PW SKILLS

Lesson Plan

BST-2

PW SKILLS

Prerequisites:

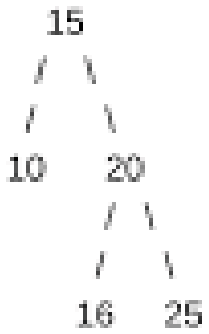
- Understanding of basic data structures like arrays and linked lists.
- Understanding of recursion.
- Proficiency in a C++ programming language

Today's Checklist:

- Inorder predecessor and successor for a given key in BST
- Deletion in a BST [Leetcode-450]

Inorder predecessor and successor for a given key in BST

The in order predecessor of a given key is the previous greater element in BST. If there is no predecessor then return null.



Key = 15, successor = 10
Key = 16, successor = 15
Key = 20, successor = 16
Key = 10, successor = null
Key = 25, successor = 20

Code:

```
#include <iostream>
using namespace std;

// BST Node
struct Node {
    int key;
    struct Node *left, *right;
};

void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{

    pre = NULL;
    suc = NULL;

    // set temp node as root
    Node* temp1 = root;
```

```
    while (temp1) {
        // the maximum value in left subtree is successor
        if (temp1->key > key) {
            suc = temp1;
            temp1 = temp1->left;
        }
        else
            temp1 = temp1->right;
    }
    Node* temp2 = root;
    while (temp2) {
        // the minimum value in right subtree is predecessor
        if (temp2->key < key) {
            pre = temp2;
            temp2 = temp2->right;
        }
        else
            temp2 = temp2->left;
    }
    return;
}

// A utility function to create a new BST node
Node* newNode(int item)
{
    Node* temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

/* A utility function to insert a new node with given key in
 * BST */
Node* insert(Node* node, int key)
{
    if (node == NULL)
        return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    return node;
}
```

Explanation:

- Takes BST root, key, and pre/suc Node pointers.
- Initializes pre and suc to NULL.
- Finds predecessor (pre) and successor (suc) nodes of the given key in BST using two while loops.
- Takes an integer item.
- Creates and returns a new Node with the given item as key, and left/right pointers set to NULL.
- Inserts a new node with the given key into the BST.
- Creates a new root if the tree is empty, else recursively inserts in the left/right subtree based on key comparison.

Time Complexity:
findPreSuc function:
Time complexity is O(h), where h is the height of the BST.
The function traverses the tree once for the predecessor and once for the successor.
insert function:
Time complexity is also O(h) where h is the height of the BST.
In the worst case, it involves traversing from the root to a leaf node.

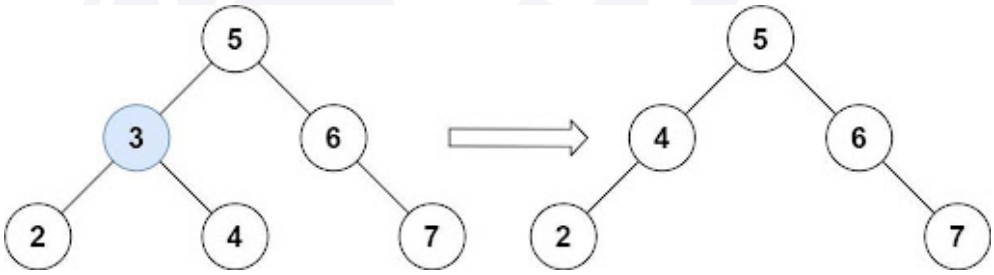
Space Complexity:
The space complexity for both findPreSuc and insert functions is O(1) excluding the recursive call stack.
Additional space is used only for the recursive call stack during function calls.
If the BST is balanced, the maximum space required for the call stack is O(log n), where n is the number of nodes.
In the worst case, when the tree is skewed (like a linked list), the space complexity for the call stack can be O(n).

Delete Node in a BST [Leetcode-450]

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:
Search for a node to remove.
If the node is found, delete the node.

Example 1:



Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]
Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.
One valid answer is [5,4,6,2,null,null,7], shown in the above BST.
Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.

Code:

```
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(root)
            if(key < root->val) root->left = deleteNode(root->left, key);
    //We frecursively call the function until we find the target node
```

```
            else if(key > root- >val) root- >right = deleteNode(root-->right, key);
            else{
                if(!root->left && !root→right) return NULL;  //No child condition
                if (!root->left || !root→right)
                    return root->left ? root→left : root->right;
                TreeNode* temp = root->left;
                while(temp->right ≠ NULL) temp = temp->right;
                root->val = temp->val;
                root->left = deleteNode(root->left, temp->val);
            }
        return root;
    }
};
```

Explanation:

- C++ class Solution with a deleteNode method for BST.
- Recursively searches for node with given key.
- Handles cases: no children, one child, and two children.
- For two children, replaces node's value with max value in left subtree.
- Continues recursively to delete the replaced node.
- Returns the modified tree.

Time Complexity:
O(h), where h is the height of the BST.
In the worst case, when the tree is skewed, the height h is equivalent to the number of nodes, making it O(n).
In balanced cases, the height h is logarithmic (O(log n)).

Space Complexity:
O(h) for the recursive call stack.
In the worst case, when the tree is skewed, the space complexity is O(n).
In balanced cases, the space complexity is logarithmic (O(log n)).

PW SKILLS

THANK YOU !

PW SKILLS