INTERNSHIP REPORT

WEEK 2 - Data Structures and Algorithms

SUBMITTED TO

PEOPLE TECH GROUP INC

GOVIND SHARMA

RECRUITMENT LEAD

SUBMITTED BY

SANJEEV REDDY SIRIPINANE

Binary Tree:

A Binary Tree is a hierarchical data structure in which each node can have up to two child nodes, called the left child and right child. Binary trees are widely used in computer science for tasks like efficient data storage, searching, and retrieval. Common operations include insertion, deletion, and traversal (e.g., in-order, pre-order, and post-order traversals).

Properties of a Binary Tree:

1. Root Node: The topmost node in the binary tree, with no parent. A binary tree has only one root.
2. Parent Node: Any node that has one or more child nodes connected to it.
3. Child Node: A node directly connected below a parent node.
4. Siblings: Nodes sharing the same parent.
5. Edges: Links connecting parent nodes to their children.
6. Leaf Nodes: Nodes without any children, marking the endpoints of the tree.
7. Subtree: A portion of the binary tree that considers any node as its root.
8. Node Depth: The distance (in edges) from a node to the root.
9. Node Height: The distance from a node to the deepest node in its subtree.
10. Tree Height: The maximum height of any node in the tree, equal to the height of the root node.
11. Level: Indicates a node's distance from the root in terms of generations. The root is at level 0.
12. Node Degree: The number of children a node has. For binary trees, this is at most 2.
13. NULL Nodes: In a binary tree with N nodes, there are exactly (N + 1) NULL pointers (unused child pointers).

Operation on Binary Tree:

1. Traversal in Binary Tree.
The process of traversing a binary tree entail visiting all of the nodes. Tree traversal algorithms are widely categorized into two categories: DFS and BFS.

Depth-First Search (DFS) algorithms traverse as far down a branch as feasible before returning. It is implemented by recursion. The primary traversal algorithms in DFS for binary trees are:

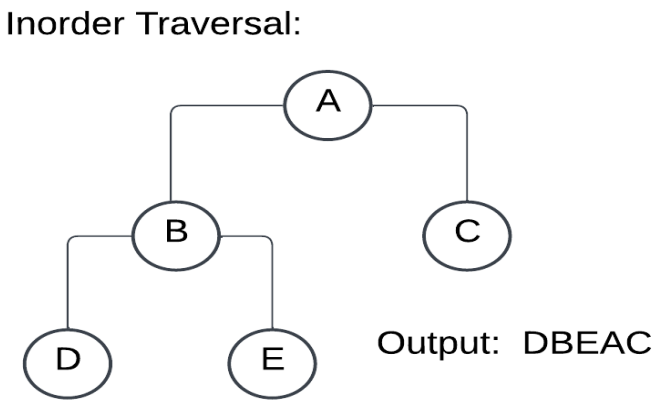
Inorder Traversal of Binary Tree: Inorder traversal is the most common version of DFS traversal of the tree.
As DFS says, we will initially focus on the depth of the chosen node before moving on to the width at that level. As a result, we will begin at the tree's root node and work our way recursively down the left subtree.
When we reach the left-most node using the procedures, we will visit that current node and travel to the left-most node of its right subtree, if one exists.
To finish the inorder traverse, execute the same procedures recursively. The sequence of the stages will be same in a recursive function:

- Navigate to the left subtree.
- Visit node.
- Navigate to the proper subtree.

Pseudocode:

```
public void inorderTraversal(TreeNode root) {
        if (root != null) {
                inorderTraversal(root.left);
                System.out.print(root.data + " ");
                inorderTraversal(root.right);
        }
}
```

Example:



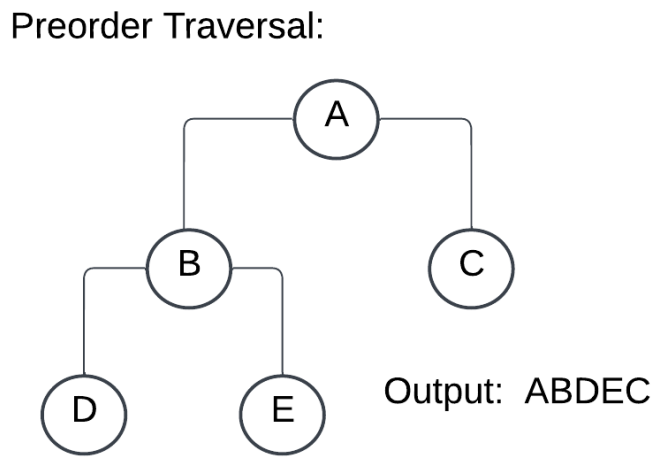
Preorder Traversal of Binary Tree: Preorder traversal is another type of DFS. The atomic actions in a recursive function are identical to those in an inorder traversal, but in a different order.
We start with the current node and work our way down to the left subtree from there. After covering every node in the left subtree, we will go to the right subtree and visit in a similar manner.
The steps will be completed in the following order:

- Visit node.
- Navigate to the left subtree.
- Navigate to the proper subtree.

Pseudocode:

```
public void preorderTraversal(TreeNode root) {
        if (root != null) {
                System.out.print(root.data + " ");
                preorderTraversal(root.left);
                preorderTraversal(root.right);
        }
}
```

Example:



Postorder Traversal of Binary Tree: Post-order Traversal is a form of Depth First Search that visits each node in a certain sequence. You may learn more about Binary Tree

traversals in general here.
Post-order Traversal works by doing a recursive Post-order Traversal of the left and right subtrees, followed by a visit to the root node. It is used to delete trees, post-fix notation for expression trees, and so on.

What distinguishes this traversal as "post" is that visiting a node occurs "after" the left and right child nodes are called recursively.

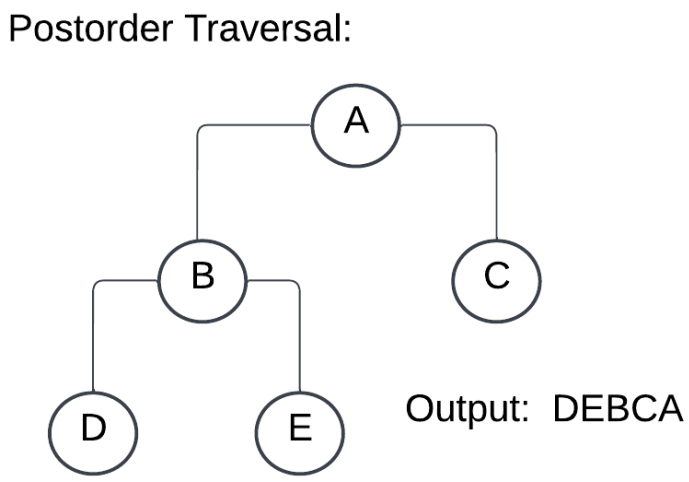
The steps will be completed in the following order:

- Navigate to the left subtree.
- Navigate to the right subtree
- Visit node.

Pseudocode:

```
public void postorderTraversal(TreeNode root) {
        if (root != null) {
                postorderTraversal(root.left);
                postorderTraversal(root.right);
                System.out.print(root.data + " ");
        }
}
```

Example:

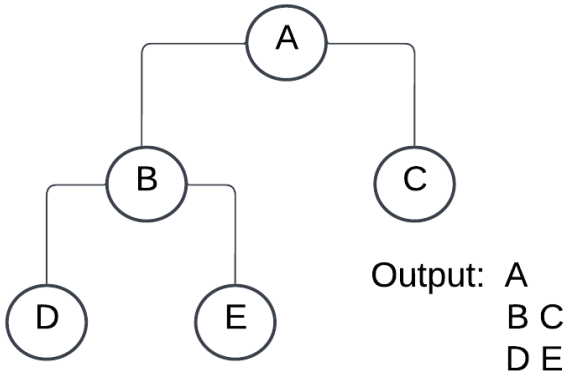


Level order traversal in Binary Tree: This is a method of visiting all the nodes of a binary tree level by level, starting from the root. It uses breadth-first search (BFS) to ensure nodes at the same depth are visited before moving to the next level. This traversal starts at the root, processes all nodes at depth 0, then moves to depth 1, visiting all nodes from left to right, and continues this pattern for subsequent levels. A queue (FIFO data structure) is typically used to keep track of nodes to be processed. After visiting a node, its left and right children are added to the queue in order, ensuring that nodes are processed in the correct sequence to maintain the left-to-right traversal within each level.

Pseudocode:

```
public void levelorderTraversal(TreeNode root) {
        if (root == null) {
                return;
        }
        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
                TreeNode node = queue.remove();
                System.out.print(node.data + " ");
                if (node.left != null) {
                        queue.add(node.left);
                }
                if (node.right != null) {
                        queue.add(node.right);
                }
        }
}
```

Level order Traversal:

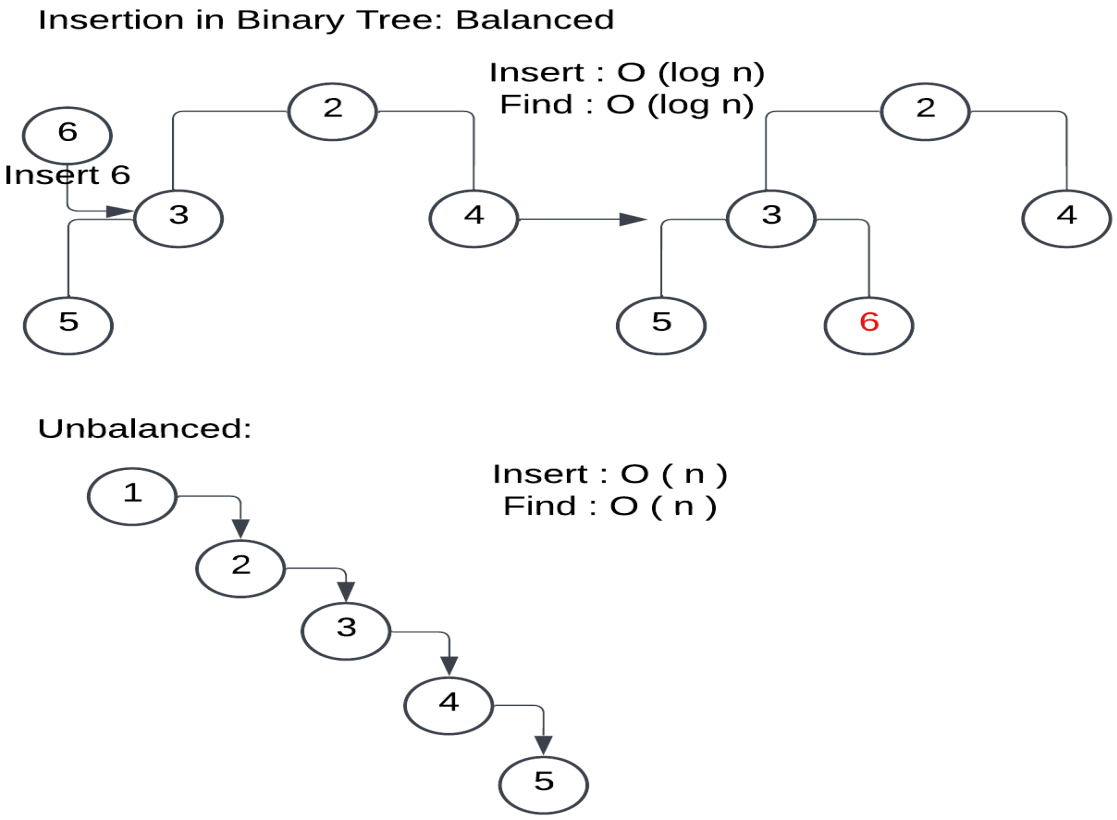


2. **Insertion in Binary Tree:**
Inserting elements involves adding a new node to the binary tree. As we know, there is no such ordering of elements in the binary tree, therefore we do not need to worry about the ordering of nodes. In the event of an empty tree, we would start by creating a root node. Then, the following insertions entail iteratively looking for an empty space at each level of the tree. When an empty left or right child is found, a new node is added there. By convention, insertion always begins with the left child node.

Pseudocode:

Insert (TREE, ITEM)

- Step 1: IF TREE = NULL
  Allocate memory for TREE
  SET TREE -> DATA = ITEM
  SET TREE -> LEFT = TREE -> RIGHT = NULL
  ELSE
  IF ITEM < TREE -> DATA
  Insert (TREE -> LEFT, ITEM)
  ELSE
  Insert(TREE -> RIGHT, ITEM)
  [END OF IF]
  [END OF IF]
- Step 2: END



3. Deletion in Binary Search Tree:

Delete a node from a binary tree involves eliminating a specific node while maintaining the tree's structure. First, we must locate the node that we wish to remove by traversing the tree using any traversal technique. Then, change the node's value with the value of the tree's final node (determined by traveling to the rightmost leaf), and remove that node. This manner, the tree structure will be unaffected. Remember to check for specific instances, such as attempting to remove from an empty tree, to avoid problems.

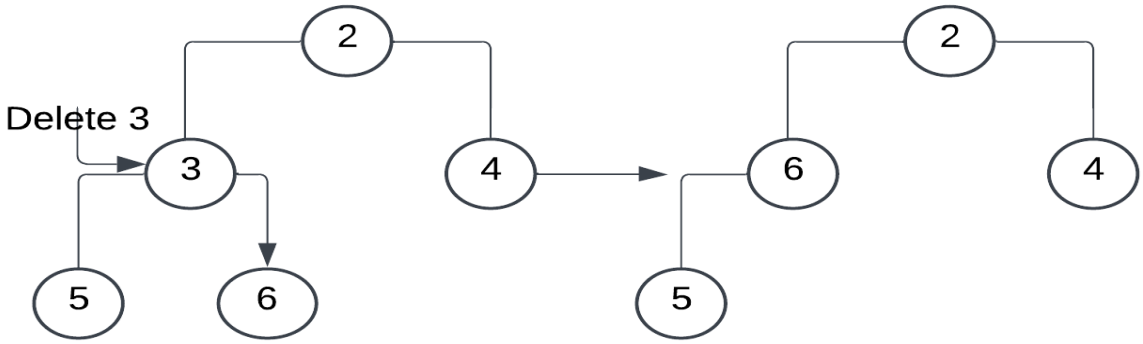
Pseudocode:

- **Step 1:** IF TREE = NULL
  Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
  Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA

```
 Delete(TREE -> LEFT, TEMP -> DATA)
ELSE
 SET TEMP = TREE
 IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
 SET TREE = NULL
ELSE IF TREE -> LEFT != NULL
SET TREE = TREE -> LEFT
ELSE
 SET TREE = TREE -> RIGHT
[END OF IF]
FREE TEMP
[END OF IF]
```

- **Step 2:** END

Deletion in Binary Tree:



Advantages of Binary Trees

1. **Organized Structure**:
   Binary trees arrange data in a clear, hierarchical format, making it easy to manage and understand relationships between elements.
2. **Fast Searching and Sorting**:
   In a Binary Search Tree (BST), finding, adding, or removing elements is quick and efficient when the tree is balanced, often taking only a few steps even for large datasets.
3. **Keeps Things Balanced**:
   Special types of binary trees, like AVL and Red-Black trees, automatically stay balanced. This balance ensures operations like searching and inserting remain fast.

4. **Versatile and Flexible**:
   Binary trees can be adapted into other useful structures like heaps for priority management or Binary Search Trees for ordered data.

5. **Great for Recursion**:
   Since binary trees have a natural hierarchy, they are an excellent fit for recursive algorithms, making certain problems easier to solve.

6. **Handles Large Data Efficiently**:
   Binary trees work well with large datasets, especially when the data changes frequently, ensuring operations remain efficient.

Disadvantages of Binary Trees

1. **Problems with Unbalanced Trees**:
   If a binary tree becomes skewed (like a linked list), the performance of operations like searching or inserting slows down significantly, taking as long as O(n)O(n)O(n).

2. **Extra Memory Usage**:
   Each node in a binary tree requires additional memory to store pointers for its left and right children, which can add up in large trees.

3. **Complex Maintenance**:
   Maintaining balance in advanced binary trees (like AVL or Red-Black trees) can be tricky, as it requires extra operations like rotations.

4. **Limited Node Connections**:
   Each node in a binary tree can only have two children, which may not suit applications where nodes need more connections, like in multi-way trees.

Binary Search Tree:

A Binary Search Tree (BST) is a data structure used in computer science to organize and store data in a sorted order. Each node in a Binary Search Tree has just two children: a left child and a right child, with the left child storing values less than the parent node and the right child carrying values larger than the parent node. This hierarchical form enables efficient search, insertion, and deletion operations on the data contained in the tree.

Binary Search Tree Properties:

- A node's left subtree only contains nodes with keys lower than its own.
- The right subtree of a node only contains nodes with keys larger than the node's.

- The left and right subtrees must both be binary search trees.

- There must be no duplicate nodes (BST may have multiple values with separate handling methods).

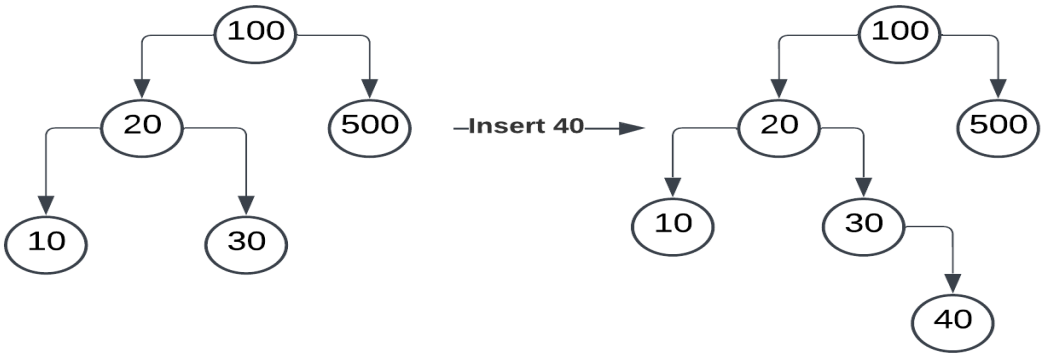
Applications of BST:
A BST allows operations such as search, insert, delete, maximum, minimum, floor, ceiling, larger, smaller, and so on in O(h) time, where the BST's height. Self-balancing BSTs (such as AVL and Red Black Trees) are commonly employed to maintain a low height. These Self-Balancing BSTs maintain a height of O (log n). As a result, all the operations listed above become O (log n). In addition, BST provides for sorted order data traversal in O(n) time.

- A Self-Balancing Binary Search Tree is used to keep a sorted stream of data. For example, assume we get online orders and wish to save the live data (in RAM) in sorted order of pricing. For example, we want to know how many things were purchased at a cost less than a certain amount at any time. Or we want to know how many goods were acquired at a greater cost than the amount provided.
- A Self-Balancing Binary Search Tree is used to create a double terminated priority queue. With a Binary Heap, we may create a priority queue using either extractMin() or extractMax(). If we want to support both operations, we utilize a Self-Balancing Binary Search Tree to accomplish so in O(Log n).
- A BST may be used for sorting a huge dataset. Inserting the dataset's elements into a BST and then conducting an in-order traverse will return the items in sorted order. When compared to traditional sorting algorithms, the advantage here is that we can insert and delete items in O(Log n) time.
- Database indexing makes use of BST variations such as B Tree and B+ Tree. TreeMap and TreeSet in Java, and set and map in C++, are internally implemented with self-balancing BSTs, also known as Red-Black Trees.

Insertion in Binary Search Tree: Maintaining the binary search tree's property results in the insertion of a new key at each leaf. We begin looking for a key from the root until we reach a leaf node. Once a leaf node is discovered, the new node is inserted as a child of that node. The following procedures are taken while attempting to put a node into a binary search tree:

- Initialize the current node (say, currNode or node) with the root node.
- Compare the key to the current node.
- Move to the left if the key is less than or equal to the current node value.
- If the key exceeds the current node value, go to the right.
- Repeat steps 2 and 3 until you get to a leaf node.
- Attach the new key as a left or right child, depending on the value of the leaf node.

Insertion in Binary Search Tree:



Time Complexity:

- The worst-case time complexity of insert operations is O(h) where 'h' is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of insertion operation may become O(n).

Searching in Binary Search Tree:

Algorithm to find a key in a given Binary Search Trees:
Assume we wish to search for the number X. We begin at the root. Then:

- We compare the value to be searched to the value of the root.
    - If it is equal, we are finished with the search; if it is less, we know we need to move to the left subtree because in a binary search tree, all items in the left subtree are smaller and all elements in the right subtree are bigger.
- Repeat the preceding method until no more traversal is feasible.
- If the key is discovered during any iteration, return True. Otherwise, it is false.

  Search (ROOT, ITEM)

    - Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
      Return ROOT
      ELSE
      IF ROOT < ROOT -> DATA
      Return search(ROOT -> LEFT, ITEM)
      ELSE
      Return search(ROOT -> RIGHT,ITEM)
      [END OF IF]
      [END OF IF]

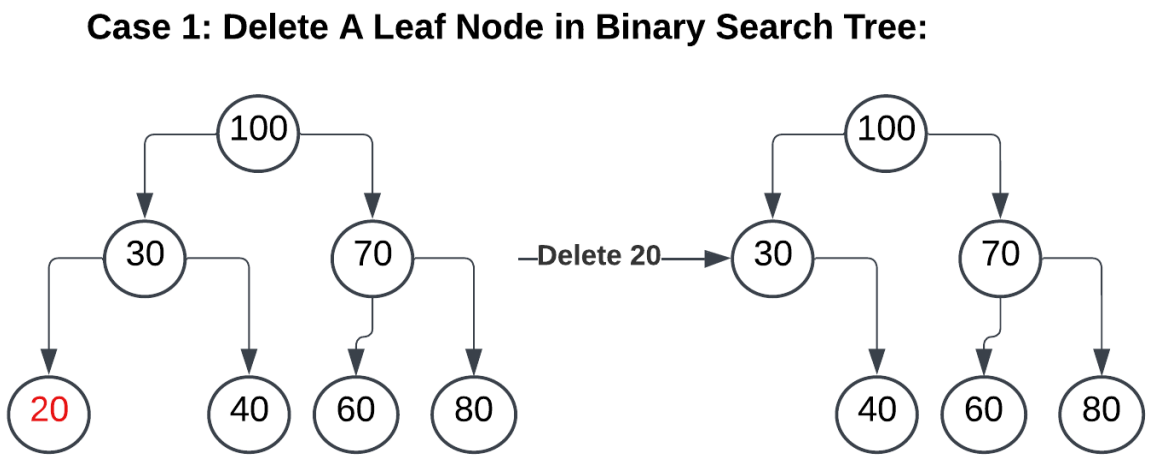
- Step 2: END

Deletion in Binary Search Tree:

The Delete function is used to remove the given node from a binary search tree. However, while deleting a node from a binary search tree, we must ensure that the binary search tree's property is not violated. There are three scenarios for removing a node from a binary search tree.

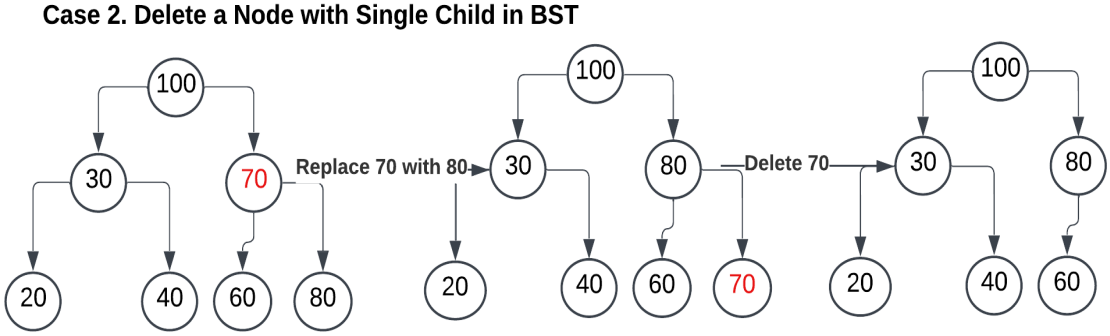
The node to be eliminated is a leaf node:

- It is the simplest scenario; simply replace the leaf node with NULL and release the allotted space.



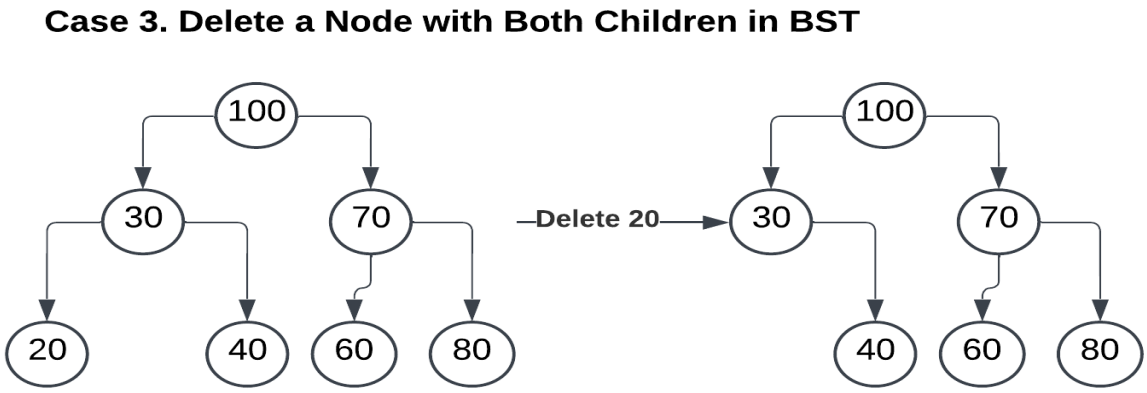
The node that will be eliminated has just one child:

- In this scenario, replace the node with its child, then remove the child node, which now holds the value to be destroyed. Simply replace it with NULL to release the allotted space.



Delete a Node with Both Children in BST:

- Deleting a node with both children is not as straightforward. Here, we must remove the node in such a way that the new tree has the features of a BST.
- The difficulty is to locate the node's in-order successor. Copy the contents of the inorder successor to the node, then remove it.



Algorithm:

- Step 1: IF TREE = NULL
  Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
  Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
  Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
  SET TEMP = TREE
  IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
  SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
  SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
  [END OF IF]
- Step 2: END

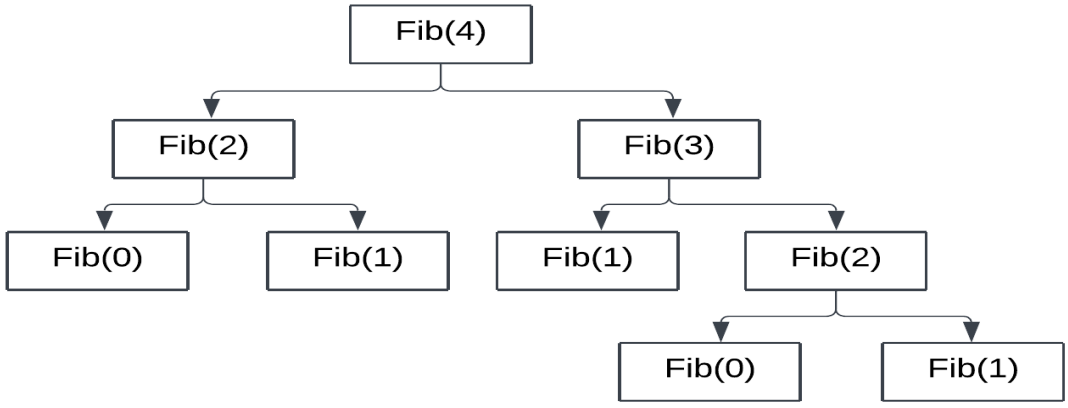
Time Complexity:

	Array(Unsorted)	Linked List	Array(Sorted)	BST
Search(x)	O(n)	O(n)	O(logn)	O(logn)
Insert(x)	O(1)	O(1)	O(n)	O(logn)
Remove(x)	O(n)	O(n)	O(n)	O(logn)

Dynamic Programming:

Dynamic programming is primarily an optimization of ordinary recursion. We may use Dynamic Programming to optimize any recursive solution that involves repeated calls for the same inputs. The goal is to simply save the subproblem results so that we don't have to recompute them later. This simple improvement often decreases the time complexity from exponential to polynomial. Dynamic Programming is commonly used to solve issues such as Fibonacci Numbers, Diff Utility (Longest Common Subsequence), Bellman-Ford Shortest Path, Floyd Warshall, Edit Distance, and Matrix Chain Multiplication.

Dynamic Programming Example:



Working of Dynamic Programming:

Breaking Down the Problem:

- Think of the big problem as a puzzle. DP breaks it into smaller pieces (subproblems) that are easier to solve individually.

Saving Each Solution:

- Once a subproblem is solved, its result is saved (stored in a table or array) so you don't have to solve it again later.

Building the Bigger Picture:

- Use the saved solutions of smaller problems to solve larger problems, step by step, until you solve the entire puzzle.

No Wasted Effort:

- By storing solutions, DP ensures that the same subproblem isn't solved multiple times. This saves a lot of time and avoids unnecessary work.

Usage of Dynamic Programming:

1. Optimization Problems

- Goal: Maximize or minimize value (e.g., cost, profit, or path length).
- Examples:
    - Knapsack Problem: Maximize the value of items that fit within a weight limit.
    - Minimum Path Sum: Find the least costly path in a grid.

2. Sequence Problems

- Goal: Work with sequences, such as finding the longest or shortest subsequence or making transformations.
- Examples:
    - Longest Common Subsequence (LCS): Find the longest sequence common to two strings.
    - Edit Distance (Levenshtein Distance): Calculate the minimum changes needed to transform one string into another.

3. Partitioning Problems

- Goal: Divide a set into subsets based on constraints.
- Examples:
    - Subset Sum Problem: Check if a subset of numbers adds up to a target sum.
    - Palindrome Partitioning: Minimize cuts to partition a string into palindromes.

4. Pathfinding Problems

- Goal: Find the shortest or longest path in a grid, graph, or matrix.
- Examples:
    - Floyd-Warshall Algorithm: Find the shortest paths between all pairs of nodes in a graph.
    - Unique Paths: Count all possible paths in a grid.

5. Game Theory Problems

- Goal: Analyze optimal moves and strategies in games.
- Examples:
    - Nim Game: Determine if a player can force a win given a set of rules.
    - Optimal Strategy for a Game: Decide the best move for a player to maximize their chances of winning.

6. String Problems

- Goal: Solve problems related to string matching, transformations, and patterns.
- Examples:
    - Regular Expression Matching: Determine if a string matches a given pattern.
    - Word Break Problem: Check if a string can be segmented into valid dictionary words.

7. Graph Problems

- Goal: Solve problems involving graph traversal or network optimization.
- Examples:
    - Bellman-Ford Algorithm: Find the shortest path in a weighted graph with negative edges.
    - Travelling Salesman Problem (TSP): Find the shortest route that visits all cities exactly once.

8. Number Problems

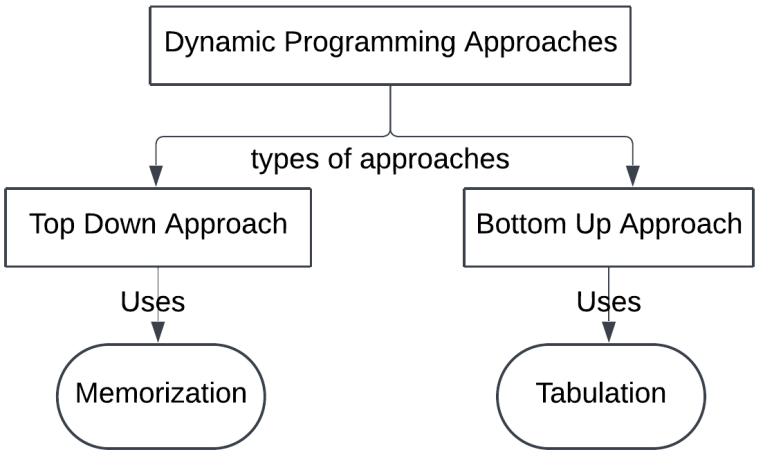
- Goal: Solve problems involving numbers or mathematical properties.
- Examples:
    - Fibonacci Numbers: Compute the nthn^{th}nth Fibonacci number efficiently.

- Coin Change Problem: Find the minimum number of coins needed to make a given amount.

Approach Of Dynamic Programming:

- **Top-Down technique (Memoization**): The top-down technique, also known as memoization, involves starting with the ultimate answer and breaking it down into smaller subproblems. To prevent unnecessary calculations, we keep the outcomes of solved subproblems in a memoization table**.**

- **Bottom-Up technique (Tabulation**): The bottom-up technique, also known as tabulation, involves starting with the simplest subproblems and progressively progressing to the final answer. To prevent unnecessary calculations, we save the outcomes of solved subproblems in a table.



Tabulation vs Memoization:

Tabulation and Memoization are two dynamic programming approaches for optimizing the execution of a function that requires frequent and costly calculations. Although both strategies aim to achieve similar aims, they differ in significant ways. Memorization is a top-down strategy in which we cache the outcomes of function calls and return them when the function is called again with the same inputs. It is employed when the problem may be divided into subproblems, some of which overlap with others. Memorization is often accomplished using recursion and is ideal for issues with a minimal number of inputs. Tabulation is a bottom-up strategy in which the outcomes of subproblems are stored in a table and then used to solve larger subproblems until the full problem is solved. It is utilized when the problem can be expressed as a series of subproblems that do not

overlap. Tabulation is often performed through iteration and is ideal for issues with a high number of inputs.

Memorization:

- Top-down strategy.
- Caches the results of function calls.
- Recursive implementation.
- Suitable for issues with a modest number of inputs.
- Used when there are overlapping subproblems.

Tabulation:

- Bottom-up method.
- Stores the outcomes of subproblems in a table.
- Iterative Implementation
- Suitable for issues with a high number of inputs.
- Used when the subproblems do not overlap.

Problems Related to Dynamic Programming:

Fibonacci Memoization:

- Memoization is a technique to make Fibonacci calculations faster by storing results of previous computations, avoiding redundant calculations.

The Problem with Naive Recursion:

- Fibonacci uses Fib(n) = Fib(n-1) + Fib(n-2).
- Plain recursion recalculates the same values repeatedly, making it slow (e.g., Fib(3), Fib(2)).

How Memoization Fixes It:

- Store previously calculated results in memory (e.g., a list or dictionary).
  Before computing a value, make sure it has not previously been computed.

Example:

```
def fib(n, memo={}):
  if n in memo:
```

```
    return memo[n]
if n <= 1:
    return n
memo[n] = fib(n-1, memo) + fib(n-2, memo)
return memo[n]
```

This ensures each Fibonacci number is calculated only once, making it efficient.

Grid Traveler Memoization:

Determine the number of distinct pathways from the top-left corner to the bottom-right corner in a m x n grid, traveling only right or down.

Recursive Idea: Base Cases.

- There are no pathways when either dimension is zero (for example, m = 0 or n = 0).
- If both dimensions are one (m = 1, n = 1), there is only one route.

Recursive Steps:

- The number of pathways to a certain cell is the total of the paths from:

- The cells immediately above (gridTraveler(m-1, n)) and straight left (gridTraveler(m, n-1)).
- The formula is: gridTraveler(m,n) = gridTraveler(m-1,n) + gridTraveler(m,n-1).

Why Memorization is Needed:

- Plain recursion involves continually calculating the same subproblems. For example:
  To calculate gridTraveler(3, 3), first compute gridTraveler(2, 3) and gridTraveler(3, 2).
  Both will calculate gridTraveler(2, 2).
  This redundancy makes recursion increasingly slower.

Memoization resolves this by:

- Storing the outcomes of previously solved subproblems in memory.
- Using existing results rather than recalculating them.

Approach:

```
def grid_traveler(m, n, memo=None):
```

```
    if memo is None:
        memo = {}
    key = (m, n)
    # Check memo
    if key in memo:
        return memo[key]
    # Base cases
    if m == 0 or n == 0:
        return 0
    if m == 1 and n == 1:
        return 1
    # Recursive calculation with memoization
    memo[key] = grid_traveler(m - 1, n, memo) + grid_traveler(m, n - 1, memo)
    return memo[key]
```

CanSum Memoization:

The canSum problem checks if it is possible to generate a target sum using numbers from a given array. You can use each number as many times as needed. If the sum can be achieved, return True; otherwise, return False.
Memoization is used to store results for previously checked target sums to avoid redundant calculations, improving efficiency.

Example:

- **Problem**: Can you create the sum 7 using the array [2, 3]?
- **Logic**:
    - Start with 7. Try subtracting 2 or 3.
    - Subtract 2: Now check if canSum(5, [2, 3]) is possible.
    - Subtract 3: Now check if canSum(4, [2, 3]) is possible.
    - Repeat until you either reach 0 (success) or can't proceed further

Code:

```
def can_sum(target, numbers, memo=None):
    if memo is None:
        memo = {}
    if target in memo:
        return memo[target]
    if target == 0:
        return True
    if target < 0:
        return False
    for num in numbers:
        if can_sum(target - num, numbers, memo):
            memo[target] = True
            return True
    memo[target] = False
    return False
print(can_sum(7, [2, 3]))  # Output: True
```

howSum Memoization:

The howSum problem asks *how* you can create the target sum using numbers from a given array. Instead of just checking if it's possible, it returns one possible combination of numbers that make up the sum. Memoization stores the results for specific target sums to avoid redundant calculations.

Example:

- **Problem**: How can you create the sum 7 using [2, 3]?
- **Logic**:
    - Start with 7. Subtract 2 or 3 and keep track of the numbers used.
    - Subtract 3: Now check if howSum(4, [2, 3]) can be solved.

- Subtract 2 repeatedly until you reach 0, building the combination [3, 2, 2].

Code:

```
def how_sum(target, numbers, memo=None):
    if memo is None:
        memo = {}
    if target in memo:
        return memo[target]
    if target == 0:
        return []
    if target < 0:
        return None
    for num in numbers:
        remainder = target - num
        result = how_sum(remainder, numbers, memo)
        if result is not None:
            memo[target] = result + [num]
            return memo[target]
    memo[target] = None
    return None
print(how_sum(7, [2, 3]))  # Output: [3, 2, 2]
```

bestSum Memoization:

The bestSum problem asks for the shortest combination of numbers that adds up to the target sum. It's similar to howSum, but instead of returning any combination, it returns the smallest (fewest elements). Memoization avoids recalculating combinations for the same target sum.

Example:

- **Problem**: What is the shortest way to create the sum 7 using [5, 3, 4, 7]?

- **Logic**:
    - Start with 7. Try subtracting 5, 3, 4, and 7.
    - Subtract 7: Reach 0 (combination [7]).
    - Subtract 5: Check bestSum(2, [5, 3, 4, 7]) (combination [5, 3, 4]).
    - Return the shortest combination [7].

Code:

```
def best_sum(target, numbers, memo=None):
    if memo is None:
        memo = {}
    if target in memo:
        return memo[target]
    if target == 0:
        return []
    if target < 0:
        return None
    shortest = None
    for num in numbers:
        remainder = target - num
        result = best_sum(remainder, numbers, memo)
        if result is not None:
            combination = result + [num]
            if shortest is None or len(combination) < len(shortest):
                shortest = combination
    memo[target] = shortest
    return shortest
print(best_sum(7, [5, 3, 4, 7]))  # Output: [7]
```

canConstruct (Memoization):

The canConstruct problem asks whether you can form a target string by using a collection of smaller words. You can use each word as many times as needed. The goal is to determine if it's even possible to create the target using the provided words.
Memoization helps by storing the results of previously solved subproblems (like checking if a smaller part of the target can be constructed), so you don't repeat work and speed up the solution.

How it works:

- You start with the target string.
- For each word in the list, you check if the target starts with that word.
- If it does, remove the word from the target and check if you can construct the rest of the target using the remaining words.
- If you can eventually reach an empty target, it means you can construct it. Otherwise, you return False.

countConstruct (Memoization):

The countConstruct problem counts how many ways you can build a target string from a given list of words. Instead of just checking if it's possible like in canConstruct, this problem tells you how many different combinations of words can form the target string. Memoization again helps by storing intermediate results, so you don't recalculate the same subproblem multiple times.

How it works:

- Start with the target string.
- For each word in the list, check if the target starts with that word.
- If it does, remove the word from the target and count how many ways you can build the rest of the target.
- Add up all the valid ways to form the target. The total is your result**.**

allConstruct (Memoization):
The allConstruct problem finds every single way to build a target string using a list of

smaller words. Unlike canConstruct (which only tells you if it's possible) or countConstruct (which counts the number of ways), allConstruct gives you every possible combination of words that can make up the target string. Memoization is used to speed up the solution by saving the results of subproblems.

How it works:

- Start with the target string.
- For each word in the list, check if the target starts with that word.
- If it does, remove the word from the target and recursively find all possible ways to form the rest of the target.
- Return a list of all possible combinations of words that can construct the target**.**

Fibonacci (Tabulation):

The Fibonacci problem requires you to discover the nth number in the Fibonacci sequence, which is the sum of the two numbers before it. Tabulation is a bottom-up strategy in which you begin with the lowest values and work your way up to the objective, saving each intermediate result in a table (often an array). This eliminates the cost of recursion and guarantees that each value is computed just once, resulting in greater efficiency.

Working:

Instead of utilizing recursion, you begin by populating a table with the basic cases (for Fibonacci, these are 0 and 1). Then, for each successive index, you complete the table by adding the two preceding integers. This procedure will continue till you reach the nth number. The last value in the table is your outcome.

Grid Traveler (Tabulation):

The Grid Traveler issue asks how many ways you can get from the top-left to the bottom-right of a grid while only going right or down. Tabulation is the process of creating a table that reflects all potential pathways at each location on the grid. By iterating over the grid in a methodical manner, you can fill in each cell with the number of pathways to that location from the beginning position without having to recalculate them.

Working:

Begin by making a table with each cell representing a location on the grid. The basic case would be the upper-left corner. The number of ways to reach each consecutive cell is then calculated as the sum of the ways to reach the cell directly above and the cell directly to

the left. The value in the table's bottom-right corner is the total number of ways to get from start to end.

Tabulation Recipe (General Tabulation Process):

Tabulation is a common approach for solving dynamic programming issues that involves creating a table (often an array or matrix) containing the solutions to subproblems. The aim is to solve smaller, easier subproblems and retain the answers so that you may apply them to larger problems. This eliminates the inefficiencies of recalculating findings and assures that each subproblem is addressed once.

Working:

- Identify the base case(s) and initialize the table.
- Set up a loop that fills the table in a systematic way, usually starting from the smallest subproblems and working up to the full problem.
- The table will contain the solution to all subproblems, with the final value being the solution to the original problem.
- The key is that you only compute each subproblem once, saving time and avoiding redundant work.

canSum (Tabulation):

The canSum issue questions if you can use a list of integers to add up to a specific amount. Instead of recursion, we use the tabulation strategy, which involves creating a table (array) with each index representing whether a sum is attainable. Starting with 0 and proceeding higher, we examine each number in the list to see if it may contribute to the intended amount.

Working:

- Initialize a table with a size equal to the target value plus one.
- Set the first index (representing 0) to True because a sum of 0 is always possible (by selecting no numbers).
- Iterate through each number in the list, and for each number, check if the target sum can be formed by adding that number to previously achievable sums.
- If the target sum is achievable at any point, the table will indicate True for that target value.

howSum (Tabulation):

The howSum issue asks not only if a goal sum is possible, but also how to achieve it, which means determining the combination of integers that add up to the objective. The tabulation strategy involves filling a table with each index containing the combination of integers that can make up that specific amount.

Working:

- Initialize a table where each index represents a sum from 0 to the target.
- The first index (0) is set to an empty list because there's no number needed to sum to 0.
- For each number in the list, if a sum can be made, store the combination of numbers that result in that sum.
- At the end, the target sum's index in the table will show the list of numbers that make up the target sum.

bestSum (Tabulation):

The bestSum issue asks for the shortest number combination that adds up to a specific value. Unlike howSum, which only discovers one valid combination, bestSum searches for the combination with the fewest numbers. The tabulation technique populates a table with the best (smallest) combination of values that make up each sum from 0 to the objective.

Working:

- Initialize a table where each index stores the best (smallest) combination of numbers that sum to that index.
- For each number in the list, if adding the number to a sum result in a better (shorter) combination, update the table for that sum.
- The final value in the table for the target will give the shortest combination of numbers that form the target sum.

canConstruct (Tabulation):

Imagine you have a puzzle where you need to use a collection of small word pieces to create a target word. The canConstruct problem asks whether it's even possible to form the target by concatenating these word pieces. Using tabulation, we build a table where each entry represents whether a portion of the target can be formed. You start with the empty target (which is always achievable) and gradually fill out the table to see if you can form the rest of the target using the given words.

Working:

- Think of the table as a checklist for each part of the target.

- Start by marking that you can form an empty target (because nothing needs to be done).
- Then, check for each word in the list: does it match part of the target starting at that position? If it does, mark the next position as achievable.
- At the end, check if you can form the whole target. If the last entry in the table is True, it means you can form the target.