**Lab-10: 19CSE212 Data Structure and Algorithms**

**Implement Binary Search Tree: ADT, Algorithms, and Applications**

**Mode: Online, Dt. 26-04-2024**

1. **Construct a BST by inserting the following keys one by one: 10, 5, 8, 15, 13, 7, 6.**
2. **Write an algorithm for BST construction.**
3. **Analyse the time complexity of your algorithm.**
4. **Implement your algorithm in Python.**

**Solution Clue:**

1. **Algorithm for BST Construction from the given list of keys**

**Node Class Definition:**

Define a class Node with attributes left, right, and value.

This class represents each node of the BST.

Initialize the attributes with None and the given key value.

**Insert Function:**

Define a function **insert(root, key)** to insert a new key into the BST.

Check if the root is None, if so, create a new node with the given key and return it.

**If the root exists, compare the key with the root's value:**

If the key is less than the root's value, recursively insert it into the left subtree.

If the key is greater than or equal to the root's value, recursively insert it into the right subtree.

Return the updated root.

**In-order Traversal Function:**

Define a function **inorder\_traversal(root)** to perform an inorder traversal of the BST.

**If the root is not None:**

Recursively, traverse the left subtree.

Print the value of the current node.

Recursively traverse the right subtree.

**Constructing the Binary Search Tree:**

Define a list of keys containing the keys to be inserted into the BST.

Initialize the root of the BST to None.

**Iterate through each key in the keys list:**

Insert the key into the BST using the insert function.

**Printing the Inorder Traversal:**

Print a message indicating that the inorder traversal of the BST is being performed.

Call the inorder\_traversal function with the root of the BST as the argument to print the elements in sorted order.

1. **Time Complexity Analysis of the above Algorithm**

**Insertion Operation:**

In the worst case, inserting all keys might require traversing the height of the BST.

Since the BST is not balanced in the general case, the worst-case time complexity for insertion is O(h), where h is the height of the BST.

However, in the average case, when the BST is reasonably balanced, the time complexity for insertion approaches O(log n), where n is the number of nodes in the BST.

Therefore, the time complexity for inserting a single key is O(h) in the worst case, and O(log n) in the average case.

**In-order Traversal:**

In the in-order traversal function, each node is visited exactly once.

The time complexity of in-order traversal is O(n), where n is the number of nodes in the BST, because in the worst case, we visit every node once.

**Constructing the Binary Search Tree:**

Iterating through the list of keys and inserting them into the BST takes O(n \* h) time in the worst case, where n is the number of keys and h is the height of the BST.

In the average case, when the keys are inserted in a balanced manner, the time complexity is O(n \* log n).

**The overall, worst-case time complexity of the algorithm is :**

**T(n) = O(h) + O(n) + O(n \* h) = O(n \* h).**

**In the average case, T(n) = O(n \* logn), where h = logn.**

1. **Python Code Snippet:**

**class Node:**

**def \_\_init\_\_(self, key):**

**self.left = None**

**self.right = None**

**self.val = key**

**def insert(root, key):**

**if root is None:**

**return Node(key)**

**else:**

**if key < root.val:**

**root.left = insert(root.left, key)**

**else:**

**root.right = insert(root.right, key)**

**return root**

**def inorder\_traversal(root):**

**if root:**

**inorder\_traversal(root.left)**

**print(root.val)**

**inorder\_traversal(root.right)**

**# Constructing the Binary Search Tree**

**keys = [10, 5, 8, 15, 13, 7, 6]**

**root = None**

**for key in keys:**

**root = insert(root, key)**

**# Printing the inorder traversal of the constructed Binary Search Tree**

**print("Inorder Traversal of BST:")**

**inorder\_traversal(root)**

**Assignments:**

**Implement the following ADTs in Python based on the above BST.**

1. **External\_nodes(root): Returns all external nodes.**
2. **Descendants(root): Returns all descendants of the root.**
3. **Ancestors(root, 6): Returns all ancestors of the node containing the key 6.**
4. **Siblings(root, 7): Returns the node's sibling containing the key 7.**
5. **Search\_key\_with\_index(root, 6): Search the key 6, and returns the node index, where 6 resides.**
6. **Path\_to\_leaf(root, 6): Finds and returns the path from the root to the leaf node containing the key 6, where the path includes the list of nodes starting from the root to the node containing the key 6.**
7. **Delete(root, 10): Deletes the node containing the key 10, and returns the modified BST.**

**Scenario 1 (Sample for your Understanding and Practice)**

A bookstore regularly manages a diverse inventory of books ranging from bestsellers to textbooks. The store requires an efficient system to handle various operations such as searching for books, updating stock levels, and processing sales quickly. Each book is identified by a unique ISBN (International Standard Book Number), making it an ideal key for a Binary Search Tree (BST) structure.

**System Requirements:**

**Data Structure for Inventory:** Use a BST where each node represents a book. Each node stores the ISBN, title, author, quantity in stock, and shelf location.

**Operations to be Supported:**

**Insertion:** Add new titles to the inventory as they arrive.

**Deletion:** Remove titles that are no longer sold or are out of stock permanently.

**Search:** Quickly find a book by its ISBN to check details or update inventory.

**Traversal:** Generate a list of all books in order of ISBN for inventory checks and to assist in restocking.

**Update:** Modify the stock level of a book after a sale or new stock arrival.

**Based on the above scenario, answer the following questions.**

1. **Write an algorithm to meet the system requirements based on the BST ADTs.**
2. **Analyse the worst-case running time of the algorithm.**
3. **Write the viable code abstractions based on your algorithm.**
4. **Implement your code abstractions in Python.**

**Solution Clue:**

1. **Algorithm to Meet System Requirements Using BST**

**BST Node Structure:**

Each node will contain:

isbn (key)

title

author

quantity

shelf\_location

left (left child)

right (right child)

**Operations:**

**Insertion:** Insert a new book into the BST based on its ISBN.

**Deletion:** Remove a book by ISBN from the BST.

**Search:** Find a book by ISBN.

**Traversal:** In-order traversal to generate a sorted list of books by ISBN.

**Update:** Update the quantity of a book in stock.

**2. Worst-case Running Time of the above algorithm**

**Insertion:** O(h), where h is the height of the BST. In the worst case (skewed tree), this is O(n).

**Deletion:** O(h), also worst case O(n) for a skewed tree.

**Search:** O(h), worst case O(n).

**Traversal:** O(n) since each node is visited once.

**Update:** O(h), worst case O(n).

**Overall worst-case running time:**

**T(n) = O(n) + O(n) + O(n) + O(n) + O(n) = O(n)**

1. **Viable Code Abstractions**

**BookStoreBST:**

**insert(isbn, title, author, quantity, shelf\_location):** Inserts a new book.

**delete(isbn):** Removes a book.

**search(isbn):** Finds and returns a book node.

**in\_order\_traversal():** Returns a list of all books in ISBN order.

**update\_stock(isbn, new\_quantity):** Updates stock level of a book.

1. **Implementation of Code Abstractions in Python (Sample code)**

**class BSTNode:**

**def \_\_init\_\_(self, isbn, title, author, quantity, shelf\_location):**

self.isbn = isbn

self.title = title

self.author = author

self.quantity = quantity

self.shelf\_location = shelf\_location

self.left = None

self.right = None

**class BookStoreBST:**

**def \_\_init\_\_(self):**

self.root = None

**def insert(self, isbn, title, author, quantity, shelf\_location):**

if not self.root:

self.root = BSTNode(isbn, title, author, quantity, shelf\_location)

else:

self.\_insert(self.root, isbn, title, author, quantity, shelf\_location)

**def \_insert(self, node, isbn, title, author, quantity, shelf\_location):**

if isbn < node.isbn:

if node.left:

self.\_insert(node.left, isbn, title, author, quantity, shelf\_location)

else:

node.left = BSTNode(isbn, title, author, quantity, shelf\_location)

elif isbn > node.isbn:

if node.right:

self.\_insert(node.right, isbn, title, author, quantity, shelf\_location)

else:

node.right = BSTNode(isbn, title, author, quantity, shelf\_location)

**def delete(self, isbn):**

self.root = self.\_delete(self.root, isbn)

**def \_delete(self, node, isbn):**

if node is None:

return None

if isbn < node.isbn:

node.left = self.\_delete(node.left, isbn)

elif isbn > node.isbn:

node.right = self.\_delete(node.right, isbn)

else:

if node.left is None:

return node.right

elif node.right is None:

return node.left

else:

min\_larger\_node = self.\_find\_min(node.right)

node.isbn, node.title, node.author, node.quantity, node.shelf\_location = \

min\_larger\_node.isbn, min\_larger\_node.title, min\_larger\_node.author, \

min\_larger\_node.quantity, min\_larger\_node.shelf\_location

node.right = self.\_delete(node.right, min\_larger\_node.isbn)

return node

**def \_find\_min(self, node):**

while node.left:

node = node.left

return node

**def search(self, isbn):**

return self.\_search(self.root, isbn)

**def \_search(self, node, isbn):**

if node is None or node.isbn == isbn:

return node

elif isbn < node.isbn:

return self.\_search(node.left, isbn)

else:

return self.\_search(node.right, isbn)

**def in\_order\_traversal(self):**

return self.\_in\_order\_traversal(self.root)

**def \_in\_order\_traversal(self, node):**

result = []

if node:

result.extend(self.\_in\_order\_traversal(node.left))

result.append((node.isbn, node.title, node.author, node.quantity, node.shelf\_location))

result.extend(self.\_in\_order\_traversal(node.right))

return result

**def update\_stock(self, isbn, new\_quantity):**

book\_node = self.search(isbn)

if book\_node:

book\_node.quantity = new\_quantity

**# Sample Inputs**

bookstore = BookStoreBST()

bookstore.insert(1, "Data Structures", "Author One", 10, "B1")

bookstore.insert(2, "Algorithms", "Author Two", 20, "B2")

print(bookstore.in\_order\_traversal())

bookstore.update\_stock(1, 8)

print(bookstore.search(1).quantity)

bookstore.delete(2)

print(bookstore.in\_order\_traversal())

**Assignment**

**Scenario 2**

A real estate agency manages a large portfolio of property listings that are dynamically added and removed based on market availability. The agency wants a system that can efficiently manage, search, and update these listings based on their prices, which frequently fluctuate.

**Data Structure Requirements:**

Properties are organized in a Binary Search Tree (BST) where each node represents a property listing. The key for each node in the BST is the listing price, which allows the agency to efficiently perform operations based on property prices.

**Each property node contains:**

Price (key)

Listing ID

Address

Square Footage

Number of Bedrooms

Number of Bathrooms

Listing Agent

**System Requirements:**

**Property Addition:** Add new properties as they come onto the market.

**Property Deletion:** Remove properties that are sold or taken off the market.

**Price Update:** Adjust the price of a property, which might require node repositioning in the BST.

**Search by Price:** Find properties within a specific price range.

**Listing Display:** Display all properties in order of their price.

Operations to Be Supported:

**Add Property:** Insert a new property into the BST based on its price.

**Remove Property:** Delete a property from the BST.

**Update Property Price:** Update the price of a property and adjust the BST structure if necessary.

**Search Properties by Price Range:** Return a list of properties whose prices fall within a given range.

**Display Properties:** Perform an in-order traversal to list all properties sorted by price.

**Based on the above scenario, answer the following questions.**

1. **Write an algorithm to meet the system requirements based on the BST ADTs.**
2. **Analyse the worst-case running time of the algorithm.**
3. **Write the viable code abstractions based on your algorithm.**
4. **Implement your code abstractions in Python.**