**Binary Trees**

A **Binary Tree Data Structure**is a hierarchical data structure in which each node has at most two children, the left and the right. It is commonly used in computer science for efficient data storage and retrieval of data, with various operations such as insertion, deletion, and traversal.

A diagram of a tree

Description automatically generated

The topmost node in a binary tree is called the **root**, and the bottom-most nodes are called **leaves**.

**Types of Binary Tree**

**Full Binary Tree**

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

**Complete Binary Tree**

A complete binary tree is **just like a full binary tree** but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. 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.

A diagram of numbers and circles

Description automatically generated

**Perfect Binary Tree**

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

**Degenerate or Pathological Tree**

A degenerate or pathological tree is the tree having a single child either left or right.

A screenshot of a diagram

Description automatically generated

**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 trees: **left-skewed binary trees** and **right-skewed binary trees**.

A group of blue circles with white text

Description automatically generated

**Balanced Binary Tree**

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.

A diagram of numbers and circles

Description automatically generated

**Real-time applications of Binary Trees:**

* DOM in HTML.
* File explorer.
* Used as the basic data structure in Microsoft Excel and spreadsheets.
* Editor tool: Microsoft Excel and spreadsheets.
* Evaluate an expression
* Routing Algorithms

**Advantages of Binary Tree:**

* **Efficient searching**: Binary trees are particularly efficient when searching for a specific element, as each node has at most two child nodes, allowing for binary search algorithms to be used. This means that search operations can be performed in O(log n) time complexity.
* **Ordered traversal:**The structure of binary trees enables them to be traversed in a specific order, such as in-order, pre-order, and post-order. This allows for operations to be performed on the nodes in a specific order, such as printing the nodes in sorted order.
* **Memory efficient**: Compared to other tree structures, binary trees are relatively memory-efficient because they only require two child pointers per node. This means that they can be used to store large amounts of data in memory while still maintaining efficient search operations.
* **Fast insertion and deletion:**Binary trees can be used to perform insertions and deletions in O(log n) time complexity. This makes them a good choice for applications that require dynamic data structures, such as database systems.
* **Easy to implement:** Binary trees are relatively easy to implement and understand, making them a popular choice for a wide range of applications.
* **Useful for sorting:** Binary trees can be used to implement efficient sorting algorithms, such as heap sort and binary search tree sort.

**Disadvantages of Binary Tree:**

* **Limited structure:**Binary trees are limited to two child nodes per node, which can limit their usefulness in certain applications.
* **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. This is because each node requires two child pointers.
* **Slow performance in worst-case scenarios:** In the worst-case scenario, a binary tree can become degenerate, meaning that each node has only one child. In this case, search operations can degrade to O(n) time complexity, where n is the number of nodes in the tree.
* **Complex balancing algorithms:** To ensure that binary trees remain balanced, various balancing algorithms can be used, such as AVL trees and red-black trees. These algorithms can be complex to implement and require additional overhead, making them less suitable for some applications.

**Binary Search Tree**

A binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

It is called a binary tree because each tree node has a maximum of two children.

It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

All nodes of the left subtree are less than the root node

All nodes of the right subtree are more than the root node

Both subtrees of each node are also BSTs i.e. they have the above two properties

**Time Complexity:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **Best Case Complexity** | **Average Case Complexity** | **Worst Case Complexity** |
| Search | O(log n) | O(log n) | O(n) |
| Insertion | O(log n) | O(log n) | O(n) |
| Deletion | O(log n) | O(log n) | O(n) |

**Space Complexity:**

The space complexity for all the operations is O(n).

**Binary Search Tree Applications**

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

**Basic operations on a BST**

* Create: creates an empty tree.
* Insert: insert a node in the tree.
* Search: Searches for a node in the tree.
  + **Breadth-first search (BFS):**  This algorithm is used to traverse a BST. It begins at the root node and travels in a lateral manner (side to side), searching for the desired node. This type of search can be described as O(n) given that each node is visited once and the size of the tree directly correlates to the length of the search.
  + **Depth-first search (DFS):** Start with the root node and travel down a single branch. If the desired node is found along that branch, great, but if not, continue upwards and search unvisited nodes. This type of search also has a big O notation of O(n).
* Delete: deletes a node from the tree.
* Inorder: in-order traversal of the tree.
* Preorder: pre-order traversal of the tree.
* Postorder: post-order traversal of the tree.

**Dynamic Programming**

 Used to solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

**Steps to solve:**

* **Identify Subproblems:** Divide the main problem into smaller, independent subproblems.
* **Store Solutions:**Solve each subproblem and store the solution in a table or array.
* **Build-Up Solutions:** Use the stored solutions to build up the solution to the main problem.
* **Avoid Redundancy:** By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

**When to Use Dynamic Programming**

* 1. **Optimal Substructure:** combine the optimal results of subproblems to achieve the optimal result of the bigger problem.

**Example**: Consider an undirected graph with vertices a, b, c, d, e and edges (a, b), (a, e), (b, c), (b, e),(c, d) and (d, a) with some respective weights. Find the shortest path between a and c.

This problem can be broken down into finding the shortest path between a & b and then the shortest path between b & c and this can give a valid solution i.e. shortest path between a and c. We need to break this for all vertices between a & c to check the shortest and also direct edges a-c if exits.

* 1. **Overlapping Subproblems:** The same subproblems are solved repeatedly in different parts of the problem.

**Example**: Consider the problem of computing the Fibonacci series. To compute the Fibonacci number at index n, we need to compute the Fibonacci numbers at indices n-1 and n-2. This means that the subproblem of computing the Fibonacci number at index n-1 is used twice in the solution to the larger problem of computing the Fibonacci number at index n.

**Implementation Techniques**

**Memorization (Top-Down Approach):** Memorization is a technique where you store the results of expensive function calls and reuse them when the same inputs occur again. It is a top-down approach where the problem is solved by recursively breaking it down into smaller subproblems.

(Or)

solves and stores subproblem results as they are encountered.

**Tabulation (Bottom-Up Approach):** Tabulation is a technique where you solve the problem by first solving all related subproblems, typically in a bottom-up fashion, and storing their results in a table (array). You then use these stored results to construct the solution to the original problem.

(Or)

solves all subproblems first and then uses these results to solve the original problem.

**Advantages:**

* Avoids recomputing the same subproblems multiple times, leading to significant time savings.
* Ensures that the optimal solution is found by considering all possible combinations.
* Breaks down complex problems into smaller, more manageable subproblems.

**Applications:**

Optimization: Knapsack problem, shortest path problem, maximum subarray problem

Computer Science: Longest common subsequence, edit distance, string matching

Operations Research: Inventory management, scheduling, resource allocation