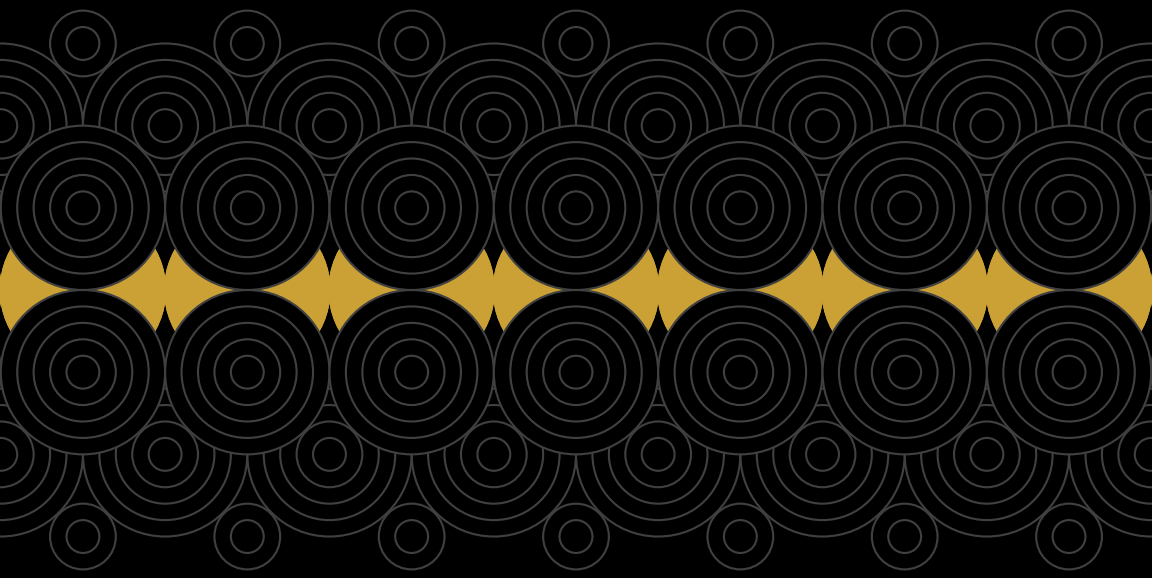
DATA STRUCTURES

BST

BINARY SEARCH TREE



DATA STRUCTURES

BST

BINARY SEARCH TREE

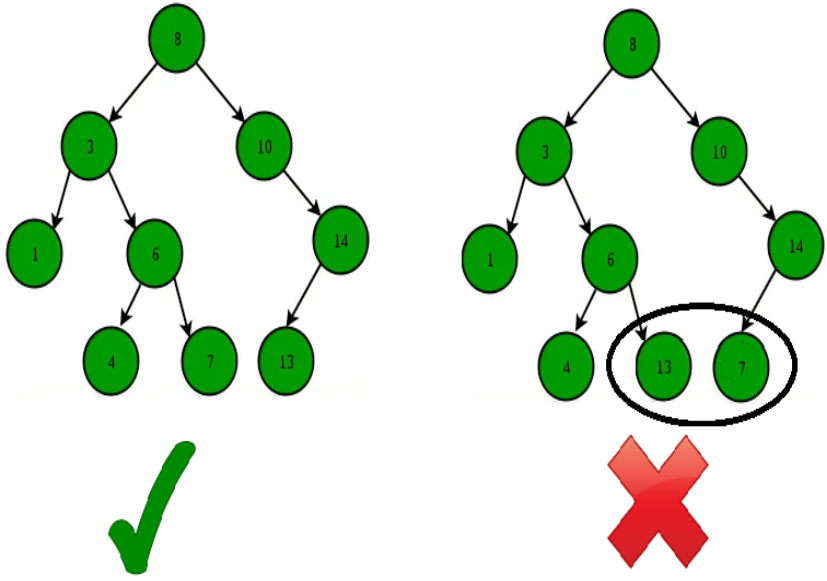
Conteúdo

I
BINARY SEARCH TREE

I.1 Introduction

* A BST is structured such that each node has at most two children, known as the left and right child. The key property of a BST is that for any given node, all values in the left subtree are less than the node's value, while all values in the right subtree are greater. This inherent ordering allows for efficient in-order traversal, producing sorted output with ease.

BST REPRESENTATION



I.2 Operations on Binary Search Trees

*** Insertion**

Inserting a new key into a BST involves comparing the key with the keys in the tree, moving left or right accordingly until a suitable null position is found.

Program:

```
struct Node
int key;
Node* left;
Node* right;
;
```

```
    Node* insert(Node* root, int key)
if (root == nullptr)
Node* newNode = new Node();
newNode.key = key;
newNode.left = nullptr;
newNode.right = nullptr;
return newNode;


    if (key < root.key)
root.left = insert(root.left, key);
else
root.right = insert(root.right, key);

    return root;
```

***Deletion**

Deleting a key from a BST can be more complex as it requires maintaining the BST properties.

Program:

```
    Node* minValueNode(Node* node)
Node* current = node;
while (current and 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
// Node with only one child or no child
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 with two children: Get the inorder successor

Node* temp = minValueNode(root.right);
root.key = temp.key;
root.right = deleteNode(root.right, temp.key);

return root;
```

I.3 Traversals of Binary Search Trees

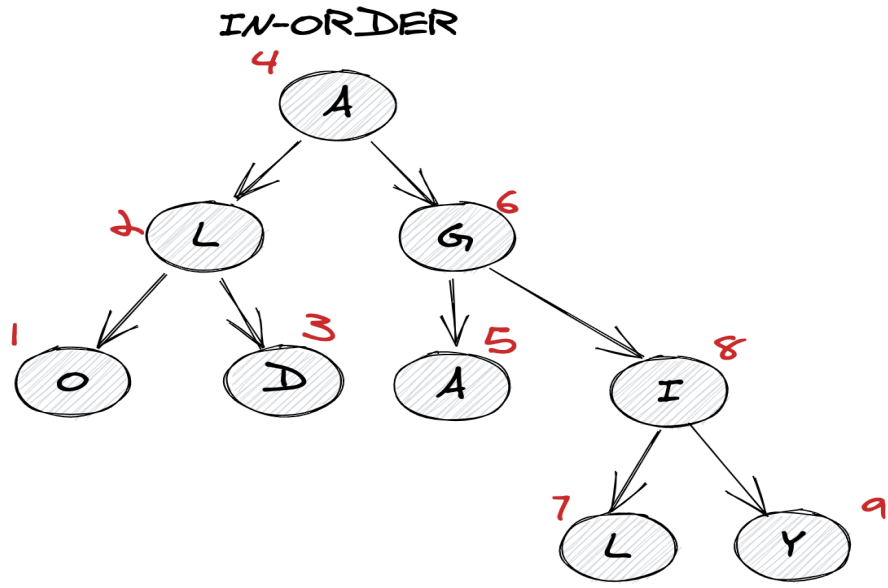
Traversing a BST means visiting all the nodes in a specific order. The most common methods are Inorder, Preorder, and Postorder.

***Inorder**

Inorder traversal visits the nodes in a non-decreasing order.

Program:

```
void inorder(Node* root)
if (root != nullptr)
inorder(root.left);
std::cout « root.key « ;
inorder(root.right);
```

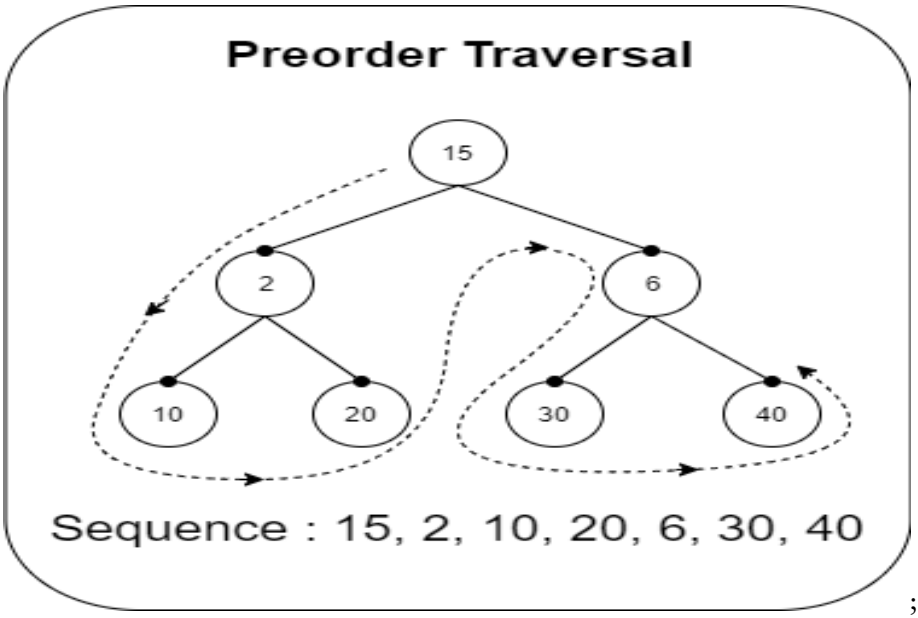


***Preorder Traversal**

Preorder traversal visits the root node first, then the left subtree, and finally the right subtree.

Program:

```
void preorder(Node* root)
if (root != nullptr)
print(root.key) ;
preorder(root.left);
preorder(root.right);
```



;

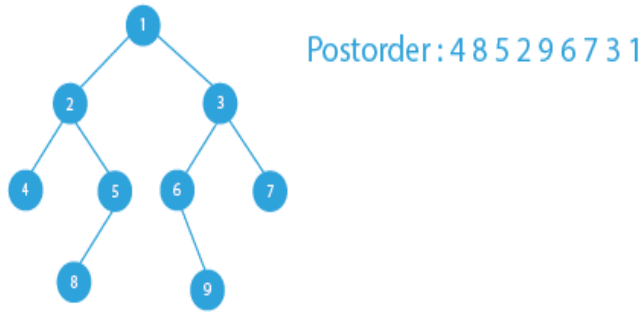
***Postorder Traversal**

Postorder traversal visits the left subtree, the right subtree, and finally the root node.

Program:

```
void postorder(Node* root)
if (root != nullptr)
```

```
postorder(root.left);
postorder(root.right);
print(root.key);
```



;

I.4 Recursive Functions in BST

Understanding Recursion: Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem. It consists of two main parts:

*Base Case: This is the condition under which the recursion ends. It prevents infinite recursion.
*Recursive Case: This is the part where the function calls itself to process smaller parts of the problem. In the context of Binary Search Trees (BST), many operations can be elegantly implemented using recursion, such as insertion, deletion, and traversal.

Recursive Functions Explained:
Let's break down how recursion is applied in BST operations with clear examples.

1. Insertion in BST

When inserting a key into a BST, we compare the key with the current node's key:

If it's less, we move to the left subtree. If it's greater, we move to the right subtree. If we find a nullptr, we insert the new node therelist.

Example: Insertion Function Node* insert(Node* root, int key)
// Base case: If the tree is empty, create a new node

if (root == nullptr)
Node* newNode = new Node();
newNode.key = key;
newNode.left = nullptr;
newNode.right = nullptr;
return newNode;
Return the new node

// Recursive case: Decide whether to go left or right if (key < root.key)
// Call insert on the left subtree
root.left = insert(root.left, key);
else
// Call insert on the right subtree
root.right = insert(root.right, key);

// Return the unchanged root pointer
return root;

I.5 Walkthrough of the Insertion Process

Suppose we want to insert the keys 50, 30, 70, 20, 40, 60, and 80 into the BST.

1.Insert 50: Tree is empty, create root node 50.

2.Insert 30:

*Compare with 50, move left (because 30 < 50).

*Left is nullptr, insert 30 here.

3.Insert 70:

*Compare with 50, move right (because 70 > 50).

*Right is nullptr, insert 70 here.

4.Insert 20:

*Compare with 50 (left to 30).

*Compare with 30, move left (because 20 < 30).

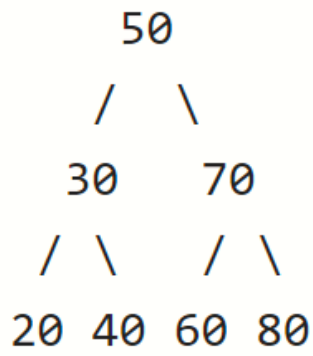
*Left is nullptr, insert 20.

5.Insert 40: Similar process leads to 40 being placed as the right child of 30.

6.Insert 60: Placed as the left child of 70.

7.Insert 80: Placed as the right child of 70.

The tree structure after all insertions will look like this:



2. Traversal of BST

Traversals allow us to visit all the nodes in the tree in a specific order. The three primary traversal methods are Inorder, Preorder, and Postorder.

Inorder Traversal Inorder traversal visits nodes in ascending order: left subtree, root, right subtree.

Program:

```
void inorder(Node* root)
if (root != nullptr)  // Base case
inorder(root.left); // Recur on left child
print( root.key); // Visit the root
inorder(root.right); // Recur on right child
```

Example of Inorder Traversal:

For the BST we created, the output of inorder(root) will be:

20 30 40 50 60 70 80

Program:

```
void preorder(Node* root)
if (root != nullptr)  // Base case
print( root.key); // Visit the root
preorder(root.left); // Recur on left child
preorder(root.right); // Recur on right child
```

Example of Preorder Traversal:

For the same BST, the output of preorder(root) will be:

50 30 20 40 70 60 80

Postorder Traversal
Postorder visits the left subtree, right subtree, and then the root.
Program:

```
void postorder(Node* root)
if (root != nullptr)  // Base case
postorder(root.left); // Recur on left child
postorder(root.right); // Recur on right child
print(root.key); // Visit the root
```

Example of Postorder Traversal:

For the same BST, the output of postorder(root) will be:

20 40 30 60 80 70 50

Deletion in BST
Deletion can also be handled using recursion, but it requires additional considerations, especially when removing a node with two children.

Example: Deletion Function
Here's how deletion is handled recursively:

```
   Node* deleteNode(Node* root, int key)
if (root == nullptr) return root; // Base case

// Traverse the tree
if (key < root.key)
root.left = deleteNode(root.left, key); // Go left
else if (key > root.key)
root.right = deleteNode(root.right, key); // Go right
else
// Node found
if (root.left == nullptr)
Node* temp = root.right; // Node with one child
delete root;
return temp;
else if (root.right == nullptr)
Node* temp = root.left; // Node with one child
delete root;
return temp;


   // Node with two children
Node* temp = minValueNode(root.right); // Get the inorder successor
root.key = temp.key; // Copy successor's value to this node
root.right = deleteNode(root.right, temp.key); // Delete the successor

return root; // Return the updated root
```

I.6 Conclusion

Binary Search Trees provide a better way to manage sorted data efficiently. Understanding the fundamental operations—insert, delete, search—and traversal methods is crucial for effective use of BSTs in various applications.

An E-Book By

ASIHF MOHAMED
MOHAMED SAMEER
MADHANAGOPAL
SHRINATH VENKATESH