Lesson:

BST One Shot

Pre-Requisites

- Binary Tree
- Traversal on Trees
- Recursion

List of Concepts Involved

- Why Binary Search Tree?
- What is a Binary Search Tree?
- Advantages
- Disadvantages
- Applications
- Insertion
- Traversal - Inorder, Preorder, Postorder
- Searching
- Deletion
- Practice problems on BST

Why Binary Search Tree?

Say, for a college trip, we have created a booking system for booking seats in a bus of capacity 30. Now, our booking system does two functions:

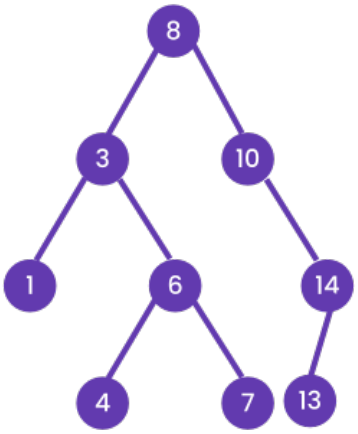
1. Book the seat number entered by the user.
2. Search if the given seat is already booked or not.

We have seat numbers from 1 to 30. And the stream of integers gets added up and the search operation for seat number can be performed anytime.
Currently following seats are booked:
Seats Booked: 3, 15, 8, 1, 29, 14
A user wishes to check if seat 10 is booked or not.
So, what is the most efficient way to search? Binary Search?
But, in that case we need to sort the integers.
Seats Booked (Sorted): 1, 3, 8, 14, 15, 29
Let's say we sorted them, then if 10 is added we need to sort it again before another search.
Adding new booked seat: 1, 3, 8, 14, 15, 29, 10
If we do this way, then let's checkout the time complexity of our system's algorithm.
Binary Seach takes O(log2N) for search operation. (N: No. of seats booked)
Everytime a new seat is booked we need to sort the seats for binary search.
So, O(N log 2N) is the cost of booking new seat. And if there are say M queries, then the time complexity becomes O(M*N*log2N) which is clearly not efficient.

Thus, whenever our data set is constantly updating i.e. dynamic and we need to perform different operations on it like search, add, update, all at once, then we use Binary Search Tree.

What is Binary Search Tree?

A binary search tree (BST) is a type of tree data structure where each node has at most two child nodes, referred to as the left child and the right child. The values stored in the left subtree of a node are less than or equal to the value of the node, while the values stored in the right subtree are greater than the value of the node. This property is true for the root node of every subtree in the binary search tree. This property makes searching and insertion of data efficient, with an average time complexity of O(log n), where n is the number of nodes in the tree.



In this example, the root node has a value of 8. The left subtree contains nodes with values 3, 1, 6, 4, and 7, while the right subtree contains nodes with values 10, 14, and 13. The values in the left subtree are all less than the value of the root node, while the values in the right subtree are all greater than the value of the root node. The same pattern can be observed for every node in respect to their left and right subtrees. This satisfies the property of a binary search tree.

Advantages

Some advantages of using a binary search tree (BST) are:

- **Efficient searching:** The structure of a BST allows for efficient searching of elements. The time complexity of searching for an element in a BST is O(log n) on average, where n is the number of elements in the tree.
- **Sorted ordering:** BSTs maintain a sorted ordering of elements, with smaller elements to the left of a node and larger elements to the right. This makes it easy to perform operations such as finding the minimum or maximum element in the tree.
- **Easy insertion and deletion:** Inserting and deleting elements in a BST is relatively easy and efficient, with a time complexity of O(log n) on average. This makes BSTs a good choice for applications where elements may need to be added or removed frequently.
- **Space efficiency:** BSTs can be implemented with a relatively small amount of memory, as each node only needs to store two-pointers (to its left and right children).
- **Versatility:** BSTs can be used to implement a variety of other data structures, such as sets, maps, and priority queues. They can also be used for a variety of applications, such as searching, sorting, and indexing.

Disadvantages

some disadvantages of using a binary search tree (BST):

- **Unbalanced trees:** If the BST is not balanced, it can lead to worst-case time complexity of O(n) for certain operations, where n is the number of elements in the tree. This can happen if the tree is heavily skewed to one side due to the order in which elements were inserted or deleted.
- **Memory requirements:** Although BSTs are relatively space-efficient, they can require a lot of memory for large trees. This is because each node in the tree requires two pointers (to its left and right children).
- **Lack of support for range queries:** Although BSTs allow efficient searching for a specific element, they do not provide built-in support for range queries (such as finding all elements between two values). This can make certain types of queries more difficult or inefficient to perform.
- **Slow performance for certain operations:** Although searching, insertion, and deletion are efficient on average, certain operations such as finding the kth smallest element in the tree or balancing the tree can have worst-case time complexity of O(n).
- **Complexity of implementation:** Implementing a binary search tree correctly can be complex, particularly for operations such as balancing the tree or handling cases where nodes have multiple children.

Applications

Binary search trees (BSTs) have a wide range of applications. Here are some common applications of BSTs:

- **Searching and indexing**: BSTs are commonly used for searching and indexing applications. Because BSTs maintain a sorted ordering of elements, they allow for efficient searching and retrieval of elements, with a time complexity of O(log n) on average.
- **Sets and maps:** BSTs can be used to implement sets and maps, where the elements in the set or map are stored in sorted order. This makes it easy to perform operations such as finding the minimum or maximum element in the set or map.
- **Priority queues:** BSTs can be used to implement priority queues, where the highest-priority element is always at the root of the tree. This makes it easy to perform operations such as adding and removing elements in priority order.

Real-life applications of BST

- **Phone book:** In a phone book, the names are usually arranged in alphabetical order. BSTs can be used to store the names and phone numbers, with each node representing a person and its left and right subtrees representing people whose names come before and after in alphabetical order. This allows for efficient searching and retrieval of phone numbers.
- **Dictionary:** A dictionary is a collection of words and their definitions arranged in alphabetical order. BSTs can be used to store the words and definitions, with each node representing a word and its left and right subtrees representing words that come before and after in alphabetical order. This allows for efficient searching and retrieval of words and definitions.
- **Code autocomplete:** Code editors often have an autocomplete feature that suggests code snippets based on the user's input. BSTs can be used to store the code snippets, with each node representing a snippet and its left and right subtrees representing snippets that come before and after in alphabetical order. This allows for efficient searching and retrieval of code snippets.
- **Stock market analysis:** In stock market analysis, it is often useful to track the performance of individual stocks over time. BSTs can be used to store the stock prices, with each node representing a price and its left

and right subtrees representing prices that come before and after in time order. This allows for efficient searching and retrieval of historical stock prices.

- **Genome sequencing:** In genome sequencing, it is often useful to search for specific patterns of DNA. BSTs can be used to store the DNA sequences, with each node representing a sequence and its left and right subtrees representing sequences that come before and after in alphabetical order. This allows for efficient searching and retrieval of DNA sequences.

Insertion

Code:
https://pastebin.com/9gqKe30d

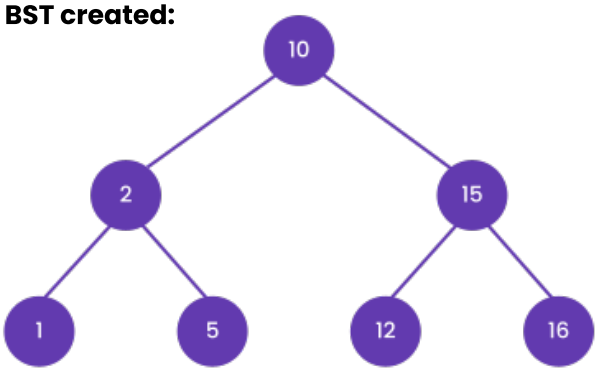
Explanation:

- The program starts by defining a Node class and a BST class.
- The Node class represents a node in the BST, and the BST class represents the BST itself.
- The insert method in the BST class takes a value as input and inserts a new node with that value in the BST.
- If the BST is empty, the new node becomes the root.
- Otherwise, the method iterates through the BST starting from the root, following the left or right branch depending on whether the value is smaller or larger than the value of the current node respectively until it finds an empty spot to insert the new node.

Output:

```
Enter a value to insert: 10
Preorder of BST: 10

Do you want to insert another value? (y/n): y
Enter a value to insert: 2
Preorder of BST: 10 2

Do you want to insert another value? (y/n): y
Enter a value to insert: 15
Preorder of BST: 10 2 15

Do you want to insert another value? (y/n): y
Enter a value to insert: 5
Preorder of BST: 10 2 5 15

Do you want to insert another value? (y/n): y
Enter a value to insert: 1
Preorder of BST: 10 2 1 5 15

Do you want to insert another value? (y/n): y
Enter a value to insert: 16
Preorder of BST: 10 2 1 5 15 16

Do you want to insert another value? (y/n): y
Enter a value to insert: 12
Preorder of BST: 10 2 1 5 15 12 16

Do you want to insert another value? (y/n): n
```

BST created:



Time Complexity

Inserting a node in a BST takes O(log n) time on average, where n is the number of nodes in the tree. This is because in each iteration of the while loop in the insert method, the size of the subtree being searched is halved, leading to a logarithmic time complexity.

If the BST is unbalanced and degenerates into a linked list, the time complexity of inserting a node becomes O(n), where n is the number of nodes in the tree.

Space Complexity

The space complexity of the program is O(n), where n is the number of nodes in the BST. This is because the program creates a new node object for each value inserted into the BST, and these node objects are stored in memory until the program terminates.

Traversal

Code: https://pastebin.com/6k1fg68t

Explanation:

This program creates a BST with some nodes and performs inorder, preorder, and postorder traversals on it. The createNode function creates a new node with the given value. The insert function inserts a new node into the BST in the correct position based on the node's value. The inOrderTraversal, preOrderTraversal, and postOrderTraversal functions perform the corresponding tree traversals.

Inorder Traversal:

1. Traverse the left subtree by calling inOrderTraversal on the left child node.
2. Visit the node and output its value.
3. Traverse the right subtree by calling inOrderTraversal on the right child node.

Preorder Traversal:

1. Visit the node and output its value.
2. Traverse the left subtree by calling preOrderTraversal on the left child node.
3. Traverse the right subtree by calling preOrderTraversal on the right child node.

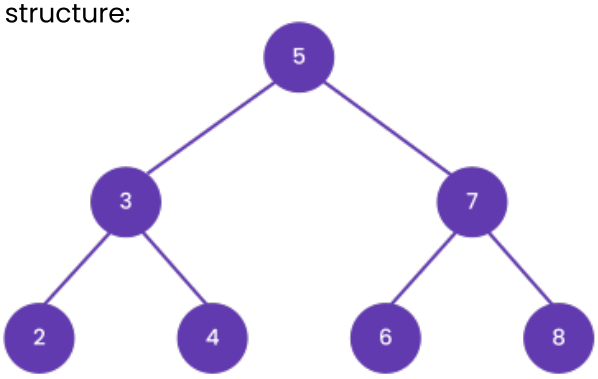
Postorder Traversal:

1. Traverse the left subtree by calling postOrderTraversal on the left child node.
2. Traverse the right subtree by calling postOrderTraversal on the right child node.
3. Visit the node and output its value.

Output:

```
Inorder traversal of the binary search tree: 2 3 4 5 6 7 8
Preorder traversal of the binary search tree: 5 3 2 4 7 6 8
Postorder traversal of the binary search tree: 2 4 3 6 8 7 5
```

This output shows the inorder, preorder, and postorder traversal of a binary search tree with the following structure:



Time Complexity

The time complexity of the traversal functions inOrderTraversal, preOrderTraversal, and postOrderTraversal is O(n), as each node is visited once.

Space Complexity

The space complexity of the traversal functions is O(h), where h represents the height of the tree, as it requires space for the recursive function calls on the call stack. In the worst case, when the tree is skewed, the space complexity of the traversal functions is O(n).

Searching

Explanation

To search in a BST, we follow the following steps:

- Start at the root node of the BST.
- Compare the value you are searching for with the value of the current node.
- If the value is equal to the current node's value, return the node.
- If the value is less than the current node's value, move to the left child node of the current node (if it exists).
- If the value is greater than the current node's value, move to the right child node of the current node (if it exists).
- Repeat steps 2-5 until either the value is found or a null node is reached (indicating that the value is not in the BST).
- If the value is not found in the BST, return null or a message indicating that the value was not found.

That's it! Searching in a BST is a simple process that takes advantage of the tree's structure and the ordering of its nodes to efficiently locate values.

Recursive Code: https://pastebin.com/SNLRz2sF

Recursive Code Explanation:

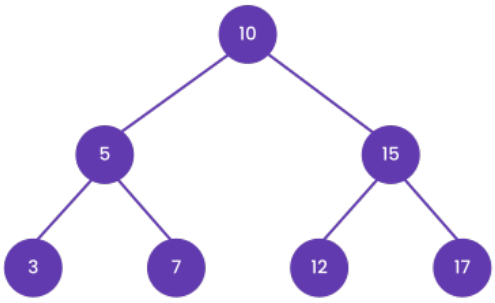
In this program, we first create a BST by getting values from the user and inserting them one by one. Then we ask the user to enter a value and check if it's present in the BST or not using the search function recursively. If the value is present, we print a message saying so, otherwise we print a message saying the value is not present.

Output:

```
Enter values to insert in BST (-1 to stop): 10 15 5 12 7 17 3 -1
Enter a value to search in BST: 12
Value 12 is present in BST.
```

```
Enter values to insert in BST (-1 to stop): 10 15 5 12 7 17 3 -1
Enter a value to search in BST: 99
Value 99 is not present in BST.
```

BST Diagram:



Time Complexity: The time complexity of this program is O(log n) in the average case and O(n) in the worst case i.e. when the tree is skewed, where n is the number of nodes in the BST.

Space Complexity: The space complexity is O(h), where h is the height of the BST and this is because of the call stack created due to recursive function calls.

Iterative Code: https://pastebin.com/d41LUEhw

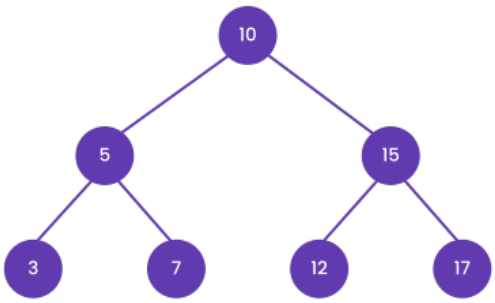
Iterative Code Explanation: In this program, we first create a BST by getting values from the user and inserting them one by one. Then we ask the user to enter a value and check if it's present in the BST or not using the search function which is implemented iteratively. If the value is present, we print a message saying so, otherwise we print a message saying the value is not present.

Output:

```
Enter values to insert in BST (-1 to stop): 10 15 5 12 7 17 3 -1
Enter a value to search in BST: 7
Value 7 is present in BST.
```

```
Enter values to insert in BST (-1 to stop): 10 5 15 7 12 17 3 -1
Enter a value to search in BST: 1000
Value 1000 is not present in BST.
```

BST Diagram:



Time Complexity: The time complexity of this program is O(log n) in the average case and O(n) in the worst case i.e. when the tree is skewed, where n is the number of nodes in the BST.

Space Complexity: The space required by the iterative implementation does not depend on the height of the tree, but only on the constant number of variables used in the implementation. Therefore, the space complexity of the iterative implementation of searching in a BST is O(1).

Deletion

Approach:
Deletion in a Binary Search Tree (BST) involves removing a node from the tree while maintaining the properties of the BST. Here are the steps for deleting a node from a BST:

- **Find the node to be deleted:** The first step in deleting a node from the BST is to find the node that needs to be deleted. We start at the root node and traverse the tree until we find the node that matches the value we want to delete.
- **Determine the type of node to be deleted:** Once we have found the node to be deleted, we need to determine what type of node it is. There are three types of nodes we need to consider:
  a) **Leaf node:** A node with no children.
  b) **Node with one child:** A node with only one child.
  c) **Node with two children:** A node with two children.
- **Delete the leaf node:** If the node to be deleted is a leaf node, we can simply remove it from the tree.
- **Delete a node with one child:** If the node to be deleted has only one child, we can replace the node with its child. We simply connect the parent of the node to be deleted with its child node.
- **Delete a node with two children:** If the node to be deleted has two children, we need to find the node with the next highest value in the tree. This node is called the successor node. We can replace the node to be deleted with the successor node and then delete the successor node using steps 3 or 4 above.
- **Update the tree:** After deleting the node, we need to update the tree to maintain the BST properties. We need to ensure that all nodes to the left of a node have a value less than the node, and all nodes to the right of a node have a value greater than the node.
- **Repeat if necessary:** If we have deleted a node with children, we need to repeat the process for those children until we have removed all the nodes that need to be deleted.

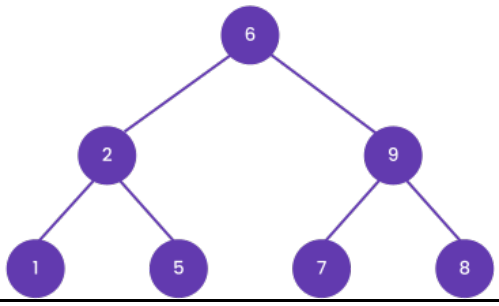
Overall, the deletion process in a BST can be complex, depending on the structure of the tree and the type of node to be deleted. However, by following the steps above, we can ensure that the tree remains a valid BST after the deletion.

Code: https://pastebin.com/Gz0q2xQz

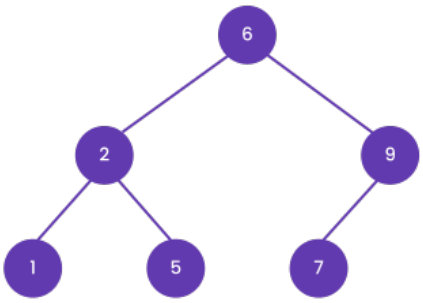
Output:

```
Enter the number of nodes in the BST: 7
Enter the values of the nodes: 6 2 9 8 5 1 7

Preorder traversal of the BST: 6 2 1 5 9 8 7
Enter the value to be deleted: 9
Preorder traversal of the modified BST: 6 2 1 5 8 7
```

BST Diagram:
Before deletion



After deletion



Time Complexity:

- Insertion and deletion operations in a BST have a time complexity of O(log n) in the average case, where n is the number of nodes in the BST.
- However, in the worst case, the time complexity can be O(n) if the BST is degenerate (e.g. all nodes have only one child) and resembles a linked list.

Space Complexity:
The space complexity of the deleteHelper() function depends on the height of the BST. In the worst case, it is O(n) i.e. when the BST is skewed and in the average case, it is O(log n).

MCQ Questions

1. **Consider a binary search tree with n nodes. What is the maximum possible height of the tree?**
   A) O(n)
   B) O(log n)
   C) O(n log n)
   D) O(sqrt(n))

   Answer: A) O(n)
   Explanation: The maximum possible height of a binary search tree occurs when the tree is a linked list, with all nodes connected in a straight line. In this case, the height of the tree would be n. Therefore, the maximum possible height of the tree is O(n).

2. **Consider a balanced binary search tree with n nodes. What is the minimum number of comparisons required to search for a value in the worst-case scenario?**
   A) O(1)
   B) O(log n)
   C) O(n log n)
   D) O(n)

   Answer: B) O(log n)
   Explanation: In the worst-case scenario, the value being searched for is either the smallest or the largest value in the tree, and it is located at the bottom-most level of the tree. In a balanced BST, the height of the tree is O(log n), so the worst-case scenario requires O(log n) comparisons to reach the bottom-most level of the tree. Therefore, the minimum number of comparisons required to search for a value in the worst-case scenario is O(log n).

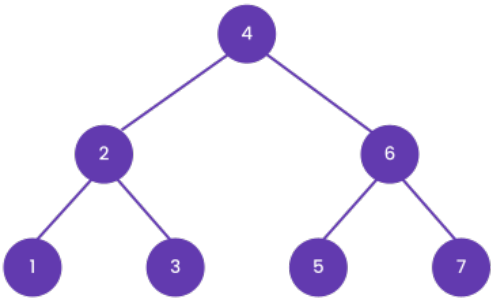
Coding Questions

Q1. Sorted Array to Balanced BST
Given a sorted array. Create a Balanced Binary Search Tree out of it.

A Balanced BST is height-balanced i.e. the difference between the height of the left subtree and right subtree is not more than 1.
You need to print the preorder traversal of the BST created.

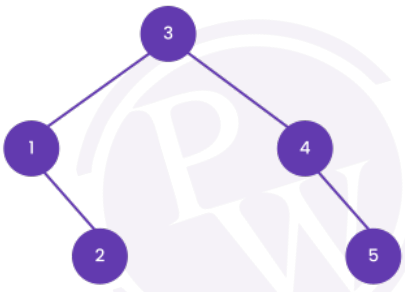
Example 1:

Input:
n = 7
Elements = [1, 2, 3, 4, 5, 6, 7]
Output: 4 2 1 3 6 5 7
Explanation: The balanced BST looks like this:



Example 2:

Input:
n = 5
Elements = [1, 2, 3, 4, 5]
Output: 3 1 2 4 5
Explanation: The balanced BST looks like this:



Solution Code:
https://pastebin.com/JQArBPfp

Code Explanation:

- The idea is to find the middle element of the array and make it the root of the tree.
- Then, recursively do the same for the left half and right half.
  - Find the middle of the left half and make it the left child of the root.
  - Find the middle of the right half and make it the right child of the root.
- Print the Preorder.

Time complexity: O(n), because the size of the array is n, and we are creating BST with n nodes. So, we need to traverse at least once.

Space complexity: O(h) ~ O(logn), where h is the height of the BST. Since, we are using recursion, so call stack will be created every time we make a call to the function.

Output:

```
Enter the size of the array: 7
Enter the elements of the array: 1 2 3 4 5 6 7

Preorder traversal of constructed BST: 4 2 1 3 6 5 7
```

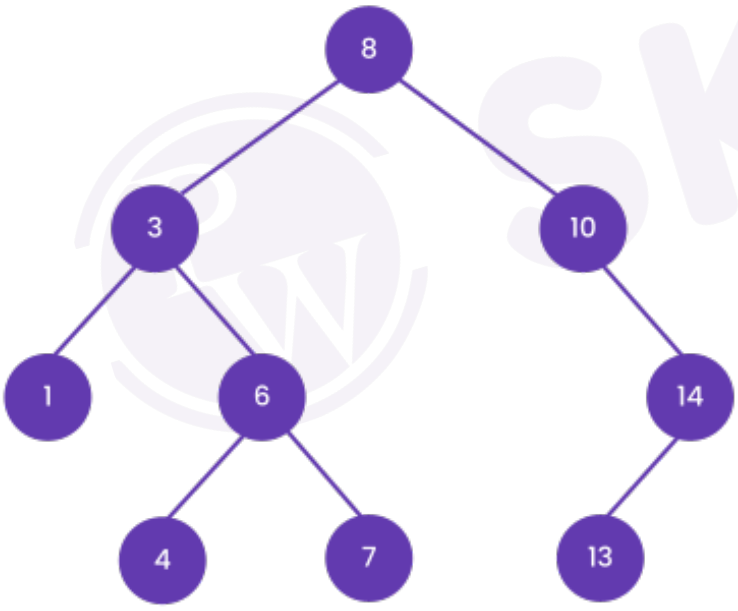
```
Enter the size of the array: 5
Enter the elements of the array: 1 2 3 4 5

Preorder traversal of constructed BST: 3 1 2 4 5
```

Q2. Lowest Common Ancestor of BST

Given a Binary Search Tree (BST) and two values. You need to find the LCA i.e. Lowest common ancestor of the two nodes provided both the nodes exist in the BST. [easy]

Example:

Consider the following BST:



Input-1:

n = 9
values = [8, 3, 1, 6, 4, 7, 10, 14,13]
node-1 = 3
node-2 = 13

Output-1:

Lowest Common Ancestor = 8

Input-2:

n = 9
values = [8, 3, 1, 6, 4, 7, 10, 14,13]
node-1 = 14
node-2 = 13

Output-2:

Lowest Common Ancestor = 14

Recursive Approach:

- We are going to create a recursive function that takes a node and the two values n1 and n2.
- If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for the right subtree.
- If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for the left subtree.
- If both the above cases are false then return the current node as LCA.

Solution Code: https://pastebin.com/57RsEapT (Recursive)

Iterative Approach:

- Start from the root node of the BST.
- While the root node is not null:
  a. If both nodes have values greater than the value of the root node, then move to the right child of the root node.
  b. If both nodes have values less than the value of the root node, then move to the left child of the root node.
  c. If one node has a value greater than the value of the root node and the other node has a value less than the value of the root node, then the root node is the LCA of the two nodes.
  d. If either of the two nodes being searched for is equal to the value of the current root node, then the current root node is the LCA of the two nodes.
- If neither of the above conditions is met, then return the current root node.
- Return the LCA as the output of the function.

Solution Code: https://pastebin.com/qSL5Y68e (Iterative)

Output:

```
Enter the number of nodes: 9
Enter the values of nodes: 8 3 1 6 4 7 10 14 13
Node-1: 3
Node-2: 13
Lowest Common Ancestor: 8
```

```
Enter the number of nodes: 9
Enter the values of nodes: 8 3 1 6 4 7 10 14 13
Node-1: 14
Node-2: 13
Lowest Common Ancestor: 14
```

Time complexity: O(H) where H is the height of the tree. As we are starting to check for LCA from the root till we find the given nodes.

Space complexity: O(H) since we need a recursive stack of size H in case of recursive solution.
However, in the case of an iterative solution, the space complexity is O(1), constant, because no call stack is created.

Q3. Remove all leaf nodes from BST
Input: The input to the program is the number of nodes in the BST, followed by the node values, in the order they are inserted.
Output: The program outputs the preorder traversal of the BST before and after removing the leaf nodes.
Explanation:

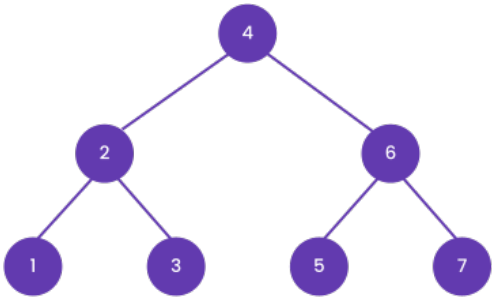
- The program first creates a Binary Search Tree by inserting nodes based on their values.
- It then prints the preorder traversal of the BST before removing the leaf nodes.
- It then removes all the leaf nodes from the BST using a recursive helper function.
- The helper function checks if the current node is null or a leaf node.
- If it is a leaf node, it returns null, otherwise, it recursively removes the leaf nodes from the left and right subtrees.
- Finally, the program prints the preorder traversal of the BST after removing the leaf nodes.

Code: https://pastebin.com/jRGvCmxM

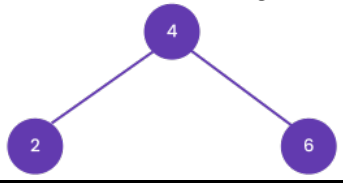
Output:

```
Enter the number of nodes: 7
Enter the node values: 4 2 6 1 3 5 7
Preorder traversal of the BST before removing leaf nodes:
4 2 1 3 6 5 7
Preorder traversal of the BST after removing leaf nodes:
4 2 6
```

BST before removing leaf nodes:



BST after removing leaf nodes:



Time Complexity: The time complexity of the insert() method is O(log n) in the average case and O(n) in the worst case, where n is the number of nodes in the BST. The time complexity of the removeLeafNodes() method is also O(n) as it visits every node once. The time complexity of the preorderTraversal() method is also O(n) as it visits every node once.

Space Complexity: The space complexity of the program is O(n), where n is the number of nodes in the BST. This is because the program creates a BST of size n and also uses a recursive call stack of size n during the execution of the removeLeafNodes() method, a recursive call stack is used. Each recursive call to removeLeafNodes() creates a new stack frame, which adds to the space complexity of the program. The maximum depth of the recursive call stack is equal to the height of the BST. In the worst case, when the BST is skewed, the height of the tree is equal to n, the number of nodes. Therefore, the space complexity of the program is O(n) in the worst case.

Q4. Inorder predecessor and successor for a given key in BST

Input:
The input to the program is the key for which we want to find the inorder predecessor and successor in the BST.

Output:
The output of the program is the inorder predecessor and successor of the given key in the BST.
Explanation:
In a Binary Search Tree (BST), the inorder predecessor of a node is the node with the largest value smaller than the given node, and the inorder successor of a node is the node with the smallest value larger than the given node.

To find the inorder predecessor and successor of a given key in a BST, we can start by traversing the BST from the root node. If the key is found in the BST, we can find its predecessor and successor as follows:

- If the left subtree of the key node is not empty, the predecessor of the key node is the rightmost node in the left subtree (i.e., the node with the largest value smaller than the key).
- If the right subtree of the key node is not empty, the successor of the key node is the leftmost node in the right subtree (i.e., the node with the smallest value larger than the key).

If the key is not found in the BST, we can still find its predecessor and successor as follows:

- If the key is smaller than the root node, we move to the left subtree of the root node and update the successor to be the root node. We repeat this process until we find the last node with a value smaller than the key, which is the inorder predecessor.
- If the key is larger than the root node, we move to the right subtree of the root node and update the predecessor to be the root node. We repeat this process until we find the first node with a value larger than the key, which is the inorder successor.

Overall, to find the inorder predecessor and successor of a given key in a BST, we need to traverse the tree once

Code: https://pastebin.com/EV4xy6AS

Output:

```
Inorder predecessor is 50
Inorder successor is 70
```

Time Complexity:
The time complexity of this program is O(h), where h is the height of the BST. In the worst case, when the BST is skewed (i.e., has only one child for each node), the time complexity is O(n), where n is the number of nodes in the BST.

Space Complexity:
The space complexity of this program is O(1), since it uses only a constant amount of extra space.

Upcoming Class Teasers:

- Generic Trees