Lesson:

Binary Search Trees-1

Pre Requisites:

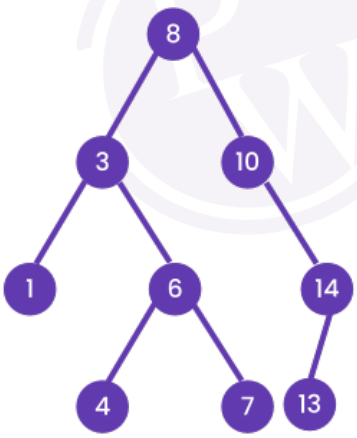
- Binary Trees
- Problems on Binary Trees

List of concepts involved

- What is Binary Search Tree?
- Why Binary Search Tree?
- Advantages
- Disadvantages
- Applications
- Searching
- Insertion
- Traversal - Inorder, Preorder, Postorder
- Deletion

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 each and every node of 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. This satisfies the property of a binary search tree. If you check for all the nodes, and their corresponding left and right subtrees, you will see that this property is satisfied at every node of this tree.

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:
Book the seat number entered by the user.
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.

Advantages

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

1. **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.
2. **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.
3. **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.
4. **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).
5. **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):

1. **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.
2. **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).
3. **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.
4. **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).
5. **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:

1. **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.
2. **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.
3. **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

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.

Searching

Explanation

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

1. Start at the root node of the BST.
2. Compare the value you are searching for with the value of the current node.
3. If the value is equal to the current node's value, return the node.
4. If the value is less than the current node's value, move to the left child node of the current node (if it exists).
5. If the value is greater than the current node's value, move to the right child node of the current node (if it exists).
6. 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).
7. 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/zG0QXbwR

Recursive Code Explanation:

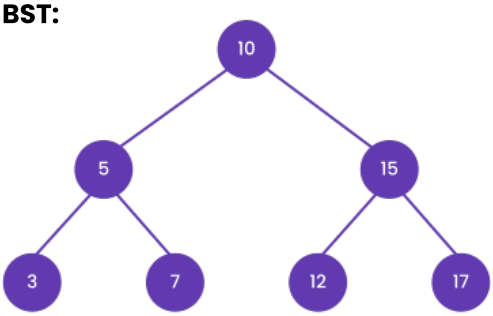
In this program, we first create a BST with some values. Then, we prompt the user to enter a value to search in the BST using the Scanner class. Finally, we check if the value is present in the BST using the search() method of the BST class by recursively calling search() on the appropriate nodes, and print the appropriate message.

Output:

```
Enter a value to search in the BST: 5
5 is present in the BST.
```

```
Enter a value to search in the BST: 0
0 is not present in the BST.
```

BST:



Time Complexity: The time complexity of the search operation in a Binary Search Tree (BST) is O(h), where h is the height of the tree. In a balanced BST, the height is O(log n), where n is the number of nodes in the tree. However, in a worst-case scenario where the tree is skewed, the height can be O(n), making the search operation take linear time.

Space Complexity: If we consider the call stack space used during the recursive search operation in a Binary Search Tree (BST), the space complexity becomes O(h), where h is the height of the tree. In a balanced BST, the height is O(log n), where n is the number of nodes in the tree.
However, in a worst-case scenario where the tree is a skewed tree, the height can be O(n), making the space complexity of the search operation O(n) due to the large number of function calls on the stack.

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

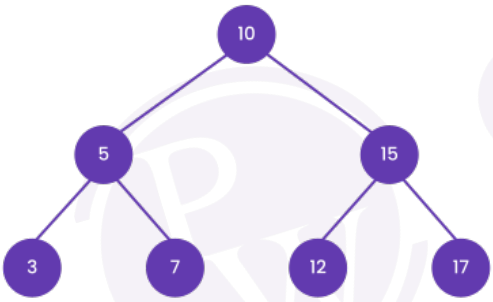
Iterative Code Explanation: In this program, we first create an empty BST and prompt the user to enter the number of values to insert in the BST and the values themselves using the Scanner class. Then, we insert the values in the BST iteratively using a while loop and the Node class. Finally, we prompt the user to enter a value to search in the BST using the Scanner class, and check if the value is present in the BST iteratively using the search() method of the BST class, and print the appropriate message.

Output

```
Enter a value to search in the BST: 7
7 is present in the BST.
```

```
Enter a value to search in the BST: 99
99 is not present in the BST.
```

BST Diagram



Time Complexity
The time complexity of searching in a binary search tree (BST) is O(h), where h is the height of the tree. In the worst case, where the BST is skewed and has only one branch, the time complexity of searching becomes O(n), where n is the number of nodes in the tree.

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).

Insertion

Code: https://pastebin.com/3Zpb6ena

Explanation:

1. The insertIntoBST method takes two parameters - the root node of the BST, and the value to be inserted.
2. If the root node is null, then the value to be inserted becomes the new root node of the BST. This is done by

creating a new TreeNode object with the given value, and returning it as the new root node.

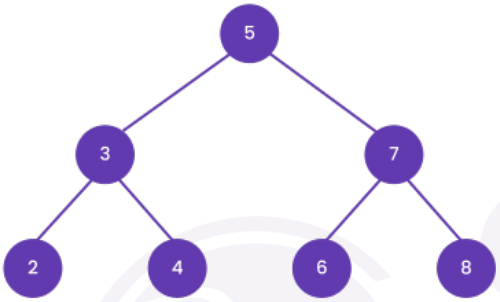
3. If the root node is not null, then the method checks whether the value to be inserted is less than or greater than the value of the root node. This is done by comparing the val field of the TreeNode object with the given value.
4. If the given value is less than the value of the root node, then the method recursively calls itself with the left child of the root node as the new root node, and the same value to be inserted.
5. If the given value is greater than the value of the root node, then the method recursively calls itself with the right child of the root node as the new root node, and the same value to be inserted.
6. The method returns the root node of the BST after inserting the new value.

Output:

```
Enter the values to be inserted into the BST: (enter -1 to stop)
5 7 3 8 2 4 6 -1
BST Created with given values

Inorder traversal of the BST:
2 3 4 5 6 7 8
```

The BST created is:



Time Complexity:
The time complexity of inserting a node into a BST is O(h), where h is the height of the BST. In the worst case, when the BST is skewed, the height of the tree can be n, where n is the number of nodes in the tree. Therefore, the time complexity of the program is O(log n) on average and O(n) in the worst case.

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.

Traversals

Code:
https://pastebin.com/QPqHW5Fv

Explanation of the code:
This program first creates a Node class to represent the nodes in the BST, with each node having a key value and left and right subtrees. It then creates a BST class with methods to insert a node, perform inorder, preorder, and postorder traversals, and a main method to test the BST. The inorderTraversal, preorderTraversal, and postorderTraversal methods perform the respective traversals recursively.

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.

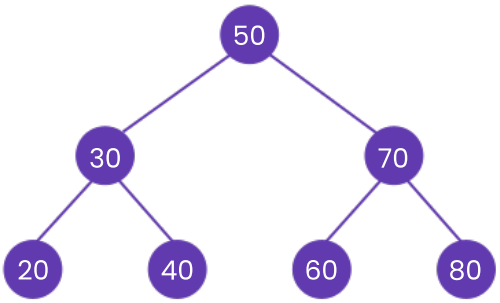
Preorder Traversal:

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.

Binary Search Tree



Output

```
Inorder traversal: 20 30 40 50 60 70 80
Preorder traversal: 50 30 20 40 70 60 80
Postorder traversal: 20 40 30 60 80 70 50
```

Time Complexity

The time complexity of the traversal operations is O(n), as each node is visited once.

Space Complexity

The space complexity of the traversal methods is also O(h), where h is the height of the tree, as it uses the call stack to keep track of the recursive function calls, and the maximum depth of the call stack is equal to the height of the tree, which can be up to O(n) in the worst case.

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:

1. **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.
2. **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:

- **Leaf node:** A node with no children.
- **Node with one child:** A node with only one child.
- **Node with two children:** A node with two children

3. **Delete the leaf node:** If the node to be deleted is a leaf node, we can simply remove it from the tree.
4. **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.
5. **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.
6. **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.
7. **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.

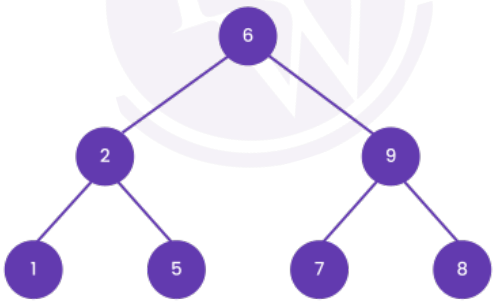
Code: https://pastebin.com/gsMgbprF

Output:

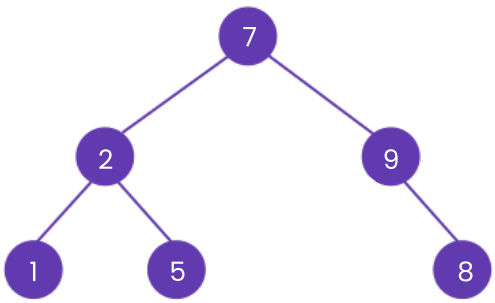
```
Enter the number of nodes in the BST: 7
Enter the values of the nodes: 6 2 9 8 5 1 7

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

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).

Upcoming Class Teasers

- Binary Search Trees 2