1. Binary tree

A binary tree is a hierarchical data structure in computer science composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node in a binary tree is called the root node. Each node in a binary tree stores a piece of data, often referred to as the "key" or "value," and may also contain references or pointers to its left and right children.

Binary trees are commonly used for representing hierarchical data such as the structure of a file system, the branching structure of decision trees, or the organization of data in databases. They are also fundamental in various algorithms and data structures, such as binary search trees, AVL trees, and heap data structures.

2. Complete binary tree

A complete binary tree is a special type of binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. In other words, all levels of the tree are filled except for perhaps the last level, which is filled from left to right.

Key properties of a complete binary tree include:

1. Every level of the tree, except possibly the last, is completely filled with nodes.
2. If the last level of the tree is not completely filled, the nodes are filled from left to right.
3. The height of a complete binary tree is always minimal for the number of nodes present.

Complete binary trees have efficient implementations and are commonly used in various applications, such as binary heaps, priority queues, and binary search trees

3. Full binary tree

A full binary tree, also known as a proper binary tree or 2-tree, is a type of binary tree in which every node other than the leaves has exactly two children. In other words, every non-leaf node in a full binary tree has exactly two children, and all leaf nodes are at the same level.

Key properties of a full binary tree include:

1. Every node has either 0 or 2 children.
2. All leaves are at the same level.
3. The number of leaf nodes is always one more than the number of internal nodes.

Full binary trees have a balanced structure, and they are often used in certain tree traversal algorithms and in implementing binary tree-based data structures such as binary search trees. They can also be useful in representing arithmetic expressions and mathematical formulas.

A binary search tree (BST) is a binary tree data structure that has the following properties:

1. **Binary Tree Structure**: Each node in a BST has at most two children, commonly referred to as the left child and the right child.
2. **Ordering Property**: For every node **n**, all nodes in the left subtree of **n** have values less than **n**, and all nodes in the right subtree of **n** have values greater than **n**. This ordering property enables efficient searching, insertion, and deletion operations

4. General tree

A general tree is a hierarchical data structure composed of nodes, where each node can have zero or more children. Unlike binary trees, which restrict each node to have at most two children, a general tree allows for an arbitrary number of children per node.

Key characteristics of a general tree:

1. **Node Structure**: Each node contains a piece of data (often referred to as the "key" or "value") and may contain references or pointers to its child nodes.
2. **Root Node**: The topmost node of the tree is called the root node. It is the entry point to access the entire tree.
3. **Parent-Child Relationship**: Nodes in a general tree are connected through parent-child relationships, where each node (except the root) has a unique parent node.
4. **Leaf Nodes**: Nodes that do not have any children are called leaf nodes. They are the terminal nodes of the tree.
5. **Height**: The height of a general tree is the length of the longest path from the root to a leaf node.
6. **Depth**: The depth of a node in a general tree is the length of the path from the root to that node.

General trees are used to represent hierarchical structures that don't necessarily adhere to the strict constraints of binary trees. Examples include file systems, organization hierarchies, XML/HTML documents, and syntax trees in parsing algorithms. Traversing and manipulating general trees typically require recursive algorithms due to the varying number of children per node.

5. Properties of binary tree

Binary trees, as fundamental data structures in computer science, possess several key properties:

1. **Root Node**: The topmost node of the binary tree, serving as the starting point for traversal and access to other nodes.
2. **Parent-Child Relationship**: Each node in a binary tree (except the root) has at most one parent node and may have zero, one, or two child nodes.

3. **Left and Right Children**: Each node can have at most two children, typically referred to as the left child and the right child. These children are ordered, with the left child usually representing smaller values and the right child representing larger values in a binary search tree.
4. **Depth and Height**: The depth of a node is the length of the path from the root to that node. The height of the tree is the length of the longest path from the root to a leaf node. These measurements help in understanding the structure and performance of operations on the tree.
5. **Leaf Nodes**: Nodes that do not have any children are called leaf nodes. They are typically found at the bottom level of the tree.
6. **Balanced vs. Unbalanced**: A binary tree is considered balanced if the heights of the two subtrees of any node differ by no more than one. If a tree is not balanced, it may lead to inefficient operations like search, insert, or delete.
7. **Traversal Methods**: Binary trees support various traversal methods, including in-order, pre-order, post-order, and level-order (BFS). These methods allow accessing or processing nodes in a particular order, facilitating different operations or analyses on the tree.
8. **Binary Search Tree (BST)**: A binary search tree is a special type of binary tree where each node's value is greater than all values in its left subtree and less than all values in its right subtree. BSTs support efficient searching, insertion, and deletion operations due to their ordered structure.

Understanding these properties is crucial for designing, implementing, and analyzing algorithms involving binary trees. They play a significant role in a wide range of applications, from organizing data efficiently to facilitating fast search and retrieval operations.

6. Binary tree traversal

Tree traversal is a fundamental operation in computer science, essential for accessing and processing all nodes in a tree data structure. There are several techniques for traversing trees, each with its own unique approach and use cases. The most common traversal techniques include:

1. **In-Order Traversal**
2. **Pre-Order Traversal**
3. **Post-Order Traversal**
4. **Level-Order Traversal**

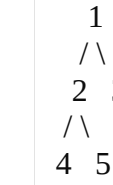
1. In-Order Traversal:
In in-order traversal, nodes are visited in the following order:

```
inOrder(node):
   if node is not null:
      inOrder(node.left)
      visit(node)
      inOrder(node.right)
```

1. Visit the left subtree.
2. Visit the current node.
3. Visit the right subtree.

In the case of binary search trees (BSTs), in-order traversal visits nodes in ascending order of their keys. This makes it particularly useful for retrieving data in sorted order.

Algorithm (Recursive):



In-order traversal of this tree would result in: **4 -> 2 -> 5 -> 1 -> 3**

2. Pre-Order Traversal:

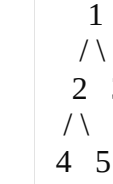
In pre-order traversal, nodes are visited in the following order:

1. Visit the current node.
2. Visit the left subtree.
3. Visit the right subtree.

Pre-order traversal is useful for creating a copy of a tree and prefix expressions (Polish notation).

Algorithm (Recursive):

```
preOrder(node):
   if node is not null:
      visit(node)
      preOrder(node.left)
      preOrder(node.right)
```



Example: Using the same example tree as above, pre-order traversal would result in: **1 -> 2 -> 4 -> 5 -> 3**

3. Post-Order Traversal:

In post-order traversal, nodes are visited in the following order:

1. Visit the left subtree.
2. Visit the right subtree.
3. Visit the current node.

Post-order traversal is useful for deleting a tree and evaluating postfix expressions (Reverse Polish notation).

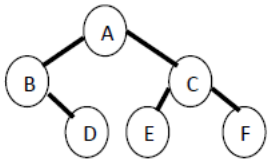
Algorithm (Recursive):

```
postOrder(node):
    if node is not null:
        postOrder(node.left)
        postOrder(node.right)
        visit(node)
```

```
     1
    / \
   2   3
  / \
 4   5
```

Example: Using the same example tree as above, post-order traversal would result in: **4 -> 5 -> 2 -> 3 -> 1**

7. Give the processing order of preorder, inorder and postorder of given tree.



Preorder: ABDCEF
Postorder: DBEFCA
Inorder: BDAECF

8. Explain the process of deletion of a node in a Binary Search Tree.

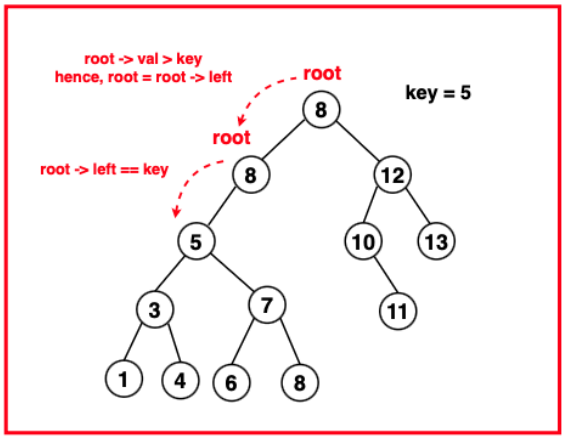
Approach:

To delete a node in a Binary Search Tree, start from the root and navigate to the node to delete based on its key. If the node is found, handle deletion based on three cases: if the node has no children, remove it; if it has one child, replace it with its child; if it has two children, find its inorder predecessor (the largest node in the left subtree), attach its right child to its parent, and connect the left child of the node to its parent's new child. Return the modified BST after deletion.

Algorithm:

Step 1: Search for the node to delete:

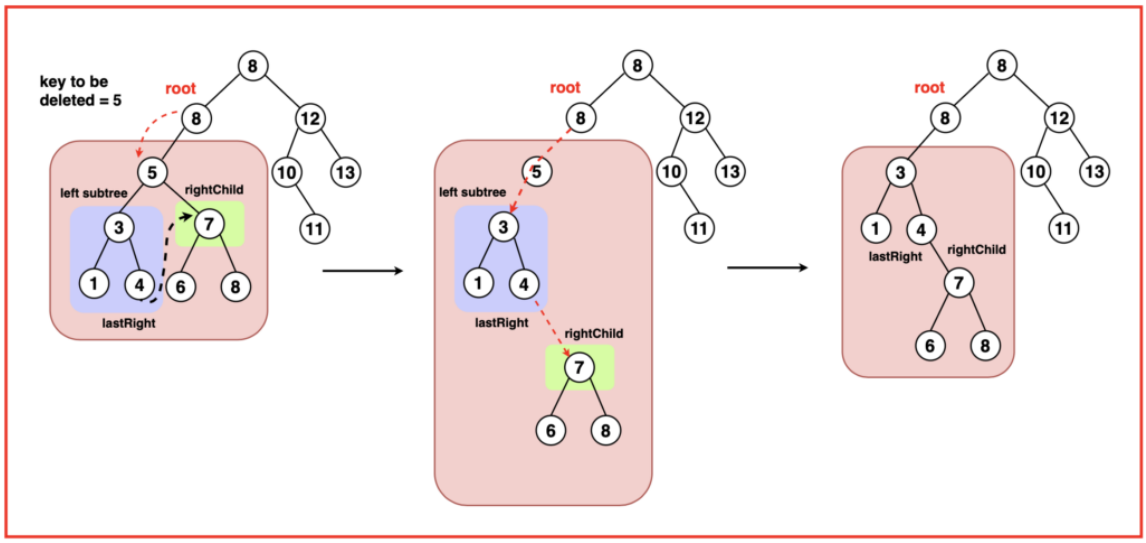
Start from the root and if the key is less than the current node, move to the left subtree and if the key is greater than the current node, move to the right subtree. Repeat this until we find the node to delete or reach a null node.



Step 2: Handle Different Cases for Deletion:
Case 1: If the node has no children (leaf nodes), simply remove the node.

Case 2: If the node has one child, replace the node to be deleted with its child.

Case 3: If the node has two children
1. Find the node's inorder predecessor by traversing the left subtree of the node to find the rightmost (largest) node. Store this as lastRight.
2. Set the right child lastRight's to the node to be deleted.
3. Skip over the node to be deleted by directly connecting the root to the node's left child ie. the root of the left subtree.



Step 3: Return the modified Binary Search Tree.