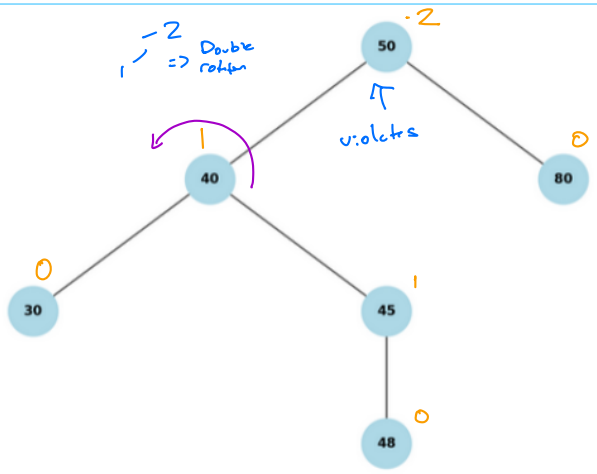
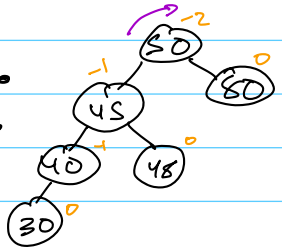
Trevor Swan **CSDS233 ASSIGNMENT 3**

Question 1) AVL Tree Operation



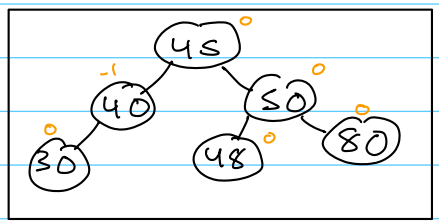
a) Balances in Orange to start

Left Rotation on 40

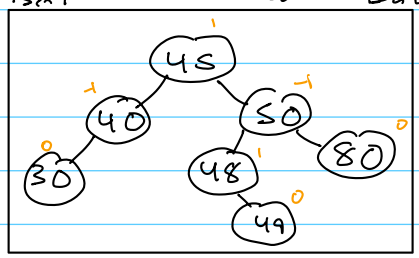


Right Rotation on 50

Guaranteed to be balanced but checked in case anyways

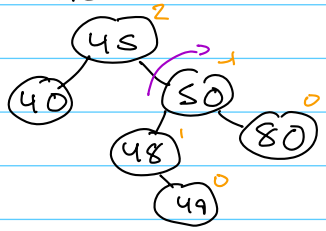


b) Insert 49 → Check Balances

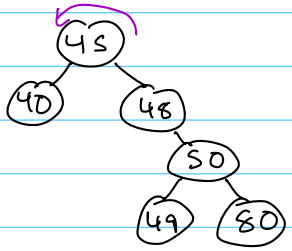


No rotations needed!!

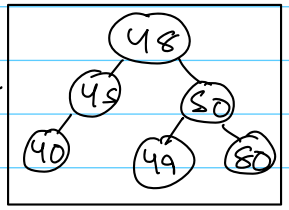
c) Delete 30



Left rotate around 50

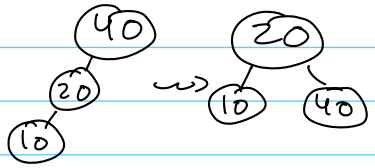


Right rotate around 45

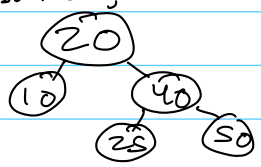


d) Insert 40, 20

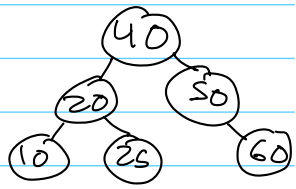
Insert 10, rotate



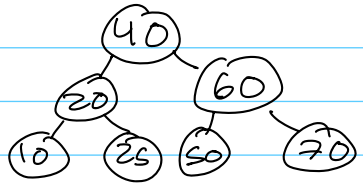
Insert 25, 50



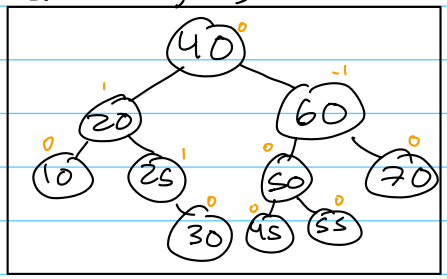
Insert 60, rotate



Insert 70, rotate



Insert 55, 30, 45



It's Balanced!

Question 2) B-Tree

1. In a B-tree of order m, internal nodes must always have exactly m/2 children, regardless of the number of keys in the tree.

False, internal nodes must have between $\lceil m/2 \rceil$ and m children. These internal nodes are not limited to having exactly $m/2$ children, as number of children increases as number of keys grows → up to atmost m children.

2. During the insertion process in a B-tree, if a node becomes full, it is always the root node that splits first.

False, it is not always the root node that is split. The keys are inserted at the leaf nodes. Any node can split once it becomes full. Root nodes may split (when they become full), or if splitting propagates upwards, but it is not always the first to split.

3. In a B-tree, the height of the tree grows logarithmically as the number of keys increases.

True, Operations involve splitting and merging → everything is logarithmic.

4. The time complexity of searching for a key in a B-tree depends on both the height of the tree and the number of keys stored at each node.

True, need to search for node containing said key which could be at max depth (height of tree). Assuming nodes keys are sorted, you need to perform a binary search. $O(\log n)$, n is number of total keys ← determined by height & number per node.

5. In a B-tree, after a deletion operation, the tree always requires a merge operation between adjacent nodes to maintain its properties.

False, merge operations are only required if a node has too few keys (must have at least $\lceil m/2 \rceil$ keys.

Question 3) Tree in General

1. Binary Trees have one criteria and it's that each node has at most 2 children. There is no ordering criteria. Binary Search Trees are a special Binary Tree in which all nodes have at most 2 children and that every node's left subtree is less than the root, and the right subtree is greater than or equal to the root.

2. AVL Tree Height 4

minimum # of nodes: $N(h) = N(h-1) + N(h-2) + 1$

$N(0) = 1$ •

$N(1) = 2$ •

$h=2 \rightarrow N(2) = N(1) + N(0) + 1$ 

$= 2 + 1 + 1 = 4$

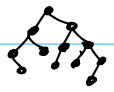
$h=3 \rightarrow N(3) = N(2) + N(1) + 1$ 

$= 4 + 2 + 1 = 7$

$\boxed{N(4) = 12}$

$h=4 \rightarrow N(4) = N(3) + N(2) + 1$

$= 7 + 4 + 1 = 12$

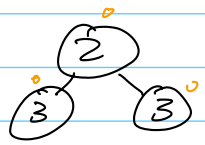


Removal of any node will violate the AVL tree rules, inducing a proper minimum ☺

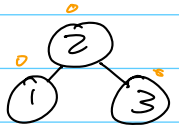
3. A node's balance factor is given by $BF = h(\text{right subtree}) - h(\text{left subtree})$. For an AVL tree, all nodes must have a balance factor of $-1, 0, 1$. All leaf nodes have a balance of 0, and all empty subtrees have a balance of -1. Imbalance is caused when the height of one of the node's subtrees has a height 2 or more greater than the other. If the right subtree is too tall, the tree is right heavy and the nodes balance is $+2$ or greater. Opposite holds for a left-heavy node/subtree.

4. Balanced binary trees abide by the balance factor rule, and every node has a balance of at most ± 1 (unless otherwise specified). Nodes are not ordered like AVL trees which are BST's at their core. They follow the same balance factor rule, but also have all keys less than a given node in the left subtree and all those greater than or equal to in the right subtree.

5.



Balanced, but not AVL as $3 > 2$ but is in left subtree



Balanced, and $1 < 2$, $3 \geq 2$ so AVL is satisfied.

Question 4) Heap

1. True/False

a. In a min-heap, every parent node must have a value less than or equal to the values of its children. (2 points)

True, node values increase down a min-heap.

b. The worst-case time complexity for inserting an element into a heap is O(logn). (2 points)

True, insertion & removal require sifting in the worst case which is a log(n) operation. O(log n) for insertions.

c. The height of a heap with n nodes is always O(logn), regardless of whether it's a min-heap or a max-heap. (2 points)

(maximum nodes) = $2^{(height)+1} - 1$ as heaps are complete binary trees. Height is always $O(\log n)$ space. True.

d. Performing the "heapify" operation on a heap is an O(n) process (2 points)

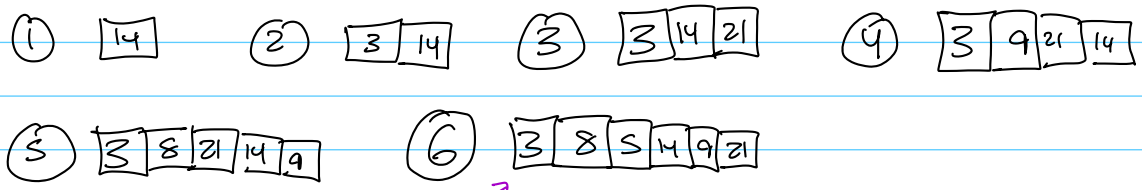
True, proof from class.

e. A min-heap can be used to find the second smallest element in O(1) time (2 points)

True, as long as the min-heap is properly maintained (stays a min-heap & complete) the second smallest is the smallest of the roots two children.

2. [14, 3, 21, 9, 8, 5]

a.



b.