Binary Search Tree

This C++ file is a focused and practical guide to the core operations of a **Binary Search Tree (BST)**, with a special emphasis on the **recursive Delete algorithm**. The entire implementation is done in a procedural, C-style.

The file begins by establishing the fundamental recursive insertion (RInsert) and iterative search (search) functions, which are the building blocks of any BST. The true highlight, however, is the sophisticated Delete function. It demonstrates a robust strategy for handling node removal, which is often the most complex part of a BST implementation.

To achieve this, it uses several crucial helper functions: Height, InPre (In-order Predecessor), and InSucc (In-order Successor). The deletion logic is particularly insightful: when deleting a node with two children, it doesn't just arbitrarily pick a replacement. Instead, it **balances the tree** slightly by checking the heights of the left and right subtrees and choosing the replacement (predecessor or successor) from the taller subtree. This is a clever optimization to help maintain the tree's structure.

**Creating Binary Search Tree**

This first section (commented out) demonstrates the basic, **iterative** way to build and search a Binary Search Tree. It provides a solid foundation for understanding how the BST's ordering property is maintained.

* void insert(int key){ ... } This is the **iterative insertion** function. It uses two pointers, t (the traversal pointer) and r (the "tailing" parent pointer), to navigate down the tree to find the correct empty spot. Once found, a new node is created and attached to the parent r.
* struct Node \* search(int key){ ... } This is the **iterative search** function. It efficiently traverses the tree by making a simple decision at each node: if the key is smaller, go left; if larger, go right. This continues until the key is found or the traversal falls off the tree (reaches NULL).
* struct Node \*RInsert(struct Node \*p, int key){ ... } This is the **recursive insertion** function. It's a more elegant, "divide and conquer" approach. The function recursively calls itself on the left or right subtree until it hits a NULL pointer (the base case), at which point it creates the new node. The return values correctly link the new node back up the chain of recursive calls.

**Insert and delete Recursive functions in BST**

This is the main, active part of the file. It focuses on a fully recursive implementation for both insertion and deletion, showcasing a robust and balanced approach to removing nodes.

* int Height(struct Node \*p) { ... } This is a standard helper function that recursively calculates the **height** of a tree (the number of nodes on the longest path from the root to a leaf). It works by finding the height of the left and right subtrees and returning 1 + the maximum of the two.
* struct Node\* InPre(struct Node \*p) { ... } This function finds the **In-order Predecessor** of a given node. For a node with two children, its predecessor is the **largest value in its left subtree**. The function finds this by starting at the left child and then traversing as far right as possible.
* struct Node\* InSucc(struct Node \*p) { ... } This function finds the **In-order Successor**. This is the **smallest value in the node's right subtree**. It's found by starting at the right child and then traversing as far left as possible.
* struct Node\* Delete(struct Node \*p, int key) { ... } This is the core recursive deletion algorithm. It's designed to find and remove a node while preserving the BST property. \* if (p == NULL){ return NULL; } This is a **base case**. If the node is NULL (meaning the key was not found in this path), it simply returns NULL.
  + if (p->lchild == NULL && p->rchild == NULL){ ... } This handles another **base case**: if the node to be deleted is a **leaf node**. It's safely freed, and NULL is returned to its parent.
  + if (key < p->data){ p->lchild = Delete(p->lchild, key); } These are the **recursive steps** to find the node. The function calls itself on the left or right subtree based on the key's value.
  + else { if (Height(p->lchild) > Height(p->rchild)){ ... } else { ... } } This is the logic for when the node to be deleted **has two children**.
    1. It first checks which subtree, left or right, is **taller**.
    2. If the left subtree is taller, it finds the **In-order Predecessor (q)**, copies its data to the current node p, and then recursively calls Delete to remove the original predecessor node from the left subtree.
    3. If the right subtree is taller (or they are equal), it does the same with the **In-order Successor**. This strategy helps keep the tree more balanced.