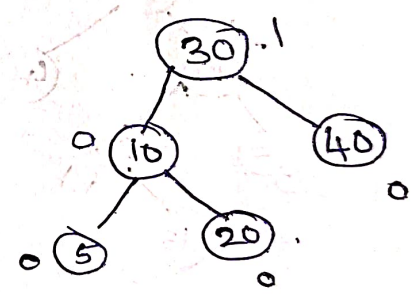
Name: Pathri Vidya Praveen
Roll. No.: CS24BTECH11047

Theory + Coding Assignment 3.

2. Theory Questions:

3. ⇒ We need to give a scenario (specific node heights or shape) where a double rotation is necessary during insertion in an AVL tree.

⇒ Let's construct an AVL tree by inserting keys in the following sequence ⟶ 30, 10, 40, 5, 20. Tree maintains correct balance after insertions.



⇒ BF = balance factor
= height of left subtree − height of right subtree

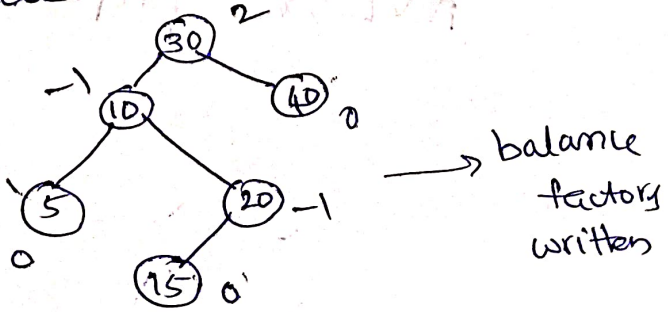
BF ∈ {−1, 0, 1} for balance in AVL tree

⇒ Nodes 5, 20, 40 ⟶ leaves ⟶ height = 0, BF = 0

Node 10 ⟶ height = 1, BF = 0.

Node 30 ⟶ height = 2, BF = 1

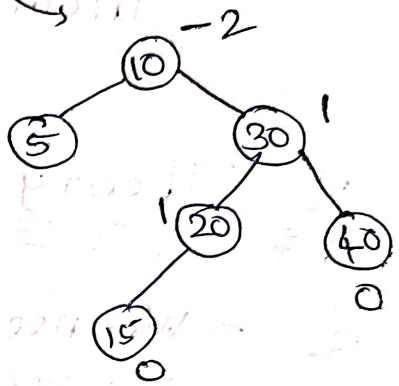
⇒ Now let's insert node 15 here. It becomes like this ⟶



⇒ Now node 30 has balance factor of +2 which results in violation of AVL tree property

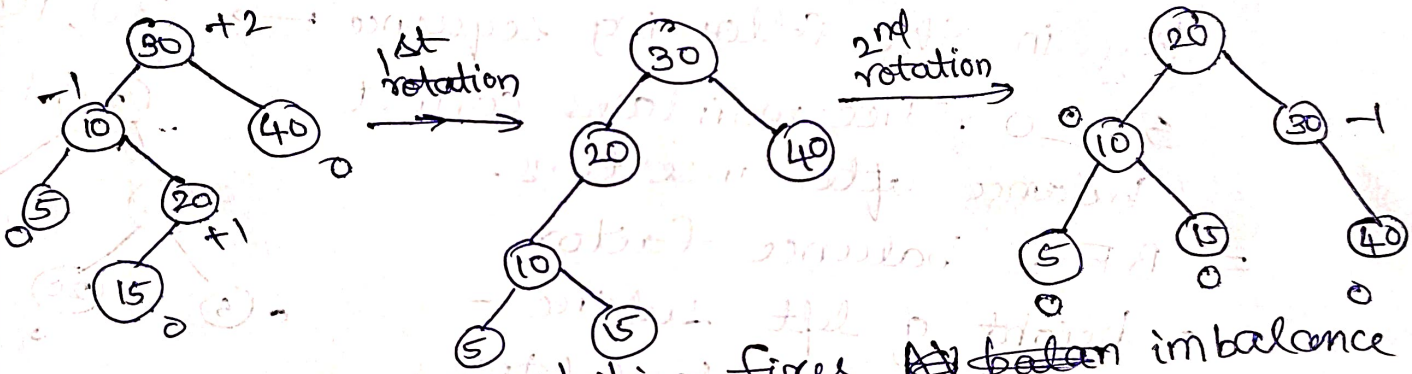
⇒ Now, if we try to fix this imbalance with a single rotation, at node 30, it gives

Now, balance factor of node 10 becomes −2.



Single rotation is failing because it is trying to shift imbalance from node 30 to node 10 but it did not change initially present zig zag pattern.

⇒ In double rotation,



So, here double rotation fixes ~~balance~~ imbalance of AVL tree when insertions give a zig-zag like structure. Single rotations can only fix when imbalance is in a linear path. LR or RL cases of double rotation properly distributes nodes, manages subtrees heights for keeping this AVL tree property of balance.

2. ⇒ Show that longest simple path from node x in red-black tree to a descendant leaf has a length atmost twice that of shortest simple path from node x to descendant leaf.

⇒ A Red-black tree is a BST with following properties.

① Color property: every node is either red or black
② Root node is black
③ Every leaf (NIL node) is black
④ If a node is red, both its children are black
i.e no 2 red nodes are adjacent
⑤ For every node x, all simple paths from x to any descendant leaf contain same no. of black nodes. this number is called black-height of x. this count does not include x itself, even if x is black.

⇒ Let x be any arbitrary node in R-B tree.
⇒ S(x) = no. of edges on shortest simple path from x to descendant leaf.
⇒ L(x) = no. of edges on longest simple path from x to descendant leaf.
⇒ Prove that: $L(x) \leq 2 \cdot S(x)$
⇒ Let b = black height of node x.
⇒ By ⑤, every path from x to leaf contains exactly b black nodes excluding x.
⇒ Consider any path from x to leaf V_k

$x = V_0, V_1, \ldots, V_k \longrightarrow$ leaf

No. of edges = k with exactly b black nodes.

⇒ On any path from x to leaf, there are exactly B black nodes. If a path has no red nodes, then it has b black nodes (excluding x) with no. of edges exactly b. If there is atleast 1 red node, path has more than b edges.

$$S(x) \geq b \quad \text{——— (a)}$$

⇒ We need to find upper bound for $L(x)$ by considering max. no. of edges possible on a path from x to leaf.

Path: $x \rightarrow V_0, V_1 - - , V_k \rightarrow$ black leaf

(B black nodes)

No 2 red nodes are consecutive

Case 1: x is black.

$V_0 = x$	V_1	V_2	V_3	—	—	—	—	V_k
B	R	B	R	—	—	—	—	B

b black nodes, b red nodes → $L(x) \leq 2b$

Case 2: x is red.

V_0	V_1	V_2	V_3	—	—	—	V_k
R	B	R	B	—	—	—	B

b black nodes, $(b-1)$ red nodes → $(2b-1)$ edges.

⇒ $\boxed{L(x) \leq 2b}$ ——— (b)

⇒ By equations (a) and (b)

$$L(x) \leq 2b \leq 2S(x)$$

$$\boxed{L(x) \leq 2S(x)}$$

⇒ For any node x in R-B tree, longest simple path from x to descendant leaf has length atmost twice that of shortest such path.

1. ⇒ Consider a relaxed R-B tree T with a red root. A relaxed R-B tree satisfies all the properties of standard R-B tree except that the root may be red or black.
Properties are:
(a) Every node is either red or black
(b) Every leaf (NIL) is black
(c) If a node is red, both its children are black.
(d) For each node, all simple paths from node to descendant leaves contain same no. of black nodes.

⇒ After coloring root of T black without any other changes, we need to determine if resulting tree is R-B tree.

⇒ In standard R-B tree, above 4 properties (a), (b), (c), (d) are satisfied along with another property that root is black.

(a) ⇒ In original tree T, every node is either red or black, so after changing root from red to black, all nodes remain either red or black and root is the only node whose color changes to black.
⇒ This property satisfied.

(b) ⇒ Leaves (NIL nodes) are unchanged while coloring root black so all leaves are black.
⇒ This property satisfied.

(c) ⇒ Since root was red in T, both its children must be black. After coloring root black, root is no longer red, so this property does not restrict any condition on children of root. Roots children remain black. For all other red nodes in T, this property holds and also no other nodes are changed and these red nodes have black children

⇒ This property satisfied.

(d) ⇒ We need to show that after changing root node color to black, for each node, all simple paths from node to descendant leaves contain same no. of black nodes.

⇒ For root node: say initially in T, all simple paths from root to leaves have same no. of black nodes (say k). Even if we change color of root to black, all paths still have same no. of black nodes (k+1).

⇒ For non root node: Consider node u which is not root, Changing roots color does not affect simple paths from u to descendant leaves because root node is not included. So, all paths from u to leaves have same no. of black nodes since coloring root black does not change other nodes' colors.

⇒ This property satisfied

(e) ⇒ Root is colored black so it satisfies this property of standard R-B tree.

Hence, resulting tree is a red-black (R-B) tree.

1. Coding Problems:

5. Pseudocode:

```
RB-ENUMERATE(x, a, b):
    if x = NULL
        return
    if x.key ≥ a and x.left ≠ NULL
        RB-ENUMERATE(x.left, a, b)
    if x.key ≥ a and x.key ≤ b
        print(x.key)
    if x.key ≤ b and x.right ≠ NULL
        RB-ENUMERATE(x.right, a, b)
    return
```

Explanation:

The above RB-ENUMERATE function is similar to in-order traversal of RB tree rooted at x, outputting all keys k where $a \leq k \leq b$. We are pruning subtrees that cannot be in the range of $[a, b]$.

① If $x.key \geq a$ and left child exists, recursively traverse left subtree

② If $x.key \geq a$ and $x.key \leq b$, output it

③ If $x.key \leq b$ and right child exists, recursively traverse right subtree.

Proof of Correctness:

⇒ In R-B tree which is a BST, for any node y, all keys in left subtree of y are ~~less than~~ $\leq$ y.key and all keys in right subtree of y are ~~greater than~~ $\geq$ y.key

⇒ We are pruning out left subtrees whose keys are $\leq x.key < a$.

⇒ If x·key > b, all keys in right subtree ≥ x·key > b so we prune right subtree also

⇒ Condition $a \leq x.key \leq b$ ensures that we output keys only in required range

⇒ By mathematical induction:

① Base case: For single node, algorithm correctly outputs it if within range

② Inductive step: Assuming correctness for left, right subtrees, algorithm correctly handles current node and combines results from valid subtrees.

Proof of time complexity:

⇒ Each of m keys in range is output exactly once. $O(m)$ time complexity ——— ①

⇒ Algorithm only traverses nodes that are either within [a, b] or on the search paths to boundaries of range. Search paths to find where range [a, b] begins and ends form 2 paths from root to leaves. Height $h = O(\log n)$. Boundary path length = $O(\log n)$ ——— ②

⇒ P_1 = path from root to node containing smallest key ≥ a

P_2 = path from root to node containing largest key ≤ b.

Algorithm visits all nodes in intersection of subtrees rooted at P_1, P_2 and nodes along paths P_1 and P_2 ($O(\log n)$ nodes). No other nodes are visited.

⇒ Total time
= Time for visiting m nodes in range + Time for traversing boundary paths
= $\Theta(m) + O(\log n)$ (By ① and ② equations)

Total time complexity = $\Theta(m + \log n)$

⇒ As $\Omega(m)$ time is necessary to output m keys and $\Omega(\log n)$ time is necessary to locate range boundaries in balanced tree, $\Theta(m + \log n)$ is optimal time complexity. ($O(\log n)$ height guaranteed by RB tree property)