BST(C++)

This C++ file is a superb, object-oriented implementation of the **Binary Search Tree (BST)** data structure. By encapsulating all the logic within a BST class, it provides a well-organized, reusable, and safe way to manage a tree.

The class covers the full spectrum of BST operations, consistently offering both **iterative** and **recursive** solutions for fundamental tasks like insertion (iInsert, rInsert) and searching (iSearch, rSearch). This provides a fantastic comparison between the two programming paradigms.

The highlight of the file is its sophisticated and well-designed Delete method. Instead of just picking a simple replacement for a node with two children, it implements a **balancing strategy**. It checks the heights of the left and right subtrees and chooses the replacement (in-order predecessor or successor) from the taller subtree, which helps to mitigate the worst-case scenarios of tree imbalance.

Finally, it showcases a powerful and classic algorithm, createFromPreorder, which demonstrates how to **reconstruct an entire BST** from just its pre-order traversal sequence, using a stack to manage the complex parent-child relationships.

**Iterative Insert**

This method (iInsert) provides a non-recursive way to add a new key to the BST. It's efficient in terms of memory because it avoids the overhead of function calls.

* Node\* t = root; Node\* r;

The algorithm uses two pointers: t is the main traversal pointer that moves down the tree, and r is a "tailing" or "follower" pointer that always points to the parent of t.

* while(t != nullptr){ r = t; ... }

This while loop continues until t becomes nullptr, meaning it has fallen off the tree. At this point, r will be pointing to the exact node that should be the parent of the new key.

* if (key < r->data){ r->lchild = p; } else { r->rchild = p; }

After the loop finds the parent (r), this final check determines whether the new node p should be attached as its left or right child, based on the BST's ordering property.

**In-order Traversal**

This is the classic recursive "Left-Root-Right" traversal. For a Binary Search Tree, an in-order traversal has the special property of visiting the nodes in **ascending sorted order**.

* if (p){ Inorder(p->lchild); ... Inorder(p->rchild); }

The logic is a direct translation of the L-Root-R rule. It makes a recursive call on the left child, then processes the current node (prints its data), and finally makes a recursive call on the right child.

**Iterative and Recursive Search**

This section provides two methods to find a key in the BST. The BST's ordering property allows both methods to be highly efficient (average time of $O(\log n)$).

* Node\* BST::iSearch(int key)

The iterative search uses a simple while loop. At each node, it compares the key to the node's data and decides whether to go left, go right, or stop because a match has been found.

* Node\* BST::rSearch(Node \*p, int key)

The recursive search is a more elegant, "divide and conquer" approach. If the current node isn't a match, it makes a single recursive call on either the left or the right subtree, effectively discarding half of the remaining search space.

**Recursive Insert**

This is the classic, recursive approach to inserting a new key. It's often considered more intuitive and easier to write than the iterative version.

* if (p == nullptr){ ... return t; }

This is the base case for the recursion. When the function reaches a nullptr, it means it has found the empty spot where the new node belongs. It creates the new node t and returns a pointer to it.

* p->lchild = rInsert(p->lchild, key);

This is the recursive step. The function calls itself on either the left or right subtree. The crucial part is assigning the return value of the recursive call back to p->lchild or p->rchild. This is how the new node gets linked into the tree structure as the recursion unwinds.

**Delete Operation**

This is the most complex and powerful method in the class. It recursively finds and deletes a node while carefully preserving the BST property, even including a strategy to keep the tree balanced.

* if (p->lchild == nullptr && p->rchild == nullptr){ ... }

This is the base case for the deletion logic. If the node to be deleted is a leaf node, it is simply removed, and nullptr is returned to its parent to update the link.

* if (Height(p->lchild) > Height(p->rchild)){ ... } else { ... }

This is the logic for deleting a node with two children.

* 1. It first calls the Height function to determine which of the node's subtrees is **taller**.
  2. If the left subtree is taller, it finds the **In-order Predecessor** (InPre), which is the largest value in the left subtree.
  3. If the right subtree is taller (or they are equal), it finds the **In-order Successor** (InSucc).
  4. It **copies the data** from the chosen successor/predecessor to the current node.
  5. Finally, it makes a recursive call to Delete to remove the original successor/predecessor node (which is now a simpler delete case). This balancing strategy helps prevent the tree from becoming too skewed.

**Helper Functions (Height, InPre, InSucc)**

These are essential helper methods used by the Delete function.

* int BST::Height(Node \*p)

This is a standard recursive function that calculates the height of a tree by finding the maximum height of its two subtrees and adding 1.

* Node\* BST::InPre(Node \*p)

This function finds the In-order Predecessor. Starting from the given node p (which is the root of a left subtree), it traverses as far to the right as possible to find the largest element.

* Node\* BST::InSucc(Node \*p)

This function finds the In-order Successor. Starting from the given node p (the root of a right subtree), it traverses as far to the left as possible to find the smallest element.

**Create BST From Pre-order Traversal**

This advanced algorithm reconstructs an entire BST from only its pre-order traversal sequence. It cleverly uses a stack and the properties of a BST to determine the correct parent for each new node.

* stack<Node\*> stk;

A stack is the key tool here. It's used to store the ancestor nodes as the algorithm traverses down a path of left children.

* if (pre[i] < p->data){ ... stk.push(p); p = t; }

The algorithm iterates through the pre array. As long as the next value is less than the current node p's data, it knows it's creating a left child. It pushes the parent p onto the stack for later and moves p down to the new left child t.

* else { ... p = stk.top(); stk.pop(); }

When pre[i] is greater than p's data, it means the leftward path is finished, and this new node must be a right child of one of the ancestors stored on the stack. The while loop (hidden inside the else logic in your code) pops from the stack until it finds the correct parent (the last node popped whose value is still less than pre[i]).