is the same as the terminal node. A path will be a closed path if V0 = VN.

Simple path: If all the nodes of the graph are distinct with an exception V0 = VN, then such path P is called a closed simple path.

Cycle: A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph: A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in the connected graph.

Complete graph: A complete graph is the one in which every node is connected with all other nodes. A complete graph contains n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph: In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive(+) value indicating the cost of traversing the edge.

Digraph: A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop/Self Loop: An edge that is associated with the similar endpoints can be called a Loop.

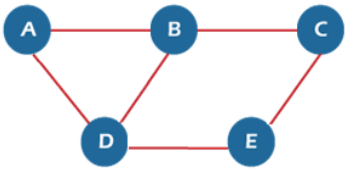
Adjacent Nodes: If two nodes u and v are connected via an edge e, then the nodes u and v are called neighbors or adjacent nodes.

Degree of the Node: A degree of a Node is the number of edges that are connected with that node. A node with degree 0 is called an isolated node.

Graph representation :

- Adjacency matrix
- Adjacency list

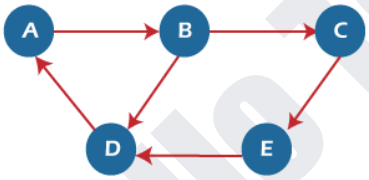
Matrix



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

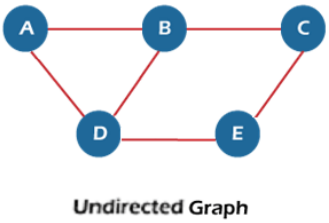


Directed Graph

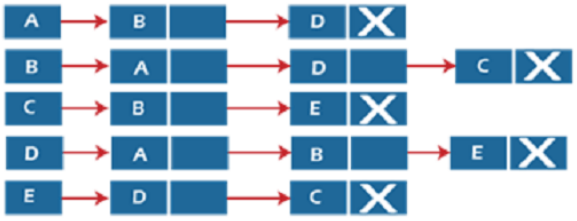
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

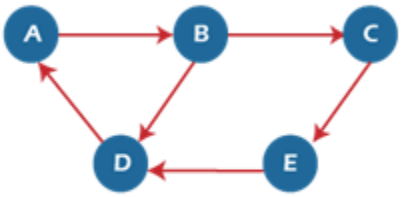
List



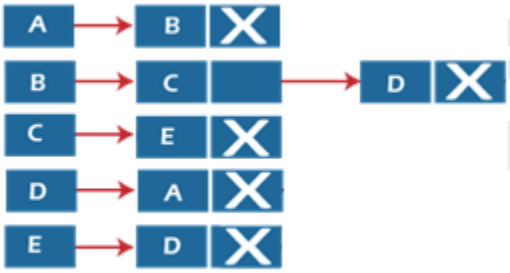
Undirected Graph



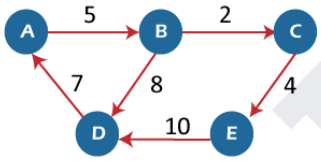
Adjacency List



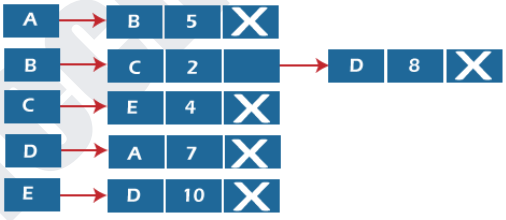
Directed Graph



Adjacency List



Weighted Directed Graph



Adjacency List

```
import 'dart:collection';

Map<int, List<dynamic>> graph = HashMap();

void insert(int vertex) {
  if (graph.containsKey(vertex)) {
    print('The $vertex is already exist in the graph');
  } else {
    graph[vertex] = [];
  }
}
```

```
void add_edge(int vertex1, int vertex2) {
  if (!graph.containsKey(vertex1)) {
    print('The $vertex1 is not in the graph');
  } else if (!graph.containsKey(vertex2)) {
    print('The $vertex2 is not in the graph');
  } else {
    graph[vertex1]!.add(vertex2);
    graph[vertex2]!.add(vertex1);
  }
}

void add_edge_weighted(int vertex1, int vertex2, int weight) {
  if (!graph.containsKey(vertex 1) && !graph.containsKey(vertex2)) {
    print('The vertices not in the graph');
  } else {
    graph[vertex1]!.add([vertex2, weight]);
    graph[vertex2]!.add([vertex1, weight]);
  }
}

bool vertexConnection(int vertex1, int vertex2) {
  if (!graph.containsKey(vertex 1) && !graph.containsKey(vertex2)) {
    print('Vertices not in the graph');
    return false;
  }
  if (graph[vertex1]!.contains(vertex2)) {
    return true;
  } else if (!graph[vertex2]!.contains(vertex1)) {
    return true;
  }
  print("$vertex and $vertex2 are not connected");
  return false;
}

void main() {
  insert(10);
  insert(20);
  insert(30);
  add_edge(10, 20);
  add_edge(20, 30);
  add_edge(10, 30);
  add_edge_weighted(10, 20, 1);
  add_edge_weighted(20, 30, 2);
  add_edge_weighted(10, 30, 3);
```

```
  print(graph);
  print(vertexConnection(10, 20));
}
```

BFS (Breadth-First Search) is another popular graph traversal algorithm. Unlike DFS, which explores vertices as deeply as possible, BFS explores the vertices in a breadth-wise manner, visiting all vertices at the current depth before moving to the next depth level.

Here's a step-by-step explanation of the BFS algorithm:

1. Start at a specific vertex: Choose a starting vertex from where you want to begin the traversal.

2. Enqueue the starting vertex: Add the starting vertex to a queue data structure. The queue will be used to keep track of the vertices to be visited.

3. Mark the starting vertex as visited: Mark the starting vertex as visited to keep track of the vertices that have been explored.

4. While the queue is not empty:
   - Dequeue a vertex from the queue: Remove a vertex from the front of the queue.
   - Visit the dequeued vertex: Process the dequeued vertex (e.g., print it, perform some operation).
   - Enqueue adjacent unvisited vertices: Add all the adjacent vertices of the dequeued vertex that have not been visited to the queue and mark them as visited.

5. Repeat step 4 until the queue is empty: Continue the process of dequeuing vertices, visiting them, and enqueueing their unvisited adjacent vertices until the queue becomes empty.

*****

The BFS algorithm ensures that vertices at the same depth are visited before moving on to vertices at the next depth level. This results in a level-by-level traversal of the graph or tree.

BFS is commonly used to solve problems like finding the shortest path between two vertices, determining the connected components of a graph, or finding all vertices within a certain distance from a given vertex.

It's important to note that BFS guarantees the shortest path for unweighted graphs. If there are multiple paths of the same length, BFS will visit them in the order they are discovered, ensuring the shortest path is found.

Overall, BFS is a versatile algorithm for exploring and traversing graphs or trees, providing a breadth-first approach that systematically visits vertices at each depth level.

DFS (Depth-First Search) is an algorithm used to traverse or search through a graph or tree data structure. It explores as far as possible along each branch before backtracking. DFS can be implemented recursively or using a stack.

Here's a step-by-step explanation of the DFS algorithm:

1. Start at a specific vertex: Choose a starting vertex in the graph or tree from where you want to begin the traversal.

2. Mark the current vertex as visited: Mark the current vertex as visited to keep track of the vertices that have been explored.

3. Explore adjacent unvisited vertices: From the current vertex, choose an adjacent vertex that has not been visited yet. Move to this vertex and repeat steps 2 and 3 recursively.

4. Backtrack if necessary: If there are no more unvisited adjacent vertices, backtrack to the previous vertex and continue the exploration from there. This involves popping the previous vertex from the stack or returning from the current recursive function call.

5. Repeat steps 2 to 4 until all vertices are visited: Continue the process of visiting adjacent unvisited vertices and backtracking until all vertices in the graph or tree have been visited.

The DFS algorithm follows a depth-first approach, meaning it explores as far as possible along each branch before moving on to the next branch. This results in a traversal that goes deeper into the graph or tree before exploring other branches.

DFS is often used to solve problems involving graph traversal, such as finding connected components, detecting cycles, or determining paths between vertices. It can also be used to generate a topological ordering of a directed acyclic graph (DAG).

It's worth noting that DFS does not necessarily visit vertices in a specific order unless additional constraints or sorting mechanisms are applied. The order of visited vertices depends on the structure of the graph and the order in which adjacent vertices are explored.

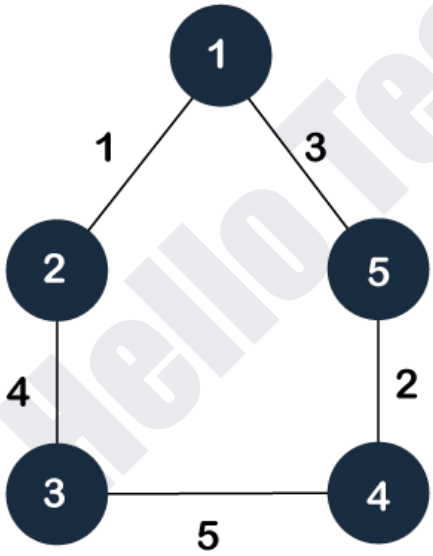
Overall, DFS is a powerful algorithm for exploring and traversing graphs or trees, providing a systematic way to visit all vertices in a depth-first manner.

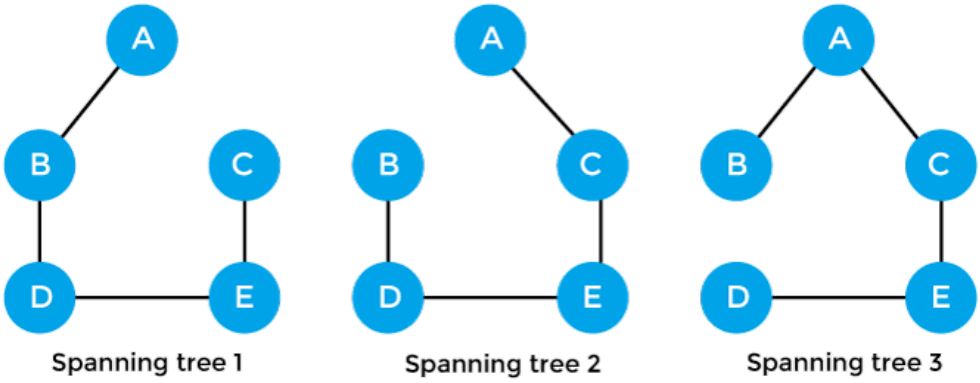
Spanning Tree: A spanning tree of a connected graph is a subgraph that includes all the vertices of the original graph while formatting a tree structure with no cycles. In other words, it is a subset of the original graph that connects all the vertices with the minimum number of edges.

1. A spanning tree has exactly n-1 edges, where n is the number of vertices in the original graph.
2. It is acyclic, meaning there are no cycles in the spanning tree.
3. It is connected, ensuring that there is a path between any two vertices in the spanning tree.
4. Removing any edge from the spanning tree will disconnect it.

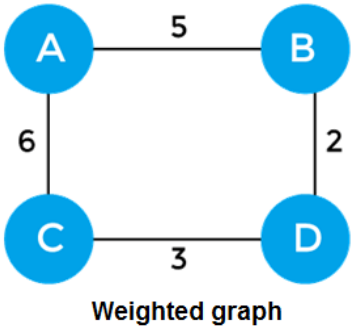
Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows;

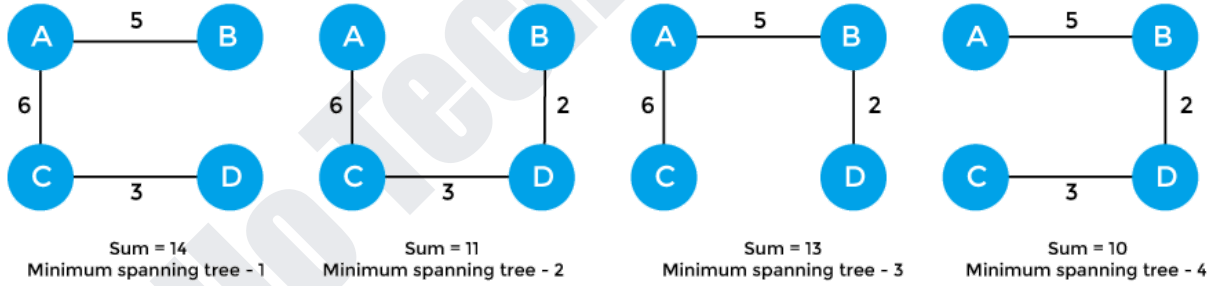
- Cluster Analysis
- Civil network planning
- Computer network routing protocol





Minimum Spanning tree: A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.





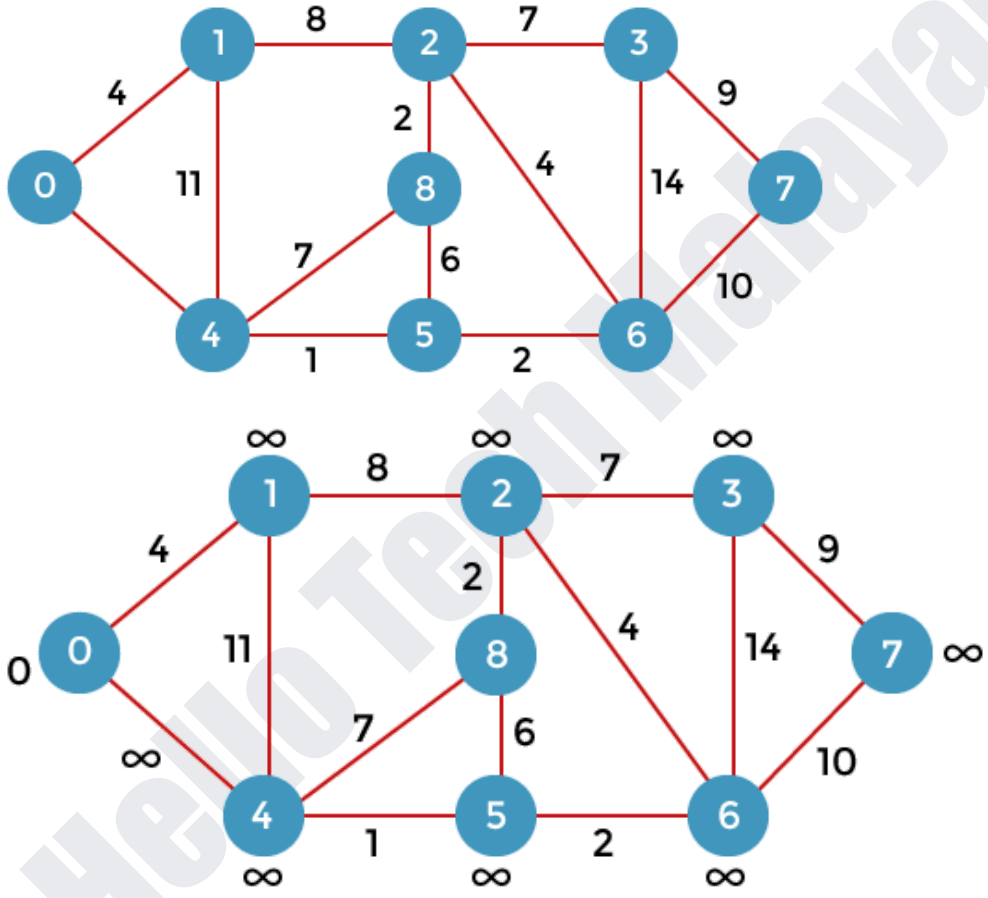
The minimum spanning tree is last(the sum = 10).

- Minimum spanning trees can be used to design water-supply networks, telecommunication networks, and electrical grids.
- It can be used to find paths in the map.

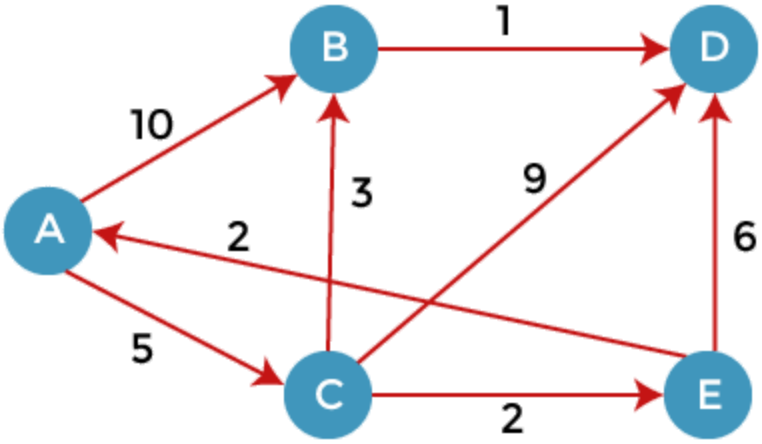
Prim's algorithm - It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.
Time complexity O(V^2)

Kruskal's algorithm - This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows a greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.
Time complexity O(E log V)

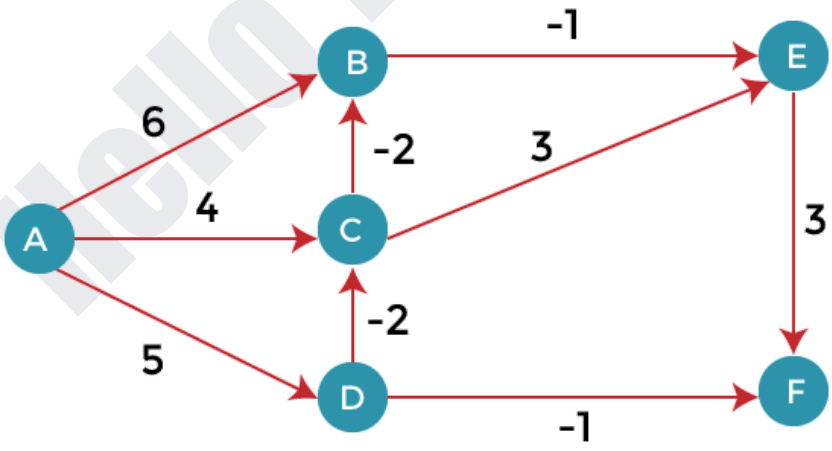
Dijkstra's Algorithm: It is a single-source shortest path algorithm. Here, single source means that only one source is given, and we have to find the shortest path from the source to all the nodes.



1. {**if**(d(u) + c(u, v) < d(v))
2. d(v) = d(u) +c(u, v) }



Bellman Ford Algorithm: Bellman ford algorithm is a single-source shortest path algorithm. This algorithm is used to find the shortest distance from the single vertex to all the other vertices of a weighted graph. There are various other algorithms used to find the shortest path like Dijkstra algorithm, etc. If the weighted graph contains the negative weight values, then the Dijkstra algorithm does not confirm whether it produces the correct answer or not. In contrast to Dijkstra algorithm, the bellman ford algorithm guarantees the correct answer even if the weighted graph contains the negative weight values.



Graph code:

```
import 'dart:collection';
import 'dart:io';

Map<int, List<dynamic>> graph = HashMap();
void insert(int v1) {
  if (graph.containsKey(v1)) {
    print('The node is already present in the graph');
  } else {
    graph[v1] = [];
  }
}

void add_edge(int vertexFrom, int vertexTo) {
  if (!graph.containsKey(vertexFrom)) {
    print('$vertexFrom is not present in the graph');
  } else if (!graph.containsKey(vertexTo)) {
    print('$vertexTo is not in the graph');
  } else {
    graph[vertexFrom]!.add(vertexTo);
    graph[vertexTo]!.add(vertexFrom);
  }
}

void add_edge_weighted(int vertex1, int vertex2, int weight) {
  if (!graph.containsKey(vertex1)) {
    print("$vertex1 not in the graph");
  } else if (!graph.containsKey(vertex2)) {
    print('$vertex2 not in the graph');
  } else {
    graph[vertex1]!.add([vertex1, weight]);
    graph[vertex2]!.add([vertex1, weight]);
  }
}

bool vertexConnection(int vertex1, int vertex2) {
  if (!graph.containsKey(vertex1)) {
    print("$vertex1 is not in the graph");
    return false;
  } else if (!graph.containsKey(vertex2)) {
    print("$vertex2 is not in the graph");
    return false;
  }
  if (graph[vertex1]!.contains(vertex2)) {
    return true;
```

```
  } else if (graph[vertex2]!.contains(vertex1)) {
    return true;
  }
  print('$vertex1 and $vertex2 are not connceted');
  return false;
}

void DFS(int vertex) {
  if (!graph.containsKey(vertex)) {
    print('Vertex not in the graph');
    return;
  }
  Map<int, bool> visited = {};
  for (int v in graph.keys) {
    visited[v] = false;
  }
  _DFS(vertex, visited);
}

void _DFS(int vertex, Map<int, bool> visited) {
  visited[vertex] = true;
  stdout.write('$vertex --> ');
  for (int n in graph[vertex]!) {
    if (!visited[n]!) {
      _DFS(n, visited);
    }
  }
}

void BFS(int vertex) {
  Map<int, bool> visited = {};
  for (int v in graph.keys) {
    visited[v] = false;
  }

  Queue queue = new Queue();
  visited[vertex] = true;
  queue.add(vertex);

  while (queue.isNotEmpty) {
    int node = queue.removeFirst();
    stdout.write("$node --> ");
    List<dynamic>? connections = graph[node];
    if (connections != null) {
```

```
      for (var connection in connections) {
        if (!visited[connection]!) {
          visited[connection] = true;
          queue.add(connection);
        }
      }
    }
  }
}

void print_graph() {
  graph.forEach((key, value) => print("$key : $value"));
}

void main() {
  insert(10);
  insert(20);
  insert(30);
  add_edge(10, 20);
  add_edge(30, 10);
  print_graph();
  print(vertexConnection(10, 20));
  DFS(10);
  print('');
  BFS(10);
}
```