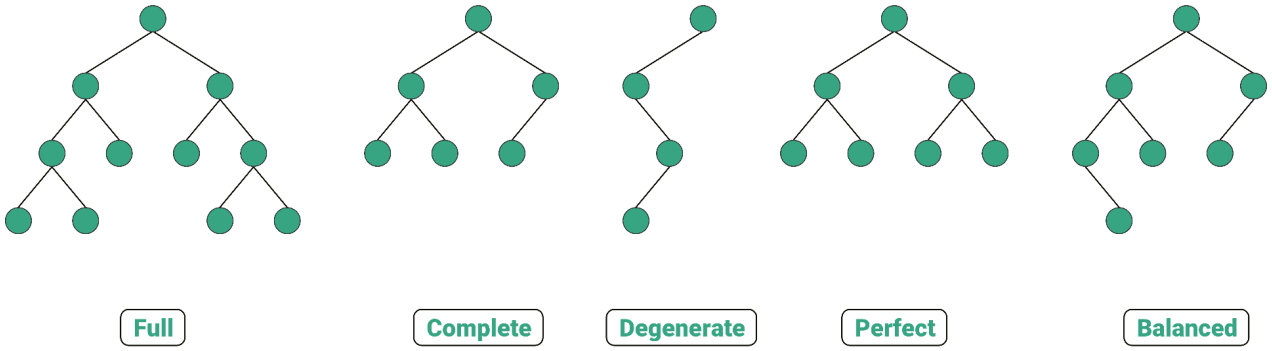
Binary Tree

Binary Trees are non linear data structures comprising of nodes. Each node has it's own data and pointers to left and right child nodes.

Types:

- **Basis of Number of Children:**
  - Full BT: each node has either 0 or 2 children.
  - Degenerate BT: each parent node has 1 child only. Like a linked list.
  - Skewed BT: a type of Degenerate BT, it has either 1 or no child such that all child are of same type (left skewed or right skewed).



- **Basis of Completion of Level:**
  - Complete BT: all the levels of BT are completely filled except the last which also fills from left.
    - **Root:** which has no parent node or no incoming edge.
    - **Child:** which has some incoming edge to it.
    - **Siblings**: sharing same parent node.
    - **Degree of Node**: number of direct childrens.
    - **External and Internal Nodes**: Leaf are external and rest internal nodes.
    - **Level**: count nodes from root to destination node (start from root as 0).
    - **Height**: count edges from root to destination node.
    - **Depth**: defined for whole tree, max height of any node.
    - In complete BT all leaves are at same depth.
    - Nodes in CBT at depth 'd' are $\mathbf{2^d}$.
    - In a CBT having n nodes, height of tree is **log (n + 1)**.
  - **Perfect BT:** all levels are completely filled, each node has two children.
  - **Balanced BT:** BT where the absolute difference between heights of children is not more than 1. And same applies to it's subtrees.
- **Basis of Node Value:**
  - **BST**: binary search Trees follow the pattern such that the value of left children is strictly less than root and the value of right children is strictly more than root. Same applies to it's subtrees.

- **AVL Trees:**
  - Self-balancing binary search tree.
  - Height difference between left and right subtrees of any node is at most 1.
  - Guarantees efficient search, insertion, and deletion operations.

- **Red-Black Trees**:
  - Self-balancing binary search tree.
  - Uses colors (red and black) to maintain balance.
  - Provides good performance guarantees, but slightly less strict balancing than AVL trees.
  - **Usecases:**
    - These are quite fast and so used in cache lookup for frequent data access
    - Used in databases for fast indexing
    - Also used in routing algorithms
- **B Trees**:
  - Balanced tree that allows multiple keys and children in a node.
  - Optimized for disk-based storage.
  - Used in databases and file systems for efficient data retrieval.
  - **Usecases:**
    - Same as AVL trees
- **B+ Trees:**
  - Variation of B-trees where all data is stored in leaf nodes.
  - Internal nodes only store keys for directing searches.
  - Excellent for indexing and sequential access.
  - **Usecases:**
    - Same as AVL trees
- **Segment Trees:**
  - Used for efficient range query operations.
  - Each node represents a range of data.
  - Supports operations like finding the sum, minimum, maximum, etc., within a given range.
  - **Usecases:**
    - Range based statistics like, percentile, variance, standard deviation
    - Image processing for dividing an image based on color, texture
- **B* trees:**
  - Variation of B-trees with higher minimum and maximum number of keys per node.
  - Aims to improve space utilization and reduce the number of disk accesses.
  - **Usecases:**
    - Same as AVL trees

Implementation of BT:

- Code

```
class Node {
public:
```

```
    int data;
    Node *left;
    Node *right;

    Node(int value) {
        this->data = value;
        this->left = this->right = NULL;
    }
};
```

Insertion in Binary Tree:

- No fixed approach:
  - Check if left child exists, if not insert new node there.
  - Check if right child exists, if not insert new node there.
  - If not found on either left or right, then iterate to left subtree or right subtree (based on your choice).
- Fixed approach (Level Order):
  - To insert node at first empty found place.
  - Use a queue and insert root.
  - Check if left subtree exists, if yes insert into queue. If not then place the node there.
  - Similarly check for right subtree.

```
static Node *build_tree(Node *&root, int value) {
    if (!root) {
        return new Node(value);
    }
    if (!root->left) {
        root->left = new Node(value);
        return root;
    }
    if (!root->right) {
        root->right = new Node(value);
        return root;
    }
    if (root->left) {
        root->left = build_tree(root->left, value);
    } else {
        root->right = build_tree(root->right, value);
    }
    return root;
}
```

Traversal Techniques:

- Depth First Search (DFS):
  - **Preorder**:
    - Traversal is like Node-Left-Right, visit node then go to left subtree and then to right subtree.
    - Code

```
static void preorder(Node *&root) {
    if (!root) {
        return;
    }
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}
```

  - **Postorder**:
    - Traversal is like Left-Right-Node
    - Code

```
static void postorder (Node *&root) {
    if (!root) {
        return;
    }
    postorder (root->left);
    postorder (root->right);
    cout<< root->data<< " ";
}
```

  - **Inorder**:
    - Traversal is like Left-Node-Right
    - Code

```
static void inorder (Node *&root) {
    if (!root) {
        return;
    }
    inorder (root->left);
    cout<< root->data<< " ";
    inorder (root->right);
}
```

- Breadth First Search (BFS):
  - **Level Order traversal**:
    - We traverse each level from left to right or vice versa
    - Code

```
static void level_order(Node *&root) {
    if (!root) {
```

```
            cout << "Empty tree!";
            return;
        }

        queue<Node *> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; i++) {
                Node *front = q.front();
                q.pop();
                cout << front->data << " ";
                if (front->left) {
                    q.push(front->left);
                }
                if (front->right) {
                    q.push(front->right);
                }
            }
            cout << endl;
        }
    }
```

Deletion from BT:

- Deletion of leaf nodes (external nodes) is very simple it can be done without shifting any node.
- But deletion of nodes in between the tree, the internal nodes is quite tricky. You gotta find the right most node and replace the data with to_delete node and delete the right one node.
- Code

```
static void find_deepest(Node *&root, Node *&deepest, Node *&parent) {
        queue<Node *> q;
        q.push(root);
        while (!q.empty()) {
            Node *front = q.front();
            q.pop();
            if (!front->left && !front->right) {
                deepest = front;
            }
            if (front->left) {
                q.push(front->left);
                parent = front;
            } else if (front->right) {
                q.push(front->right);
```

```
                parent = front;
            }
        }
    }

    static void delete_node(Node *&root, int value) {
        if (!root) {
            return;
        }
        if (root->data == value) {
            if (!root->left && !root->right) {
                root = NULL;
                delete root;
                return;
            }
            Node *parent = NULL;
            Node *deepest = NULL;
            find_deepest(root, deepest, parent);
            root->data = deepest->data;
            if (parent->left == deepest) {
                parent->left = NULL;
                delete deepest;
            } else if (parent->right == deepest) {
                parent->right = NULL;
                delete deepest;
            }
            return;
        }
        delete_node(root->left, value);
        delete_node(root->right, value);
    }
```

Searching in a BT:

- Normal search operation, can use any of traversals we studied above.

Morris Traversal:

Questions:

1. Height of BT:
    - Going to use recursion
    - Find height of left subtree
    - Find height of right subtree
    - Find max of left and right, add 1 of current root node

    - Return final value
2. Level of a target node:
    - We'll use level order traversal, set initial level = 1 and increase after each for loop
    - If we find front->data == target, return level
    - Else return 0
3. Get size of BT:
    - We'll use recursion, find size of left subtree, then right subtree, add 1 to answer and return
    - If null node, return 0
    - If leaf node, return 1

Operations	Time Complexity	Space Complexity
Insertion	O(N)	O(N)
Preorder Traversal	O(N)	O(N)
Inorder Traversal	O(N)	O(N)
Postorder Traversal	O(N)	O(N)
Level Order Traversal	O(N)	O(N)
Deletion	O(N)	O(N)
Searching	O(N)	O(N)

Applications/ Advantages and Disadvantages of BTs:

- Applications (most used BT is a BST):
    - File Systems are stored in a tree structure and root folder at root node
    - Database Systems also use BSTs for searching and sorting records
    - Sorting algorithms also utilize binary search trees as insertion in O(log n)
    - DOM in HTML, it's a tree structure with childrens arranged (it used BT)
    - Routing tables utilze variant of BT called trie to link routers in a network
    - File explorers in almost every OS
    - BTs are also utilized in ranking of web pages
    - Google Servers also utilize a variation of BTs called trie for providing fast results
    - Data Compression
        - Huffman encoding, leaves represent characters and their frequency of occurence
- Advantages:
    - Efficient searching
    - Ordered traversals

    - Fast insertion and deletion
    - Easy to implement
    - Useful in places that require sorting
- Disadvantages:
    - A slight unbalance in the tree can make sub-structure worthless
    - Balancing algorithms in AVL, red-black trees are complex to implement

Binary Search Trees

These are a variant of Binary Trees but with the property of holding a sorted structure in their values. The value of left node < root node < right node, this exists for each node except for leaf nodes, i.e., both left and right subtrees are also BSTs.

Generally there aren't any duplicates allowed in BSTs, but if allowed then should be kept either to left or to right (fixed fashion to avoid confusion).

Structure:

```
class Node {
public:
    int data;
    Node *left;
    Node *right;

    Node(int value) {
        this->data = value;
        this->left = this->right = NULL;
    }
};
```

Build height balanced BST from array:

- Code:

```
static Node *build_tree(Node *&root, vector<int> &arr, int start, int end) {
    if (start > end) {
        return NULL;
    }

    int mid = (start + end) / 2;
    root = new Node(arr[mid]);
    root->left = build_tree(root->left, arr, start, mid - 1);
    root->right = build_tree(root->right, arr, mid + 1, end);
    return root;
}
```

Deletion in a BST:

- There are 3 possibilities when it comes to delete a node in a BST.
- 0 child case, 1 child case and 2 child case.
- If 0 children, then delete node and return NULL
- If 1 children, store whichever left node or right one child, delete node and return child
- If 2 children, find next inorder predecessor of node, make node->data = pred->data, then recursively delete predecessor

- Code:

```
static Node *deleteNode(Node *root, int x) {
    if (!root) {
        return NULL;
    }

    if (root->data == x) {

        // 0 child case
        if (!root->left && !root->right) {
            delete root;
            return NULL;
        }

        // 1 child case
        if (!root->right) {
            Node *temp = root->left;
            delete root;
            return temp;
        }

        if (!root->left) {
            Node *temp = root->right;
            delete root;
            return temp;
        }

        // 2 child case
        Node *temp = root->left;
        while (temp->right) {
            temp = temp->right;
        }

        root->data = temp->data;
        root->left = deleteNode (root->left, temp->data);
        return root;
    }

    else if (root->data > x) {
        root->left = deleteNode (root->left, x);
```

```
        }
        else {
            root->right = deleteNode (root->right, x);
        }

        return root;
    }
```

Applications/ Advantages/ Disadvantages:

- Applications same as AVL trees (databases, file systems, etc)
- Advantages:
    - Efficient Searching
    - Dynamic insertion and deletion
- Disadvantages:
    - If the structure becomes large, it becomes difficult to manage, so dividing the structure into subproblems is considered good