Data Structure

Recitation XII
Topic

- Balancing BST
- Exercise for tree
Balancing a Binary Search Tree

- Extreme case: Linked list, $O(\log N)$ becomes $O(N)$
- Static approach: balance()
- Dynamic approach: AVL tree, single_rotate/double_rotate
Static approach

- Try 1: Inorder + Natural Order
  - Inorder traverse => get a sorted array
  - Use add() on consecutive elements => get a line.
  - Not working

- Try 2: Preorder + Natural Order
  - Get back the same tree we started with.
  - Not working

- Try 3: Postorder + Natural Order
  - Not working
Working one: Inorder + Middle First

balance()

\[
\begin{align*}
n &= \text{tree.reset(INORDER)}; \\
\text{for (int } i=0; \ i<n; \ i++) & \\
\quad \text{array}[i] &= \text{tree.getNext()} \\
\text{tree} &= \text{new BinarySearchTree()} \\
\text{tree.insertTree}(0, \ n-1)
\end{align*}
\]

insertTree(lo, hi)

\[
\begin{align*}
\quad \text{if (hi} &= \text{lo)} \\
\qquad \text{tree.add(array[lo])} \\
\quad \text{if (hi} &= \text{lo+1)} \\
\qquad \text{tree.add(array[lo])} \\
\qquad \text{tree.add(array[hi])} \\
\quad \text{mid} &= (lo+hi)/2 \\
\quad \text{tree.add(mid)} \\
\quad \text{tree.insertTree}(lo, \ mid-1) \\
\quad \text{tree.insertTree(mid+1, hi)}
\end{align*}
\]
Nonlinked Representation of Binary Trees

- Idea: Store the tree elements in the array, level by level, from left to right.

- Approach:
  - Store the root in slot[0].
  - If the node in slot $n$ has a left child, store it in slot $2n + 1$.
  - If the node in slot $n$ has a right child, store it in slot $2n+2$. 
Complete/Full

- A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- A full binary tree is one in which all leaves are at the same depth or same level, and in which every parent has two children.
Priority Queues

- Like a regular queue, where additionally each element has a "priority" associated with it.

- In a priority queue, an element with high priority is served before an element with low priority.

- If two elements have the same priority, they are served according to their order in the queue.
Tree Question 1

- How to do a Level Order Tree Traversal?

- Level order traversal of the above tree is 1 2 3 4 5
- Algorithm: For each node, first the node is visited and then it’s child nodes are put in a FIFO queue.

- printLevelorder(tree)
  1) Create an empty queue q
  2) temp_node = root /*start from root*/
  3) Loop while temp_node is not NULL
     a) print temp_node->data.
     b) Enqueue temp_node’s children (first left then right children) to q
     c) Dequeue a node from q and assign it’s value to temp_node
Tree Question 2

- Given a tree, how to check whether it is a binary search tree or not?
- Simple way: inorder traversal
int isBST(Node node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its values are >= min and <= max. */
int isBSTUtil(Node node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;
    /* false if this node violates the min/max constraint */
    if (node.data < min || node.data > max)
        return 0;
    /* otherwise check the subtrees recursively, tightening the min or max constraint */
    return
        isBSTUtil(node.left, min, node.data) &&
        isBSTUtil(node.right, node.data+1, max);
}
Tree Question 3

- What is Lowest Common Ancestor in a Binary Search Tree of two node?

- Here the common ancestors of 4 and 14, are \{8,20\}, the lowest one is 8.
The main idea of the solution is — While traversing Binary Search Tree from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 is the Lowest or Least Common Ancestor(LCA) of n1 and n2 (where n1 < n2).

So just traverse the BST in pre-order, if you find a node with value in between n1 and n2 then n is the LCA, if it's value is greater than both n1 and n2 then our LCA lies on left side of the node, if it's value is smaller than both n1 and n2 then LCA lies on right side.
int leastCommonAncestor(Node root, int n1, int n2)
{
    /* If we have reached a leaf node then LCA doesn't exist
     * If root->data is equal to any of the inputs then input is
     * not valid. For example 20, 22 in the given figure */
    if(root == NULL || root->data == n1 || root->data == n2)
        return -1;

    /* If any of the input nodes is child of the current node
     * we have reached the LCA. For example, in the above figure
     * if we want to calculate LCA of 12 and 14, recursion should
     * terminate when we reach 8*/
    if((root->right != NULL) &&
        (root->right->data == n1 || root->right->data == n2))
        return root->data;
    if((root->left != NULL) &&
        (root->left->data == n1 || root->left->data == n2))
        return root->data;

    if(root->data > n1 && root->data < n2)
        return root->data;
    if(root->data > n1 && root->data > n2)
        return leastCommonAncestor(root->left, n1, n2);
    if(root->data < n1 && root->data < n2)
        return leastCommonAncestor(root->right, n1, n2);
}