**Binary search Tree**

* No cycle (Function calls in C++ are also tree structure, with the root as the main() function)

Shape

Description automatically generated with medium confidence**Binary Search Tree**

* Each tree has at most 2 children
* Nodes in left subtree have values <= value of node
* Nodes in right subtree have values > value of node
* Min value: left most child
* Max value: right most child

Why this data structure?

* Compare with binary search for a sorted array? (array sorted, pick middle value…)
* For arrays, it's still the same problem: hard to insert values (have to move all values)

**Traversing orders**

1. **Pre-order**: node, left subtree, right subtree (act on the node before visit left, then right)

8, 3, 1, 6, 4, 7, 10, 14, 13

1. **Post-order**: left subtree, right subtree, node

1, 4, 7, 6, 3, 13, 14, 10, 8

1. **In-order**: left subtree, node, right subtree

1, 3, 4, 6, 7, 8, 10, 13, 14

**Text

Description automatically generatedText

Description automatically generatedClass declarations**

Note that TreeNode also has insert, print\_inorder, etc. (because we are treating each node as a root 🡪 **All the implementations are recursive**!

**Tree Class**

**Text

Description automatically generatedText

Description automatically generated**

// for valueExists: add **if (root == NULL)**

// del passes treeNode pointer by reference

**TreeNode Class**

Text

Description automatically generated**Text

Description automatically generatedText

Description automatically generated Constructors**  **Insert Minimum value**

Text

Description automatically generated **In-order traversal**

**Destructors**

Must traverse **post-order** (must delete left and right before deleting itself)

Text

Description automatically generated

Shape

Description automatically generated with medium confidence**TreeNode::del (int v, TreeNode \*& pp)**

* If node (1) has no children, simply delete & set to NULL
* If the node (10) has only right child (14), we can directly set the child (14) to right of parent (8)
* If the node (14) has only left child (13), we also do the same, set child (13) to the left of parent (10)
* If the node has both right and left child (6), (3), (8)
  + Replace the node with the smallest value of the right subtree (A), or with the largest value of the left subtree (B)
  + Text

    Description automatically generatedText

    Description automatically generatedWhen this happens, there will be 2 instances of A or B in the tree 🡪 do a recursive call again for A or B, but this time the case is already covered above (A will not have left child, B will not have right child)

// Find the node to be deleted

// The 4th case is "soft" delete, where we only changed the value on the current node.

|  |
| --- |
| // We are passing in the reference of the poiner which points to current node. That pointer is the attribute of the parent node, so if we want to delete the current node, we changed what "left" of the parent node points to 🡪 pp = NULL; delete this;  // Normally, to change an object, we pass in its reference (act directly on it) or **or pointer to it (act on object by dereferencing).** Here, we want to change what the poiner attribute points to 🡪 Pass in by reference the pointer, or pass in the pointer to this pointer. |