**Algorithms**

**Linear Search**

It is a straightforward algorithm that checks every element in the list or array until it finds the target value. If the element is found, the algorithm returns its index. If the list doesn’t contain the element, it indicates that as well. This method doesn’t require the list to be sorted and is very intuitive. However, its simplicity comes at the cost of efficiency, especially with large datasets, as it has time complexity of O(n).

**Binary Search**

It is a divide-and-conquer search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array, if they are not equal, the half in which the target is eliminated, and the search continues on the remaining half until the target is found or the search space is empty.

This method requires the array to be sorted beforehand and has a time complexity of O(log n).

**Bubble Sort**

It is one of the simplest sorting algorithms that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name because smaller elements bubble to the top of the list (beginning) with each iteration.

Despite its simplicity, bubble sort is not suitable for large datasets as its average and worst-case complexity are both O(n2)

**Selection Sort**

It is a straightforward sorting algorithm the list into two parts, sorted and unsorted. It repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted segment and moves it to the end of the sorted segment. Though not efficient on large lists compared to more advanced algorithms like Quick Sort, Merge Sort, or Heap Sort, its simplicity makes it easy to understand and implement. The algorithms have a time complexity of O(n2 ).

Here's the step-by-step process:

1. Start with the entire list as the unsorted section.
2. Find the smallest element in the unsorted section of the list.
3. Swap this smallest element with the first element of the unsorted section.
4. Shrink the unsorted section of the list by one from the left, and if any element is left unsorted, repeat from step2.

The name selection sort comes from this process of selecting the next smallest (or largest) element from the unsorted section of the list and then swapping it into place. This selection process is repeated until the entire list is sorted.

The selection sort algorithm has a time complexity of O(n2) in all cases, this makes it inefficient for larger lists compared to more advanced sorting algorithms like merger sort or quicksort.

**Insertion Sort**

It is a simple and efficient comparison-based sorting algorithm. It builds the final sorted array (or list) one item at a time. The algorithm iterates through the input elements and removes one element in each iteration, finds the location it belongs to in the already sorted section of the array, and inserts it there. This process repeats until no unsorted elements remain.

The algorithm is efficient for small data sets and even larger datasets where the data is mostly sorted. It is stable, adaptive, and has an average and worst-case complexity of O(n2).

**Why is more efficient:**

* Fewer Swaps: Insertion sort generally performs fewer swaps compared to bubble sort, especially if the elements are nearly sorted. Each insertion operation can move an element directly to its position, whereas bubble sort swaps adjacent elements, which can be less efficient.
* Adaptive: Insertion sort is adaptive, meaning its efficiency increases if the input is partially or nearly sorted. It can achieve linear time complexity on an almost sorted list.
* Better Best Case: The best-case time complexity of insertion sort is O(n), which is significantly better than the best case of bubble sort, which is also O(n2) in its traditional implementation.

**Pre Order Traversal**

It is one of the primary methods used to explore and interact with tree data structure. It is especially useful in binary trees, where each node has at most two children. It is a method of visiting all the nodes in a tree data structure in a specific order: the current node first (Root), then the left subtree, and finally the right subtree.

In preorder traversal: each node is processed before its child nodes.

The process follows this order:

1. Visits the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

**Post Order Traversal**

It is a technique to visit all the nodes in a tree data structure in a specific sequence: first the left subtree, then the right subtree, and finally the current node.

The process follows this order.

1. Traverse the left subtree in post order.
2. Traverse the right subtree in post order
3. Visit the root node.

**In Order Traversal**

It one of the fundamental tree traversal techniques, particularly suited for binary trees. It ensures that all nodes are visited in their non-decreasing order when applied to binary tree.

It follows a specific sequence to visit all the nodes in binary tree.

1. Traverse the left subtree in order.
2. Visit the root node.
3. Traverse the right subtree in order.

**Level Order Traversal (BFS)**

Also known as Breadth – First Search traversal of a binary tree involves visiting al the nodes of the tree level by level, from up to bottom and from left to right. This traversal technique uses a queue to track nodes and their children as the algorithm progress through the tree.

It is a fundamental technique for exploring and manipulating binary trees, particularly useful in scenarios where problems are naturally structured in layers or levels. By implementing and practicing the algorithm. You’ll develop a strong foundation in tree – based algorithms and their applications in solving complex problems.

**Binary Search Tree (BST)**

It is a fundamental data structure in computer science that organizes elements in a sorted manner for efficient searching, insertion, and deletion operations. It is a binary tree where each node has at most two children, referred to as the left child and the right child, and it satisfies the binary search property.

**Binary Search Property:**

The key feature of a BST is its binary search property, which stipulates that:

* For any node n, all elements in the left subtree of n are less than n.
* All elements in the right subtree of n are greater than n.

This property ensures that the tree remains balanced in terms of its depth, which in turn guarantees operations such as search, insertion and deletions can be performed in logarithmic time complexity (O(log n)) under ideal conditions.

**Binary Search Tree**: is a non-linear data structure in which one node is connected to on number of nodes.

It is a node-based data structure. A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).

Every node in the left subtree must contain a value less than the value of the root node, and the value of the each node in the right subtree must be bigger than the value of the root node.

**Application of BST:**

BSTs are widely used in computer science and applications such as:

* Implementing associative arrays, sets, and multisets.
* Database indices for quick data retrieval.
* Autocomplete features where a prefix search is performed.
* Sorting algorithms.

**Advantages of BST:**

* Efficient operations: Offers O(log n) search, insertion, and deletions operations in the best average cases.
* Sorted Data: Maintains data in a sorted order, facilitating operations like minimum, maximum successor, predecessor, etc., in O(h) time. Where h is height of the tree.

**Disadvantages of BTS:**

* Worst-Case Performance: In the worst case (e.g., inserting sorted data), the BST become unbalanced, resembling a linked list with O(n) time complexity for operations.
* Maintenance: Requires additional logic (e.g., tree balancing techniques) to maintain optimal performance.

**Conclusion:**

Binary Search Trees are a versatile and efficient way to store data for quick search, insertion, and deletion. Understanding how to implement and manipulate BSTs is a valuable skill in computer science, applicable to a wide range of problems and technologies.

**BST Insertion Algorithm:**

1. Start at the root node.
2. If the tree is empty, the new value becomes the root.
   1. Otherwise, compare the new value with the value of the current node. If the new value is less than the current nod’s value. Move to the left child.
   2. If the new value is greater than the current node’s value, move to the right child.
3. Repeat step 2 until you find an empty spot where the new value can be inserted.
4. Insert the new value at the empty spot.

**Algorithm to Search for a Value in s BST:**

1. Start at the root.
2. If the tree is empty or the root node’s value is the target value, return the root node (or None if the tree is empty or the value isn’t found).
3. If the target value is less than the current node’s value, search the left subtree.
4. If the target value is greater than the current node’s value, search the right subtree.
5. If the end of the tree is reached without finding the target value, the target is not in the tree.