AVL Trees

Jan 16

1. Background:

- AVL Trees were invented by Landis and Adelson-Velskii in 1962.
- An AVL Tree is a self-balancing BST, where the difference between the heights of the left and right subtrees cannot be more than 1 for all nodes.
- Since an AVL Tree is a BST, it has many of the properties a BST has. These properties include:
  - Insertion
  - Deletion
  - Searching

  } Operations
  - Storing values in its internal nodes
  - Has a property relating the values stored in a subtree to the values in the parent node.

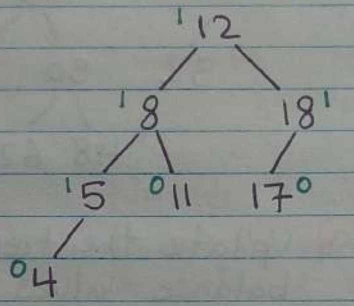
    Note: In an AVL tree, the value of the left subtree is less than the value of the parent node. Furthermore, the value of the right subtree is greater than the value of the parent node.
- The height of an AVL Tree is $O(\log n)$.
- Each internal node has balance property equal to -1, 0, or 1. The purpose of the balance property is to ensure that the height is always a function of $\log(n)$.

To keep track and update the balance property easily, we need to Store the height of the tree at each node.

- Balance Value = height of the left st - height of the right st

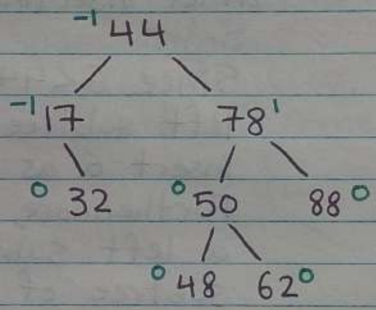
2. Examples of AVL Trees:

1.



Note: The number in green is that node's balance value.

2.

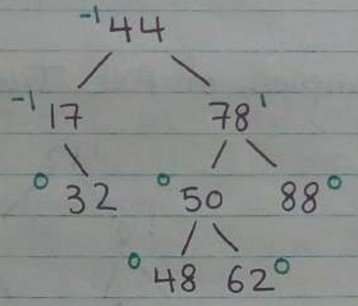


3. Operations:

- There are 3 operations that we will look at. They are insert, delete and search. In a BST, the worst case complexity for the 3 operations is O(n). In an AVL Tree, it is O(log n).

- Searching: Searching in an AVL Tree is the same as a BST.
- Inserting:

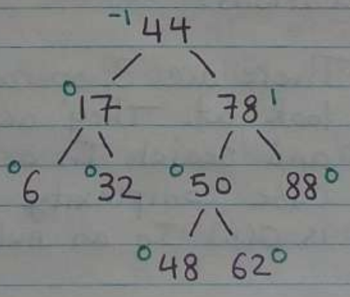
1. Consider the AVL Tree below



a) Update the tree and the balance values of each node, after inserting 6.

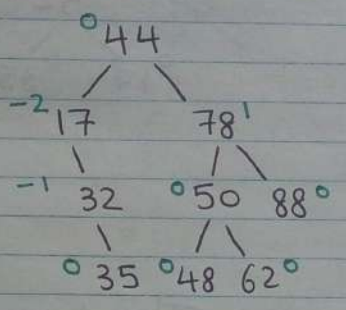
Soln:

Since $6 < 44$, we go to the left subtree. $6 < 17$, so we insert 6 as 17's left child. Furthermore, 17 now has a left subtree and a right subtree of the same height, so 17's balance value is now 0.

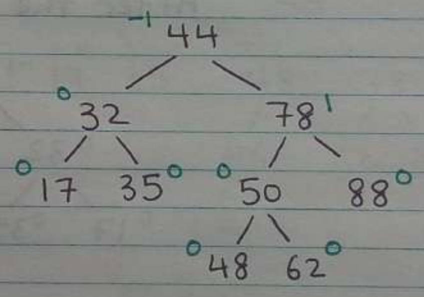


b) Update the tree and the balance values of each node, after inserting 35.

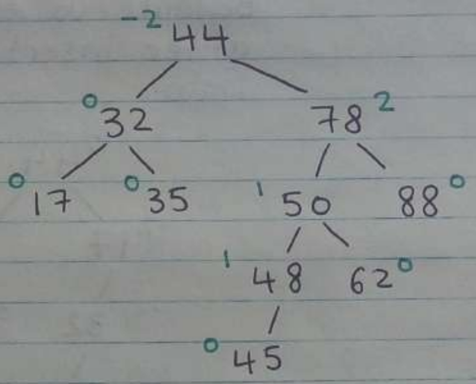
Soln:



Notice that the left subtree is no longer balanced. 17 has a balance value of -2. We can fix this problem with a single rotation. If we rotate ccw, 32 moves up and 17 comes down.

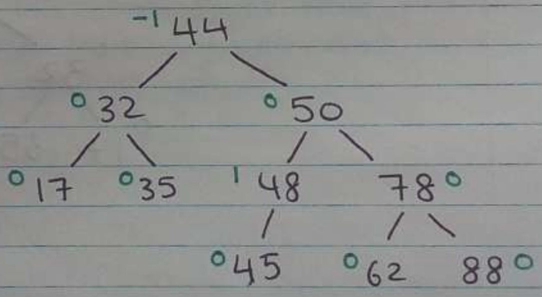


Now, if we insert 45, we get

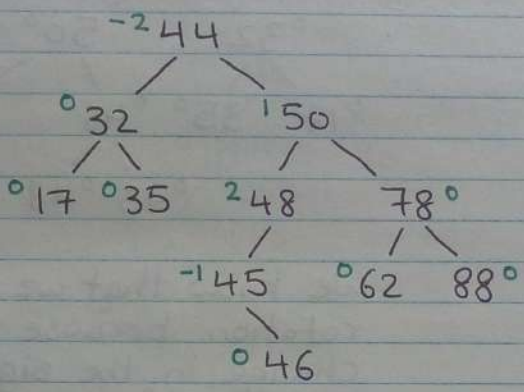


The tree is out of balance, and to fix it, we need to do a single rotation. If we rotate cw about 78, 50 goes up and 78 comes down. Furthermore, 62 becomes the left child of 78.

After the rotation, we get

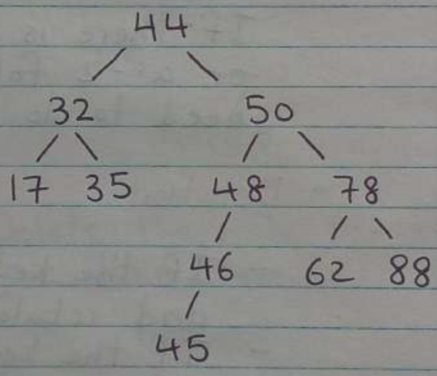


If we insert 46, we get:

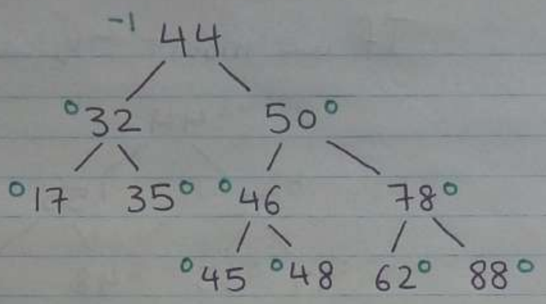


This time, we need a double rotation to balance the tree.

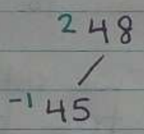
First, we rotate ccw about 45. The tree becomes:



Then, we rotate cw about 48. The tree is shown on the next page.



We know that we need a double rotation because there was a change in the sign of the balance values. If we look at the tree after 46 was inserted, we see

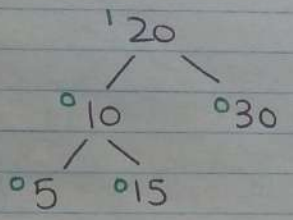


If there is a 2 followed by a -1 or a -2 followed by a 1, we need to do a double rotation.

- Deletion:

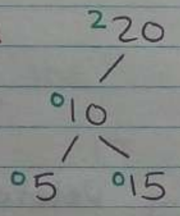
  - If the key is a leaf node, delete and rebalance.
  - If the key is an internal node, replace with predecessor / successor and rebalance.

- E.g. Consider the tree below.
  Delete the node 30.

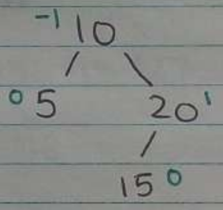


Soln:

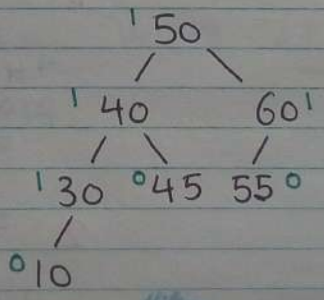
1.



2.

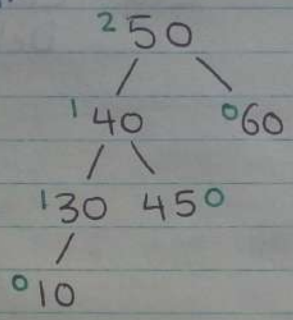


- E.g. Delete the node 55 from the tree below.

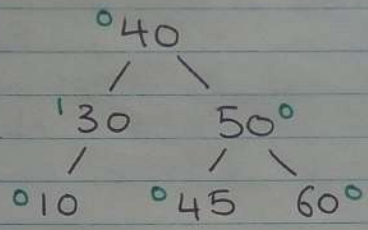


Soln:

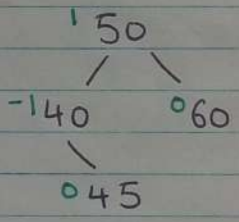
1.



2.

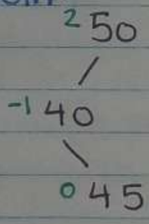


- E.g. Delete the node 60 from the tree.

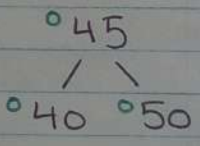


Soln:

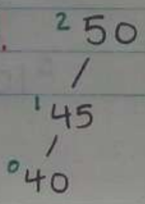
1.



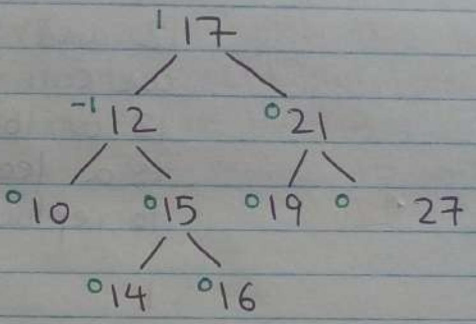
3.



2.



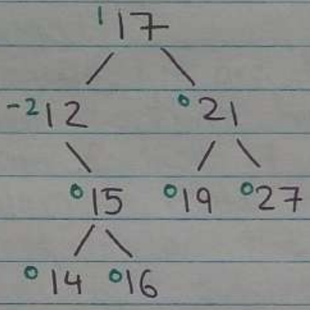
- E.g. Delete the nodes 10, 15, 17, 27, 19 and 12 from the tree.



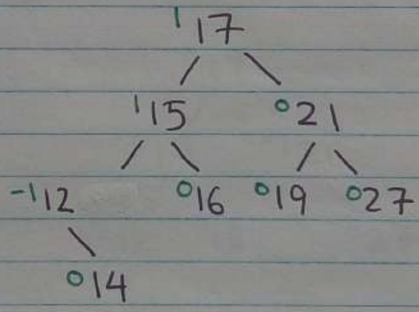
Soln:

1. Deleting 10:

1.



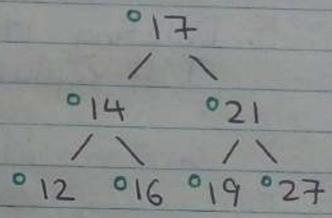
2.



2. Deleting 15:

Notice that 15 has 2 children, 12 and 16. If we delete 15, we can replace it with the largest node in the left subtree or the smallest node in the right subtree. I'll replace 15 with 14.

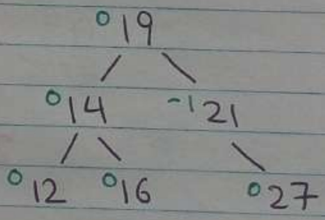
1.



3. Deleting 17:

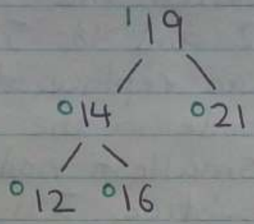
Once again, we are deleting a node that has 2 children. We can replace 17 with either the largest node in the left subtree, 16, or the smallest node in the right subtree, 19. I'll replace 17 with 19.

1.



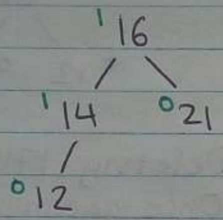
4. Deleting 27:

1.



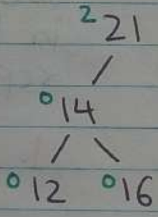
5. Deleting 19:
I'll replace 19 with 16.

1.

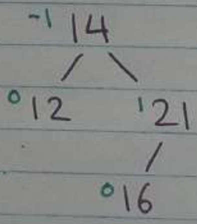


Note: If I replaced 19 with 21,
I'd get:

1.



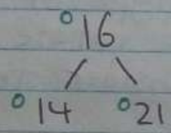
2.



Either way works.

6. Deleting 12:

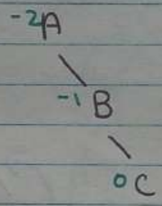
1.



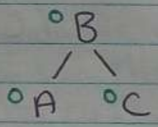
4. Rotations:

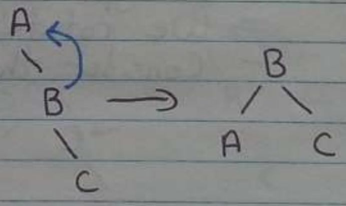
- There are 4 main/basic rotations:

1. Left Rotation:
   - A type of a single rotation.
   - We rotate ccw.
   - Consider the tree below



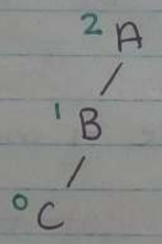
If we rotate ccw about A, then we get:



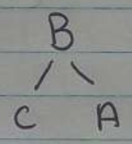


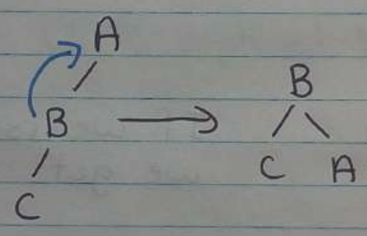
2. Right Rotation:

- A type of single rotation.
- We rotate cw.
- Consider the tree below



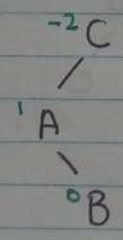
If we rotate cw about A, then we get:



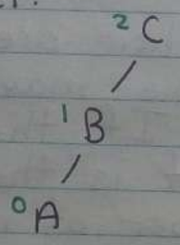


3. Left-Right Rotation:

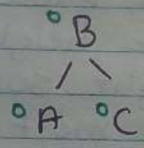
- A type of double rotation.
- We rotate ccw then cw.
- Consider the tree below

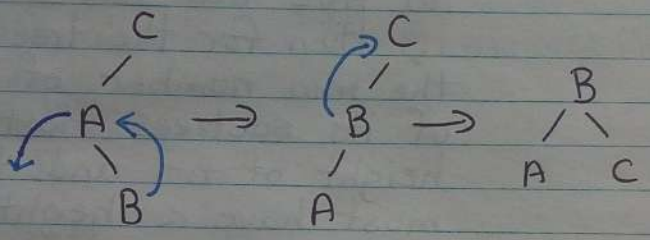


If we rotate ccw about A, then we get:



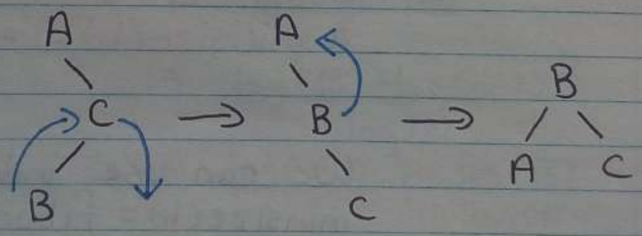
Then, if we rotate cw about C, we get:





4. Right-Left Rotation:

- A type of double rotation.
- We rotate cw then ccw.
- Consider the tree below



4. AVL Tree Height:

- If there are n nodes, what is the max possible height? I.e. If the height is h, then what is the min possible number of nodes?

Soln:
We know that in an AVL Tree, the heights of the left and right subtrees can differ by at most 1. This means that if an AVL Tree has a height of h, then for the tree to have the min number of nodes, one of its subtrees must have a height of h-1 and the other must have a height of h-2.

Let minsize(h) be the min number of nodes for an AVL Tree of height h. Then,

1. minsize(0) = 0
2. minsize(1) = 1
3. minsize(h+2) = 1 + minsize(h+1) + minsize(h)

We can use induction to prove minsize(h) = fibonacci(h+2) - 1 and given the golden ratio, ϕ, which equals to $\frac{\sqrt{5}+1}{2} \approx 1.618$, We can say that

$$\text{minsize}(h) = \frac{\phi^{h+2} - (1-\phi)^{h+2}}{\sqrt{5}} - 1$$

$$= \frac{\phi^{h+2}}{\sqrt{5}} - \frac{(1-\phi)^{h+2}}{\sqrt{5}} - 1$$

$$> \frac{\phi^{h+2}}{\sqrt{5}} - 1 - 1$$

$$> \frac{\phi^{h+2}}{\sqrt{5}} - 2$$

We also know that $\text{minsize}(h) \leq n$

This means:

$$\frac{\phi^{h+2}}{\sqrt{5}} - 2 < n$$

$$\frac{\phi^{h+2}}{\sqrt{5}} < n+2$$

$$\phi^{h+2} < \sqrt{5}(n+2)$$

$$(h+2)(\log \phi) < \log(\sqrt{5}(n+2))$$

$$h+2 < \frac{\log(\sqrt{5}(n+2))}{\log(\phi)}$$

$$< \frac{\log(\sqrt{5}) + \log(n+2)}{\log(\phi)}$$

$$< \frac{\log(n+2)}{\log(\phi)} + \frac{\log(\sqrt{5})}{\log(\phi)}$$

$$h < \frac{\log(n+2)}{\log(\phi)} + \frac{\log(\sqrt{5})}{\log(\phi)} - 2$$

$$= 1.44 \log(n+2) - 2$$

$$\in O(\log n)$$

5. Union, Intersection and Difference of 2 AVL Trees

1. Divide and Conquer: This strategy splits the inputs into smaller pieces, apply the algo and conquers the problem by tying together the return values to get the answer.

2. Operations:
For each algo $A(T,V)$, where T and V are 2 AVL Trees, we will assume that T is taller and let k be the root of V. Then,
   a) Split T into $T_{<k}$ and $T_{>k}$
      Let V_L and V_R be the subtrees rooted at k's left and right children.
   b) Compute $L \leftarrow A(T_{<k}, L)$ and $R \leftarrow A(T_{>k}, R)$. This is the divide part.
   c) Merge L and R back together, and depending on if it's union, intersect or difference, we may also merge it with k to get the required AVL Tree. This is the conqueror part.

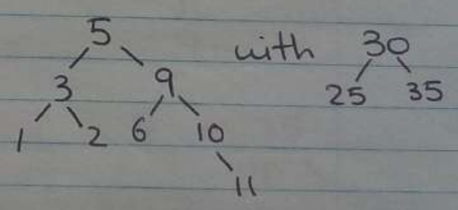
3. Merging:

a) Implementation Algo

- Assume that there are 2 AVL Trees, T and V, and assume that keys in T are less than keys in V.
- Let k be a value that is greater than or equal to T's largest key and smaller than V's root.
- There are 3 cases:

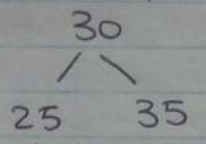
1. $h(T) > h(V) + 1$
   Here, you go down T's right-most path until you reach a height of $h(V) + 2$. Then, insert k as that node's right child. The previous/old right subtree of that node is now k's left child and V is k's right child. Finally, rebalance if needed.

   E.g. Merge



Soln:

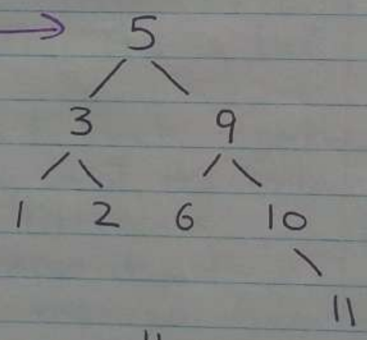
In this case, V is the tree



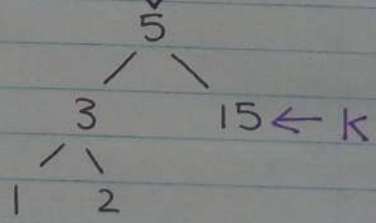
and it has a height of 2. That means we have to go down the other tree's right-most path until we reach a height of 4. Then, we'll insert k as its right child.

Let k = 15

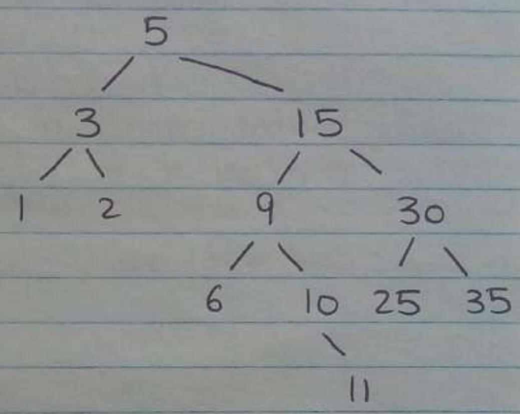
Has a height of 4



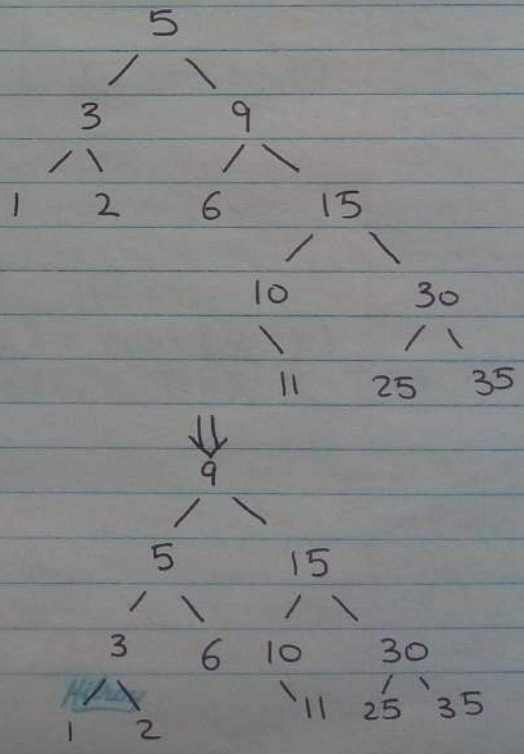
⇓



After you insert k, the right subtree of 5 becomes the left subtree of k and V becomes the right subtree of k.



Lastly, rebalance the tree.



2. $h(V) > h(T)+1$

This is very similar to the first case. Start at V's root and go down its leftmost path until you reach a height of $h(T)+2$. Then, insert k as that node's left child. T becomes k's left child and the old left subtree of the node becomes k's right subtree. Finally, rebalance.

3. The heights of T and k are within 1

Let k be the root of a new AVL Tree. T becomes k's left subtree and V becomes k's right subtree.

4 Splitting Implementation Algo

- Let T be an AVL Tree.
- Let k be the value which we split T by.
- Let L be r's left child.
- Let R be r's right child.
- Let r be T's root.
- Let b be a boolean value, whose value depends on if k is in T.

Consider this pseudo code for splitting an AVL Tree:

```
split (T, k)
  r = root(T)
  if r == k:
     return(L, True, R)
  elif r == NULL:
     return (NULL, False, Null)
  elif r < k:
    If r<k, then we go down T's
    right path. Furthermore, we add
    r and r's left child to T<k. We
    recursively do this on r's right child.
    (T<k, b, T>k) = split (R, k)
    return (merge (L, r, T<k), b, T>k)
  else:
    In this case, r>k. We need to
    go down the left path. Further-
    more, we add r and r's right
    child to T>k. We recursively do
    this on r's left child.
```

Hilroy

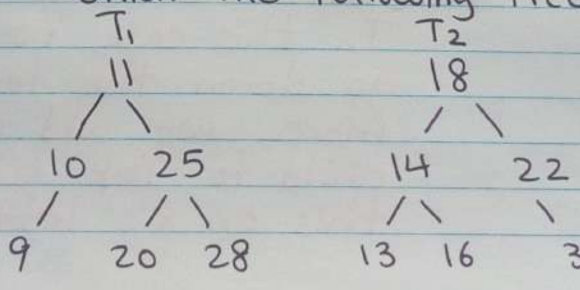
$(T_{<k}, b, T_{>k}) = split(L, k)$
return $(T_{<k}, b, merge(T_{>k}, r, R))$

Note: b is not used for splitting, but is used for intersection.

5. Union of AVL Trees:
   - Given two AVL Trees T and V, return an AVL Tree with all the keys in T and V.
   - Consider this pseudo code:

```
Union(T,V)
  if T == null:
     return V
  elif V == null:
     return T
  else:
     k = root(V)
     (T<k, has_k, T>k) = split(T,k)
     VL = k.left
     VR = k.right
     L = Union(T<k, VL)
     R = Union(T>k, VR)
     return merge(L, k, R)
```

   - Union the following trees:



Soln:

1. $K = \text{root}(T_2)$
   $= 18$

Find $\text{split}(T_1, 18)$:
Since the root of T_1 is 11, we go down T_1's right path. We hit 25, which is larger than 18, so we go down 25's left path. We hit 20, which is greater than 18. We stop at this point.

$T_1 < 18 = \text{merge}(((9), 10), 11, \text{NULL})$
$= ((9), 10, (11))$

10
/ \
9 11

$T_1 > 18 = \text{merge}(20, 25, 28)$
$= ((20), 25, (28))$

25
/ \
20 28

2. Now, we recursively union.
$T_L = \text{Union}(T_1 < 18, T_{2L})$

$T_1 < 18$
10
/ \
9 11

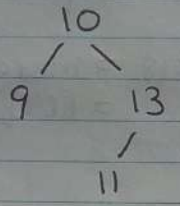
T_{2L}
14
/ \
13 16

We have to find split($T_{1<18}$, 14).
This will help us find $T_{1<14}$ and $T_{1,14<x<18}$.

Since all the values in $T_{1<18}$ are less than 14, nothing happens.

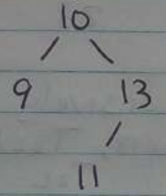
Then, we union $T_{1<18}$ with 13 to get $T_{1<14}$.

$T_{1<14}$

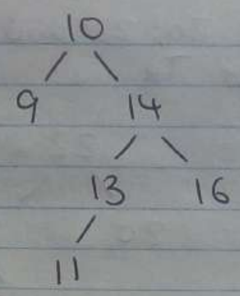


Since no values in $T_{1<18}$ are greater than 14, and only 16 is greater than 14 from T_{2L},
the union of $T_{1,14<x<18}$ with 16 is just 16.

Then, we get T_L by merging with 14 and 16.

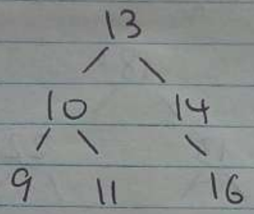


$T_L =$



and after a double rotation, we get

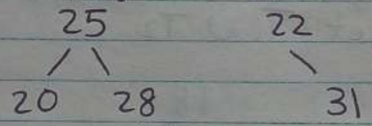
$T_L =$



After we find T_L, we find T_R.

$T_R = \text{Union}(T_{1>18}, T_{2R})$

$T_{1>18}$ $\quad\quad$ T_{2R}



The root of T_{2R} is 22, so we split on 22.

$(T_{1,\,18<x<22},\ b,\ T_{1>22}) = \text{split}(T_{1>18}, 22)$

$T_{1,\,18<x<22} = 20$

$T_{1>22} = (\text{NULL}, 25, 28)$

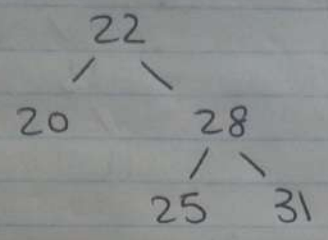
$\quad = (25(28))$

$\text{Union}(20, \text{left child of } 22) = 20$

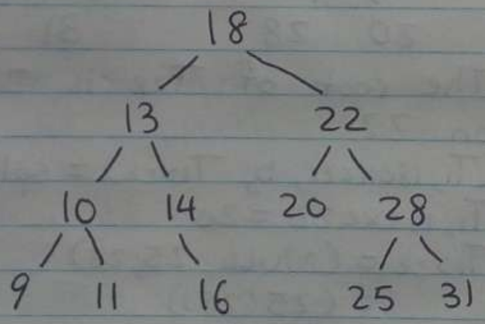
$\text{Union}((25(28), 31) = ((25)\,28\,(31))$

Merging 20, 22 and ((25) 28 (31)), we get T_R.

$T_R =$



Merging T_L with 18 and T_R, we can get $T_1 \cup T_2$.



$T_1 \cup T_2$

6. Intersection of 2 AVL Trees

- Given T and V, we want an AVL Tree containing the keys in both T and V.

- Pseudo Code:

```
intersection(T,v)
  if T==null or V==null:
     return null
  else:
     k=root(v)
     (T<k, has_k, T>k)=split(T,k)
     VL= k.left
     VR= k.right
     L= intersect(T<k, VL)
     R= intersect(T>k, VR)
     if has_k:
        return merge(L, k, R)
     else:
        return merge(L, null, R)
```

Note: This is where b from split is used. If b is true, that means both T and V has it, so it is returned. Otherwise, only V has it, so it's not returned.

7. Difference of 2 AVL Trees

- Given T and V, we want an AVL Tree containing the keys in T and not in V.

- Pseudo Code:

```
diff(T,V)
    if T==null or V==null:
        return T
    else:
        k=root(V)
        (T<k, has_k, T>k) = split(T,K)
        VL = k.left
        VR = k.right
        L = diff(T<k, VL)
        R = diff(T>k, VR)
        return merge(L, null, R)
```

Note: The algorithims for union, intersection and difference are very similar.

6. Supplemental Notes:

- For a node, if $|height(R) - height(L)| \leq 1$ then we say that it's AVL balanced.
- A tree is AVL balanced if every node is AVL balanced.
- After inserting or deleting a node, go back the way the node came down to see if you have to rebalance the tree.

- If we delete a node that has 2 children, in this course, we will replace the deleted node by its successor. I.e. We will go down the left-most path of the node's right subtree to get the smallest node in that subtree and replace the deleted node with that node.

E.g.

```
        30
       /  \
    10      40
   /  \    /  \
  5   15  35   50
```

If we delete 30, we replace it with the smallest node in its right subtree, 35.

Note: Previously, I said that you can replace a node with either its predecessor or successor. Ignore that. In this course, we will always replace a node with its successor, if it exists.