Data Structures
Lab Manual 8

Topic: Binary Search Trees

Session: Spring 2023

Faculty of Information Technology

UCP Lahore Pakistan

Objectives:
The objective of learning about binary search trees (BSTs) is to understand their properties, operations, and algorithms associated with them.
Here are the key objectives:

1. Efficient searching: Binary search trees provide an efficient searching mechanism for data. Understanding BSTs helps in analyzing and implementing algorithms like binary search, which leverages the properties of BSTs to quickly find elements of interest. By learning about BSTs, you gain insights into the underlying principles of binary search and its time complexity.

2. Ordered data storage and traversal: Binary search trees store data in an ordered manner. Learning about BSTs helps in understanding algorithms for traversing the tree in specific orders like in-order, pre-order, and post-order. These traversals are fundamental for many tree-related algorithms and operations.

3. Insertion and deletion operations: BSTs support efficient insertion and deletion operations while maintaining the order of the elements. Learning about BSTs equips you with the knowledge to implement algorithms for inserting new elements into the tree and removing existing ones while ensuring the integrity of the tree structure.

4. Balanced BSTs: Understanding binary search trees is crucial for grasping the concepts of balanced tree structures such as AVL trees and red-black trees. These balanced BSTs provide guarantees on the tree's height, ensuring efficient search, insertion, and deletion operations in worst-case scenarios. Learning about BSTs forms a foundation for understanding and implementing these more complex self-balancing tree structures.

5. Analysis of algorithms: BSTs serve as a basis for analyzing the time complexity of various algorithms. By learning about BSTs, you gain the ability to analyze the average and worst-case time complexities of operations performed on BSTs, enabling you to make informed decisions about algorithm design and selection.

6. Real-world applications: Binary search trees have numerous real-world applications, such as database systems, symbol tables, spell-checking, and more. Understanding BSTs in the context of DSA helps you recognize and utilize these applications effectively.

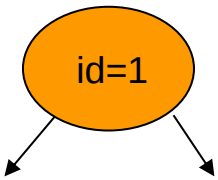
Overall, learning about binary search trees provides you with a solid understanding of a fundamental data structure, its operations, associated algorithms, and its relevance in solving various computational problems efficiently.

Binary Search Tree:
A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

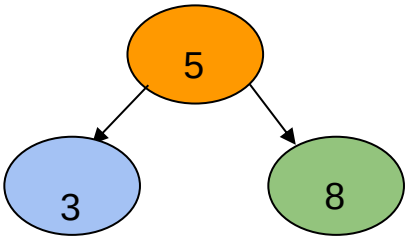
The root of a BST is the node that has the smallest value in the left subtree and the largest value in the right subtree. Each left subtree is a BST with nodes that have smaller values than the root and each right subtree is a BST with nodes that have larger values than the root.
Binary Search Tree is a node-based binary tree data structure that has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches)

Data Items
The data items in a binary search tree are of generic data type. Each data item has a key (of generic type) that uniquely identifies the data item. Data items usually include additional data.



Structure
The data items form a binary tree. For each data item D in the tree, all the data items in D's left subtree have keys that are less than D's key and all the data items in D's right subtree have keys that are greater than D's key.



//3 Tree traversal functions (Depth first)
PreOrderTraversal ()
Traverse the BST to print in **root->left->right** order
PostOrderTraversal ()
Traverse the BST to print in **left->right-> root** order
InOrderTraversal ()
Traverse the BST to print in **left-> root ->right** order

List of basic operation that can perform on BTS:
Binary search trees (BSTs) support several basic operations that can be performed on them. Here is a list of the common operations performed on a BST:

1. Insertion: Adding a new element to the BST while maintaining its ordered structure. The element is placed in the appropriate position based on its value, comparing it with the existing nodes' values.

2. Deletion: Removing a specific element from the BST while preserving the BST properties. This operation involves restructuring the tree to maintain the ordering of the remaining elements.

3. Searching: Finding whether a given element is present in the BST. Starting from the root node, comparisons are made to navigate the tree by moving left or right based on the element's value, until either the target element is found or a leaf node is reached.

4. Traversal: Visiting all the elements of the BST in a specific order. There are different types of tree traversals:
In-order traversal: Visiting the left subtree, then the current node, and finally the right subtree. This traversal visits the nodes in ascending order for BSTs with unique values.
Pre-order traversal: Visiting the current node, followed by the left subtree, and then the right subtree.
Post-order traversal: Visiting the left subtree, followed by the right subtree, and finally the current node.

5. Minimum and Maximum: Finding the smallest and largest elements in the BST, respectively. The minimum element is located in the leftmost node, while the maximum element is located in the rightmost node of the BST.

6. Successor and Predecessor: Finding the element that comes immediately after or before a given element in the BST's sorted order. The successor is the smallest element greater than the given element, while the predecessor is the largest element smaller than the given element.

7. Height: Determining the height of the BST, which is the length of the longest path from the root node to any leaf node. The height of an empty tree is typically defined as -1, and a tree with a single node has a height of 0.

8. Checking BST Properties: Verifying if a given tree satisfies the properties of a BST. These properties include the ordering of the nodes, where the left subtree contains values smaller than the current node, and the right subtree contains values greater than the current node.

These are some of the basic operations performed on binary search trees. Understanding and implementing these operations allows for efficient data manipulation and retrieval within a binary search tree data structure.

Binary Search Tree Applications:

- In multilevel indexing in the database
- For dynamic sorting
- For managing virtual memory areas in Unix kernel

Insert a node into a BST: A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

Sample Task: C++ program to implements a binary search tree (BST) and performs insertion and deletion operations on the tree.

Pseudocode:

1. Define a structure called Node to represent a node in the BST. Each node has an integer key value and two pointers, left and right, pointing to its left and right child nodes.

2. The createNode function is defined to create a new node with a given key. It allocates memory for the node, initializes its members, and returns a pointer to the new node.

3. The insertNode function is defined to insert a key into the BST. It takes the current root node and the key to be inserted as input. If the tree is empty (root is null), it creates a new node with the given key and returns it as the new root. Otherwise, it recursively traverses the tree by comparing the key with the current node's key. If the key is smaller, it calls insertNode on the left child of the current node. If the key is larger, it calls insertNode on the right child. The function returns the root node after insertion.

4. The findMinNode function is defined to find the minimum key value in a given subtree. It starts from the given node and traverses left until it reaches the leftmost node (minimum key). It returns a pointer to the minimum node.

5. The deleteNode function is defined to delete a key from the BST. It takes the current root node and the key to be deleted as input. If the tree is empty (root is null), it simply returns. Otherwise, it recursively searches for the key by comparing it with the current node's key. If the key is smaller, it calls deleteNode on the left child. If the key is larger, it calls deleteNode on the right child. If the key is found, it handles three cases:

    - If the node to be deleted has no children or only one child, it adjusts the pointers and deletes the node.
    - If the node has two children, it finds the minimum node from the right subtree, replaces the key of the node to be deleted with the minimum key, and then recursively deletes the minimum node from the right subtree. The function returns the root node after deletion.

6. The inorderTraversal function performs an in-order traversal of the BST (left subtree, current node, right subtree) and prints the keys in ascending order.

7. The main function is the entry point of the program. It initializes the root node as null and inserts several nodes into the BST using the insertNode function.

8. After the insertions, it prints the BST by calling inorderTraversal and displays the keys in ascending order.

9. It then deletes a node with a key of 20 using the deleteNode function and displays the updated BST by calling inorderTraversal again.

10. Finally, the program returns 0 to indicate successful execution.

Implementation:

```
#include <iostream>

struct Node {
    int key;
    Node* left;
    Node* right;
};

Node* createNode(int key) {
    Node* newNode = new Node;
    newNode->key = key;
    newNode->left = nullptr;
    newNode->right = nullptr;
    return newNode;
}

Node* insertNode(Node* root, int key) {
    if (root == nullptr) {
        return createNode(key);
    }


    if (key < root->key) {
        root->left = insertNode(root->left, key);
    }
    else if (key > root->key) {
        root->right = insertNode(root->right, key);
    }

    return root;
}


Node* findMinNode(Node* node) {
    Node* current = node;
    while (current && current->left != nullptr) {
        current = current->left;
    }
    return current;
}


Node* deleteNode(Node* root, int key) {

    if (root == nullptr) {
        return root;
    }


    if (key < root->key) {
        root->left = deleteNode(root->left, key);
    }
    else if (key > root->key) {
        root->right = deleteNode(root->right, key);
    }
    else {

        if (root->left == nullptr) {
```

```
            Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }


        Node* temp = findMinNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

void inorderTraversal(Node* root) {
    if (root != nullptr) {
        inorderTraversal(root->left);
        std::cout << root->key << " ";
        inorderTraversal(root->right);
    }
}

int main() {
    Node* root = nullptr;
    root = insertNode(root, 50);
    root = insertNode(root, 30);
    root = insertNode(root, 20);
    root = insertNode(root, 40);
    root = insertNode(root, 70);
    root = insertNode(root, 60);
    root = insertNode(root, 80);

    std::cout << "BST after insertion: ";
    inorderTraversal(root);
    std::cout << std::endl;


    root = deleteNode(root, 20);

    std::cout << "BST after deletion: ";
    inorderTraversal(root);
    std::cout << std::endl;

    return 0;
}
```

Task For Lab:

Task 1: Implement the following operations of BST. Note that the Node class must have 3 members i.e., **int data, Node* left, Node* right**.

The BST class should have a pointer of type Node, called **root**, i.e., **Node* root**.

BSTree ()
Constructor. Creates an empty binary search tree.

insert ()
Inserts new DataItem into a BST. If a data item already exists in the BST, then by definition of having distinct values in BST, you should not accept the duplicate value. Else if the provided data is unique, insert it into the tree.

retrieve ()
Searches BST for the data item with the user given given value. Return true if it already exists in BST, else return false.

Task 2: Write a C++ program to implements a binary search tree (BST) and performs insertion with the following functions.:

1. Pre-Order Traversal. (**root->left->right)**
2. In-Order Traversal. (**left->root->right)**
3. Post-Order Traversal. (**left->right-> root)**

Task 3: A **tree** is a nonlinear hierarchical data structure that consists of nodes connected by edges. One of the nodes is designated as "**root node**" and the remaining nodes are called **child nodes** or the **leaf nodes** of the root node. In general, each node can have as many children but only one parent node.

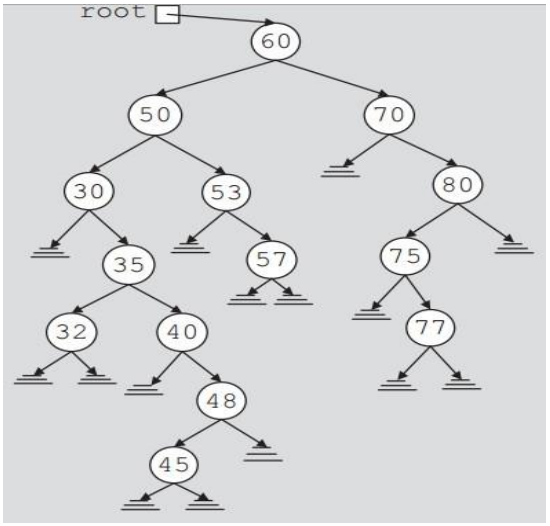
Your task is to create generic abstract classes named **Tree** containing a root node and **Node** having a key and links to the left and right children as follows:

```
template <class Type>
class Node
{
public:
Type key;
Node<Type>* leftChild;
Node<Type>* rightChild;
};

template <class Type>
class Tree : public Node
{
protected:
```

```
Node<Type>* root;
public:
        Tree();
        virtual void insert(Type) = 0;  // Inserts a new node with an element of type Type in
        the Tree.
        virtual bool deleteNode(Type) = 0;  // Removes any particular node from the Tree
        and returns true if the node is deleted and false if not.
};
```

Task 4: Convert the **insertNode** function into a recursive code of insertion in the BST. Using your code, create the binary tree (given below).



Note: Must submit proof of dry-run with your codes as well.