**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.

**What is AVL Trees**

They are name after their inventors Adelson – Velsky and Landis, are self – balancing binary search trees. In an AVL tree, the heights of the two child subtrees of any node differ by no more than 1. If at any time they differ by more than one, rebalancing is done to restore this property.

**Key Concepts:**

* Balance Factor: The difference between the heights of the left and right subtrees. It helps in deciding whether a subtree needs rebalancing.
* Height of a Tree: The height of a node is the number of edges on the longest downward path between that node and a leaf.
* Balance Factor = Difference between Hight (Left Subtree) – Hight (Right Subtree)
* If Abs (BF) > 1 then tree is not balanced, otherwise it is balanced.

**Intro to Rotations:**

The AVL trees are type of self-balanced binary search tree.

The balance of a VAL tree ensures that operations like search, insertion, and deletion can be executed efficiently, ideally in O (log n) time.

This balance is maintained through various types of rotations.

**Understanding Tree Balance:**

In a VAL Tree, the balance of a node is calculated as the height of its left subtree minus the height of its right subtree. This value, known as the balance factor, should be -1, 0, or +1 for each node. If the balance factor of any node is outside this range, the tree must be rebalanced using rotations.

**What are Rotations:**

* To maintain balance after insertion and deletions, AVL trees use rotations.
* Rotations are pivotal tree manipulations that shift nodes and their subtrees around to move higher nodes lower and lower nodes higher, thus restoring the required balance of the tree.
* Concept of rotations as operations that reorder the nodes of the tree to maintain or restore balance.
* There are four types of basic rotations: LL, RR, LR, and RL. Each rotation type addresses a specific kind of imbalance.

**Types of Rotations:**

* **RR** – Right Rotation (Single Right Rotation).
* **LL** – Left Rotation (Single Left Rotation).
* **LR** – Left – Right Rotation (Double Rotations).
* **RL** – Right – Left Rotation (Double Rotations).

1. **RR – Right Rotation (Single Right Rotation):**

Used when the node become too left – heavy (balance factor of +2), particularly when its left child has a balance factor of +1 or 0. Here is how it’s performed:

* The left child node becomes the new root of the subtree.
* The original root node becomes the right child of the new root.
* If the new root already had a right child, it becomes the left child of the new right child (the original root).

RR Rotation, or Right – Right Rotation (Single Right Rotation), is applied when a particular imbalance occurs due to inserting a new node into the right subtree of the right child of unbalanced node.

**When to Use**: Left – Heavy scenario where the balance factor of a node becomes +2, Particularly when its left child has a balance factor of +1 or 0.

After this rotation, the balance of the tree is restored. This operation ensures that the height difference between the left and right subtree of any given node remain at most 1, which is crucial for maintaining the efficiency of operations like search, insertion, and deletion in a binary search tree.

1. **LL – Left Rotation (Single Left Rotation):**

Used when a node becomes too right – heavy (balance factor of -2), particularly when its right child has balance factor of -1 or 0. The steps are mirror image of the right rotation:

* The right child of the node becomes the new root of the subtree.
* The original root node becomes the left child of the new root.
* If the new root already had a left child, it becomes the right child of the new left child (the original root).

LL Rotation, or Left – Left Rotation (Singel Left Rotation), is used to correct a specific imbalance in an AVL tree. this imbalance occurs when a new node is inserted into the left subtree of the left child of an unbalance node.

**When to Use**: Right – heavy scenario where the balance factor of a node becomes -2, particularly when its right child has a balance factor of -1 or 0.

After this rotation, the balance of the tree is restored. This operation ensures that the height difference between the left and right subtrees of any given node remains at most 1, which is crucial for maintaining the efficiency of operations like search, insertion, and deletion in a binary search tree.

1. **LR – Left Right Rotation (Double Rotations):**

These are needed when the tree becomes unbalanced due to a child being unbalanced in the opposite direction to the parent. For example, left-right rotation is used when a node has a balance factor of +2 but its child has a balance factor of -1.

* Step 1: Perform a single left rotation on the left child (the problem child).
* Step 2: Perform a single right rotation on the unbalanced node.

LR Rotation involves a two-step process combining a left rotation followed by a right rotation (Double Rotations Left Then Right).

**When to Use**: Imbalanced scenarios that require a preliminary rotation on the child before the main rotation on the parent. When a node has a balance factor of +2 but its left child has a balance factor of -1.

The LR rotation ensures that the height difference between the left and right subtrees of any given node remains at most 1, thereby maintaining the efficiency of search, insertion, and deletion operations in the binary search tree. it’s essential operation in self-balancing binary search tree implementations like AVL trees and Red – Black trees.

1. **RL – Right Left Rotation (Double Rotations):**

These are needed when the tree becomes unbalanced due to child being unbalanced in the opposite direction to the parent. For example, a right-left rotation is used when a node has a balance factor of -2 but its right child has a balance factor of +1.

* Step 1: Perform a single right rotation of the right child (problem child).
* Step 2: Perform a single left rotation for the parent.

RL Rotation involves a two – step process combining a right rotation followed by a left rotation (Double Rotation Right Then Left).

**When to Use**: Imbalanced scenarios that require a preliminary rotation on the child before the main rotation on the parent, when a node has a balance factor of -2 but its right child has a balance factor of +1.

**Deletion in Binary Search Tree (BST):**

* Leaf Node: Simply remove the leaf node.
* Single Child Node: Remove the node and replace it with its child.
* Two Children Node: Replace the node with its in – order successor (the smallest node it its right subtree) or predecessor (the largest node in its left subtree), then delete the successor or predecessor.

**Challenges of Deletion in AVL Tree:**

Deleting a node in an AVL tree might disrupt the balance of the tree, leading to increased time complexities for operation. Restoring the balance involves rotations.

**Rotation to Restore Balance:**

**Single Rotations:**

* Right – Right (RR): Rotate left at the unbalanced node.
* Left – Left (LL): Rotate right at the unbalanced node.

**Double Rotations:**

* Right – Left (RL): Rotate right at the right child, then rotate left at the unbalanced node.
* Left – Right (LR): Rotate left at the left child, then rotate right at the unbalanced node.

**Step – by – Step Deletion Process in AVL Trees:**

1. Perform standard BST Deletion: Start by removing the node as you would in a regular BST.
2. Check Balance Factors: From the deleted node’s parent upward to the root, check each node’s balance factor.
3. Apply Rotation if Necessary: If a node becomes unbalanced (balance factor of +2 or -2), apply the appropriate rotation to balance the tree.

**Summary:**

Deleting nodes in an AVL involves more than just removing the node, it requires recalculating heights and possibly applying rotations to maintain the tree’s balance.

**Red-Black Tree**

It is a type of self-balancing binary search tree, a data structure used in computer science to organize pieces of comparable data, such as numbers. Each node in a Red-Black tree contains an extra bit for denoting the color of the node, either red or black. This color attribute is essential for balancing the tree during insertion and deletions. This balance ensure that the tree remains efficient for operations like insertion, deletion, and look up, which all have time complexity of O(log n) in the worst case.

**Why Red Black Trees:**

They are crucial because they provide a good balance between the complexity of operations and the performance guarantees. They are used in various real-world applications, including:

* Implementing associative arrays.
* Building memory efficient maps and sets in programming languages like C++, Java, and others.
* Maintaining a stored stream of data.

**Properties of Red Black Tree:**

Every node in a Red Black tree has a color, either red or black. The tree must satisfy these five essential properties.

1. Color Property: Every node is either red or black.
2. Root Property: The root of the tree is always black.
3. All leaves (NIL nodes) are black.
4. Red Property: If a red node has children, then both are black (no two red nodes appear in a sequence).
5. Depth Property: Every path from a node to its descendent NIL nodes has the same number of black nodes.

These properties ensure that the tree remains approximately balanced, with no path more than twice as long as any other, keeping operations efficient.

These properties guaranties that the longest path from the root to the leaf is no more twice as long as the shortest path, ensuring the tree remains approximately balanced. As a result, operations such as insertion, deletion, and look up can be completed in logarithmic time, making Red – Black trees an efficient choice for various application in computer science, including implementing associative arrays, priority queues, and in the construction of many other data structures.

**Insert Operation:**

When inserting a new node into a Red Black tree, the new node is initially colored red. The reason for starting with red is to maintain property 5 (balanced black height), which is easier to manage initially as red. If the insertion of a red node violates the properties of the Red Black tree, adjustments are made.

**Steps Of Insertion:**

1. Insert the new node as in a regular binary tree.
2. Color the new node red.
3. Fix any violations of the Red Black tree properties using rotations and recoloring.

**Step By Step Insertion Process:**

* Step 1: Regular BST insertion.
  + Insert the new node as you would in a BST. New nodes are always inserted as red nodes.
* Step 2: Restoration of Properties.
  + If the new node is the root node, simply recolor it to black.
  + If the new node’s parent is black, do nothing.
  + If the new node’s parent is red, further adjustments are needed to fix potential violations.

**Handling Violations:**

When the parent of the node is red, there are a few scenarios to consider based on the color of the uncle node (the sibling of the parent):

* Case 1: Uncle is red
  + Recolor the parent and uncle black and the grandparent red.
  + Move the current node pointer to the grandparent and repeat the restoration process.
* Case 2: Uncle is black or NULL, which requires rotations.
  + Left-Left (LL) Case: Straight line formation on left side. Right rotate the grandparent.
  + Left – Right (LR) Case: Triangle formation on left side. Right rotate the grandparent.
  + Right – Right (RR) Case: Straight line formation on the left. Left rotate the grandparent.
  + Right – Left (RL) Case: Triangle formation on the right. Right rotate the parent, then left rotate the grandparent.

After each rotation, recolor the involved node appropriately.

**Explanation of Red-Black Tree Insertion Method:**

It inserts a new value into the tree as a standard BST, then call FixInsert to maintain Red-Black Tree properties.

1. FixInsert Method: Adjusts the tree after an insertion to fix potential violations of the properties using recoloring and rotations.
2. Rotate Method: RightRotate and LeftRotate are used to rebalance the tree during fixes.

**Delete Operation:**

**Steps for Deleting a Node:**

1. Find the node to delete:

* Start by locating the node that you want to delete. This is done by comparing the value you want to delete with the values in the tree, similar to searching in a regular search tree.

1. Delete the node:

* The actual deletion depends on the node’s children.
* Case 1: Node has no children (it’s a leaf). Simply remove the node.
* Case 2: Node has one child. Replace the node with its child.
* Case 3: Node has to children. Find the node’s in – order successor (the smallest node in its right subtree), replace the node with its successor, and the delete the successor.

1. Fix the tree:

* After deletion, the tree might violate the Red-Black properties. To fix this, you may need to recolor nodes or perform rotation (left or right) to restore balance.

**Types of Violations Post – Deletion:**

After deletion, there are two primary cases to consider.

* **Deleting a Red Node**: This is relatively simple because it doesn’t disrupt the Black – Hight property. Red nodes don’t contribute the black height, so the tree remains balanced.
* **Deleting a Black Node**: This is more complex. The removal of a black node can cause a violation of the Black-Height property, leading to an imbalance. Specifically, the paths from the root to some leaves might have fewer black nodes than others, resulting in a condition known as Double Black.

The Double Black condition is a concept used to track this imbalance and the extra blackness that must be resolved.

Double black is conceptual tool used to describe a temporary violation that occurs when a black node is deleted, or when a black node is replace by a black child (which could be NIL or null node). The term Double Black isn’t a literal extra black color but rather a way to signify that a particular node or position in the tree is missing one black color compare to the other paths in the tree, thus violating the Red-Black tree’s properties.

**Understanding Double Black:**

* **Black – Height Property**: One of the key properties of a Red-Black tree is that every path from a given node to its descendant NIL nodes must have the same number of black nodes. This called the Black Height.
* **Double Black Representation**: To manage this violation, we conceptually treat the node that replaced the deleted node as a double black. This is not a real color but a representation that this node is over counted as black (because we lost one black node). The goal is to resolve this double black condition so that the tree’s black – height property is restored.

**Fixing Double Black:**

1. Recoloring: Adjusting the colors of the sibling and parent nodes to redistribute the black height property.
2. Rotations: Performing tree rotations to rebalance the tree and eliminate the double black condition.
3. Propagation: Sometimes, resolving a double black condition at one node causes another double black condition at the parent node, requiring the process to be repeated higher up the tree.

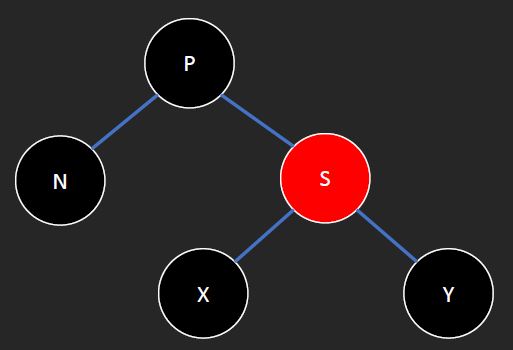
**Example Scenario:**

Suppose you delete a black leaf node, and it is replaced by a NIL node (which is also black). The parent of this NIL node now experiences a discrepancy in black height because one of its subtrees has one fewer black node. We treat the NIL node as Double Black to indicate this imbalance and then proceed to fix it through the aforementioned technique.

**Violation Cases:**

* Case 1: Sibling is Red.
* Case 2: Sibling is Black.
  + Sub Case 2.1: Sibling’s children are both black.
  + Sub Case 2.2: At least one of the sibling’s children is red.
    - Sub Case 2.2.1: Sibling’s far children is red.
    - Sub Case 2.2.2: Sibling’s near children is red.

Case 1: Sibling is Red.

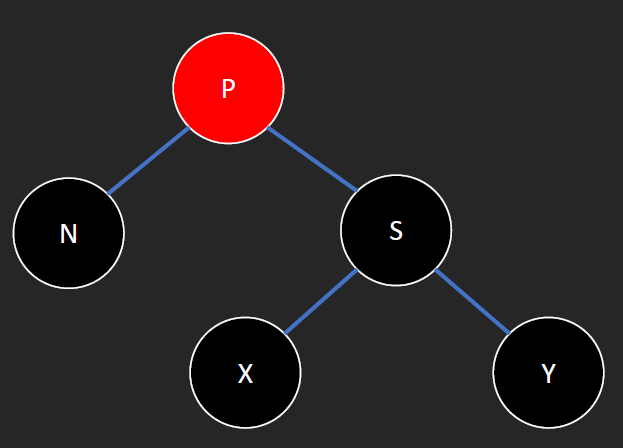
* P is the parent node (black).
* N is the node being deleted or its replacement.
* S is the sibling of N (which is red).
* X and Y are children of S.

**Action:**

1. When you delete the node, it becomes double black.
2. Perform a rotation to move the red sibling to the parent’s position.
3. Swap the colors of the sibling and the parent (color the sibling black and the parent red).
4. The double black situation still exists, but now the sibling of the double black node is black, allowing you to continue with the appropriate steps for case 2.

Case 2: Sibling is Black.

Sub Case 2.1: Sibling’s children are both black.

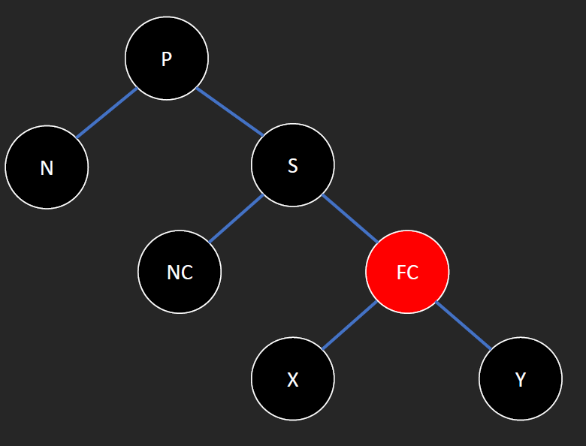
* P is the parent node.
* N is the node being deleted or its replacement.
* S is the sibling of N.
* X and Y are children of S are both black.

Action:

1. When you delete the node, it becomes double black.
2. Color the sibling red.
3. Move the double black up to the parent (effectively reducing the problem to the parent).
   1. If the parent is red, color it black to resolve the double black.
   2. If the parent is black, the double black situation may continue, requiring further handling up the tree.

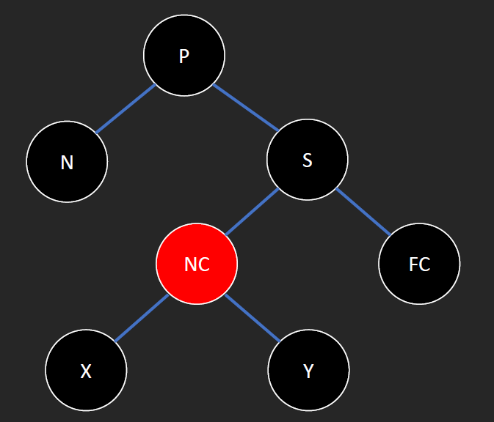
Sub Case 2.2: At least one of the sibling’s children is red .

Sub – Sub Case 2.2.1: Sibling’s far children is red.

* P is the parent node.
* N is the node being deleted or its replacement.
* S is the sibling of N.
* NC is and FC are children of S where NC (near child) and FC (far child).
* X and Y are children of FC.

**Action:**

1. When you delete the node, it becomes double black.
2. Perform a rotation on the parent and sibling (if the sibling is a right child, perform a left rotation, otherwise perform a right rotation).
3. Color the far child black.
4. Color the original sibling with the parent’s original color.
5. Color the parent black.
6. The double black situation may be resolved and may be not, further steps are required to resolve the double black, potentially involving moving the double black up the tree and applying the rules from case 2.1 or case 2.2 again.

Sub – Sub Case 2.2.2: Sibling’s near children is red.

* P is the parent node.
* N is the node being deleted or its replacement.
* S is the sibling of N.
* NC and FC are children of S where NC (near child) and FC (far child).
* X and Y are children of NC.

**Action:**

1. When you delete the node, it becomes double black.
2. Perform rotation on the sibling and its parent (right rotation if the sibling is left child, left rotation if the sibling is a right child).
3. Swap the colors of the sibling and its near child.
4. Now, the sibling is red, turning this into sub – sub case 2.2.1.
5. Follow the actions in the sub-sub case 2.2.1 to resolve the double black.

**Key Points:**

* The primary goal is to restore the Red-Black properties, maintaining the balance of the tree and preserving the black height.
* Actions taken depend on the relative colors of the sibling, the sibling’s children, and the double black node.
* The process may involve multiple iterations, propagating the double black situation upward if necessary.

**Which is better AVL Tree of RedBlack Tree:**

Deciding whether an AVL tree or a RedBlack tree is better depends largely on the specific use case, as each has its strength and trade-offs.

**AVL Tree**

* Balance Factor: They are more rigidly balance than Red-Black trees. They maintain a balance factor of -1, 0, or +1 for every node (the heights of the left and right children of any node differ by at most 1).
* Rotations for Balancing: Due to the strict balancing, AVL trees require more rotations during insertion and deletions to maintain the balance factor.
* Search Operations: The tighter balance guarantees the it has better lookup times, as the tree height is strictly controlled, typically making them faster for lookup-intensive applications.
* Usage: Best suited for lookup-intensive applications where frequent search are more common than insertion and deletions, such as database where read operations dominate.

**Red – Black Trees**

* Balance Factor: They are less strictly balanced. They ensure that no path is more than twice as long as any other, which is less strict compared to AVL trees.
* Rotations for Balancing: They typically require fewer rotations than AVL trees during insertion and deletions because of the more relaxed balance criteria.
* Search operations: Although they may have slightly longer search times compared to AVL trees, the difference is often negligible in practice.
* Insertion and Deletion: The relaxed balancing rules allows for faster insertions and deletions, making them suitable for applications with frequent updates.
* Usage: Commonly used in environments where insertions and deletions are more frequent such as memory managed applications in C++ standard libraries (e.g., std::map, std::set).

**Conclusion**

* **For Lookup Efficiency**: Choose AVL trees if the application involves frequent search and lookups operations and the cost of rebalancing the tree for insertions and deletions is less critical.
* **For Insert/Remove Efficiency**: Choose RedBlack trees if the application requires frequent insertions and deletions and can tolerate slightly longer search time for the benefit of faster updates.

**Difference Between BFS and DFS**

**Breadth – First Search (BFS)**: explores a graph level by level, visiting all neighbors of a node before moving to the next level. It uses a queue to keep track of nodes to visit next. BFS is ideal for finding the shortest path in an unweighted graph because it examines all nodes at the current depth before moving deeper.

**Depth – First Search (DFS)**: On other hand, dives as deep as possible along one bath before backtracking to explore other paths. It uses a stack of recursion. DFS is better suited for problems that require exploring all possible solutions, such as puzzles or finding connected components in a graph.

**When to Use Each:**

* **Use BFS:**
  + When you need to find the shortest path in an unweighted graph.
  + For problems requiring traversal by levels, such as hierarchal structures or games where the shortest move matters.
* **Use DFS:**
  + When you need to explore all paths such as in backtracking problems or detecting cycles in a graph.
  + For memory efficiency when the graph is deep but has fewer branches.

**Which is Faster**

Both of them have the same time complexity of O(V+E), where V is the number of vertices and E is the number of edges.

However

* BFS can be faster for finding the shortest path due to its level-order traversal.
* DFS can feel faster for quickly locating a single solution without considering path optimally.

The choice depends on the problem at hand rather than inherent speed differences.

**Shortest Path**

**Overview**

The shortest path is a fundamental concept in graph theory. It involves finding the shortest path between nodes in a graph.

**What Is the Shortest Path**

The shortest path problem seeks to find the minimum-cost path between two nodes in a graph. Common applications include:

* GPS navigations systems.
* Network routing.
* Game development (pathfinding for characters).

**Types of Shortest Path Algorithms.**

Shortest path algorithms are fundamental in computer science, primarily used in graph theory to find the shortest path between two nodes in a graph. Theses algorithms are widely used in networking, mapping applications, artificial intelligence, and logistics.

**Introduction to Shortest Path Algorithms**

* **Definition**: Shortest path algorithms find the minimum distance or cost from a starting node to a target node in a graph.
* **Graph Representation**: A graph G=(V,E) consist of:
  + **V:** Set of vertices (nodes).
  + **E:** Set of edges, where each edge has a weight representing distance, cost, or time.

**Types of Shortes Path Algorithms**

The choice of algorithm depends on the graph properties and the specific requirements (e.g., single source, all pairs, non-negative weights).

1. **Dijkstra’s Algorithm:**
   * **Purpose**: Finds the shortest path from a single source node to all other nodes in a graph with non-negative weights.
   * **Efficiency**: Time complexity O ((V+E) log V) using a min-heap.
   * **Use Case**: Navigation systems like Google Maps.
2. **Bellman – Ford Algorithm:**
   * **Purpose**: Calculates shortest path from a single source node to all other nodes, allowing negative weights.
   * **Efficiency**: Time complexity O (V.E).
   * **Use Case**: Financial applications involving arbitrage detection.
3. **Floyed – Warshall Algorithm:**
   * **Purpose**: Finds the shortest path between all pairs of nodes.
   * **Efficiency**: Time complexity O (V^3).
   * **Use Case**: Network routing where all pairs shortest paths are required.
4. **A\* Algorithm:**
   * **Purpose**: Optimized for single-pair shortest path with a heuristic to guide the search.
   * **Key Feature**: Combines Dijkstra’s algorithm with a heuristic function h(n) to estimate the cost to the goal.
   * **Efficiency**: Time complexity depends on heuristic quality but can be exponential in the worst case.
   * **Use Case**: Game development for pathfinding (e.g., NPC movement).

**Shortest Path Algorithms Categories:**

Shortest path algorithms can be categorized based on the source from which the shortest paths are calculated.

* **Single-source shortest path**: These algorithms calculate the shortest paths from a single source node to all other nodes in the graph.
  + **Dijkstra’s Algorithm:** Works with non-negative weights.
  + **Bellman-Ford Algorithm:** Handles graphs with negative weights and detects negative weight cycles.
* **Used Cases:**
  + Network routing form a central server to all connected nodes.
  + Navigation systems calculating routes from the current location.
* **Single – Pair Shortest Path:** These algorithms are optimized to find the shortest path between specific pair of nodes (source to destination).
  + **A\* Algorithm:** Uses a heuristic to focus the search and is suitable for graphs with non-negative weights.
* **Use Cases:**
  + Game development for pathfinding (e.g., NPC movement).
  + Logistics applications where only one source-destination path is needed.
* **All – Pairs Shortest Path**: These algorithms compute the shortest paths between all pairs of nodes in the graph.
  + **Floyed – Warshall Algorithm:** Suitable for dense graphs, regardless of weight signs (no negative cycles).
  + **Johnson’s Algorithm:** Efficient for sparce graphs and handles negative weights.

**Comparison of Algorithms**

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Graph Type | Use Case | Time Complexity |
| Dijkstra’s | Non – negative weights | Navigation, routing | O((V+E) log V) |
| Bellman-Ford | Allows negative weights | Finance, cycle detection | O (V.E) |
| Floyed-Warshall | Dense graphs | Network analysis | O (V3) |
| A\* | Singel-pair with heuristic | AI, game development | Varies based on heuristic |
| Johnson’s | Sparse graphs, negative weights | Multi-source shortest paths | O (V2 log V+V.E) |

**Practical Applications**

1. **Navigation systems**: Google Maps and GPS navigation systems.
2. **Network Routing**: Finding optimal data transmission paths.
3. **Game Development**: NPC pathfinding and strategy optimization.
4. **Logistics**: Optimizing delivery routes.

**Conclusion:**

Each shortest path algorithms has specific strength tailored to different graph types and uses cases.

**Dijkstra Algorithm**

**What is Dijkstra Algorithm**

It is an algorithm for finding the shortest path between nodes in a weighted graph, which may represent, for example, road network. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

It is classic and very important algorithm in computer science.

It is a graph-based algorithm used to find the shortest path from a single source node to all other nodes in graph with non-negative weights.

**Key Features:**

* Guarantees the shortest path in graphs with non-negative weights.

**When to use Dijkstra Algorithm:**

* When the graph has non-negative weights.
* To find the shortest path from a single source to multiple destinations.
* In applications like navigation systems, routing protocols, and network optimization.

**Time Complexity**

* **Graph Representation**
  + Adjacency Matrix: O (V2)
  + Adjacency List: O ((V+E) log V).
* V number of vertices, E number of edges.

**Limitations**

* **Non – Negative Weights**: Fails with negative weights edges.
* **Scalability**: Performance depends on graph density and size.

**Applications**

1. **Navigation Systems**: Google Maps calculates the shortest route.
2. **Network Optimization**: Data routing in networks.
3. **Robotics**: Path planning in autonomous vehicles.

**Explanation of implementing Dijkstra’s algorithm using priority queue**

**Purpose of Priority Queue**

The priority queue is used is Dijkstra’s algorithm to efficiently retrieve the vertex with the smallest distance at each step. It helps maintain an ordering of vertices base on their shortest distance (priority).

To improve the time complexity of the algorithm, we used the Priority Queue (Min Heap) to keep track of the vertex with the minimum distance efficiently.

**Key Differences:**

* **Priority Queue:**
  + Implemented using sorted set for maintaining the minimum distance efficiently.
  + Automatically orders vertices based on distance and vertex index.
* Distance Update
  + Updated logic ensures the queue priority is properly managed when distances change.
* Improved time complexity:
  + From O (V2) to O ((V+ E) log V), where V is the number of vertices and E is the number of edges.