CSCB63
Week 2 - AVL trees

Purva R. Gawde

Topics

- AVL Trees
  - Deletion
  - Height analysis
- Augmenting AVL trees
  - Rank
  - Select

AVL Insert - Summary

- First, insert like a BST
- If still balanced, return.
- Else: (need rebalancing)
  - Left Heavy:
    - No bend: right rotation
    - Bend: Left + right rotation
  - Right Heavy:
    - No bend: left rotation
    - Bend: Right + left rotation

Updating balance factors

- Just update accordingly as rotations happen.
- And nobody outside the picture needs to be updated, because the height is the same as before and nobody above would notice a difference.
- So, only need $O(1)$ time for updating BFs.
- **Note:** this nice property is only for Insert. Delete will be different.

Running time of AVL insert

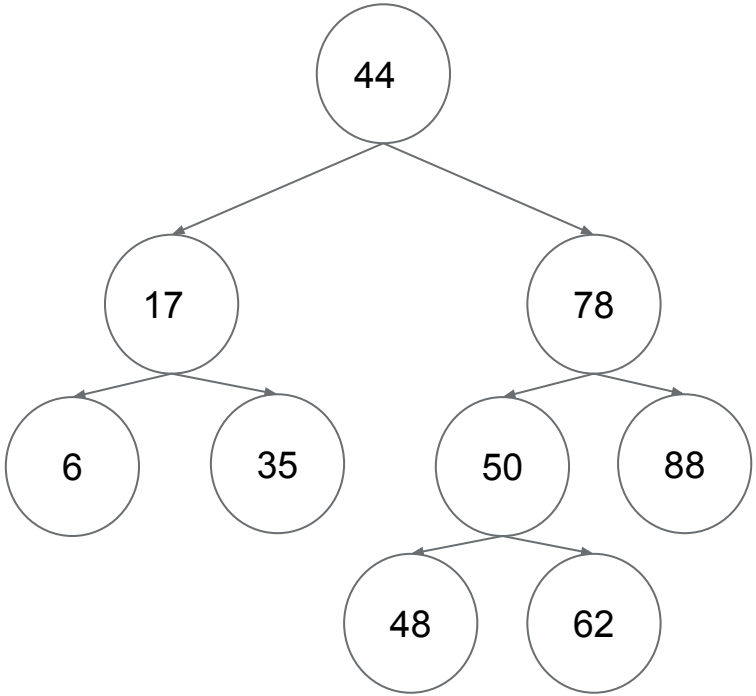
- Just Tree-Insert plus some constant time for rotations and BF updating.
- Overall, worst case O(h) since it's balanced, O(log n)

AVL trees Delete

Delete

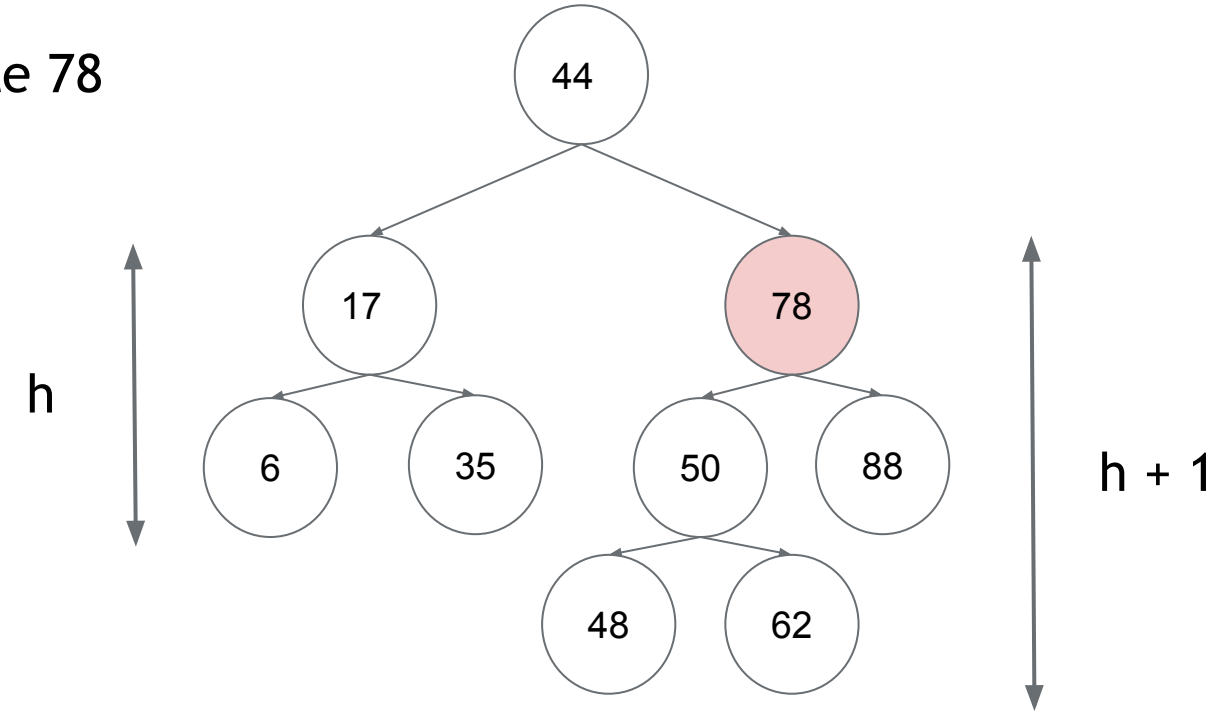
- If the key is a leaf node, delete and rebalance
- If the key is an internal node, replace with predecessor/successor and rebalance
- Process:
- First do a normal BST Tree Delete
  - The deletion may cause changes of subtree heights, and may cause certain nodes to *lose balance factors* within threshold
  - Then rebalance by single or double rotations, similar to what we did for AVL Insert.
  - Then update balance factors of affected nodes

Example

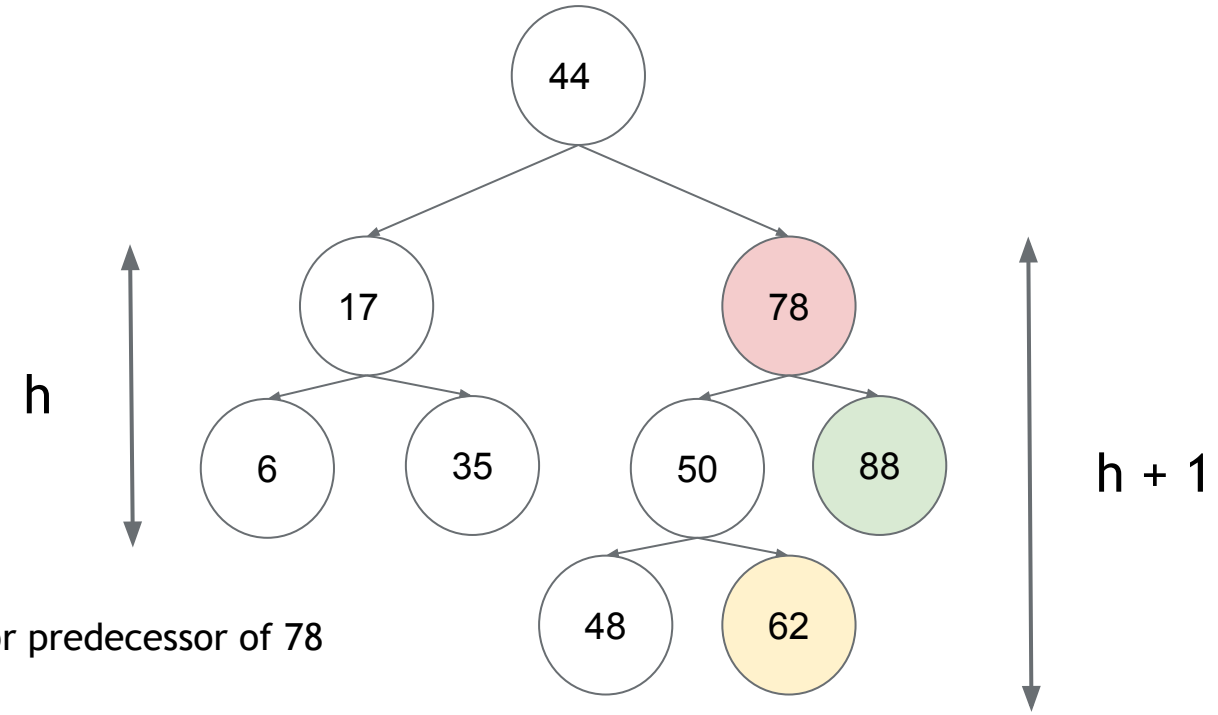


Example

Delete 78



Example



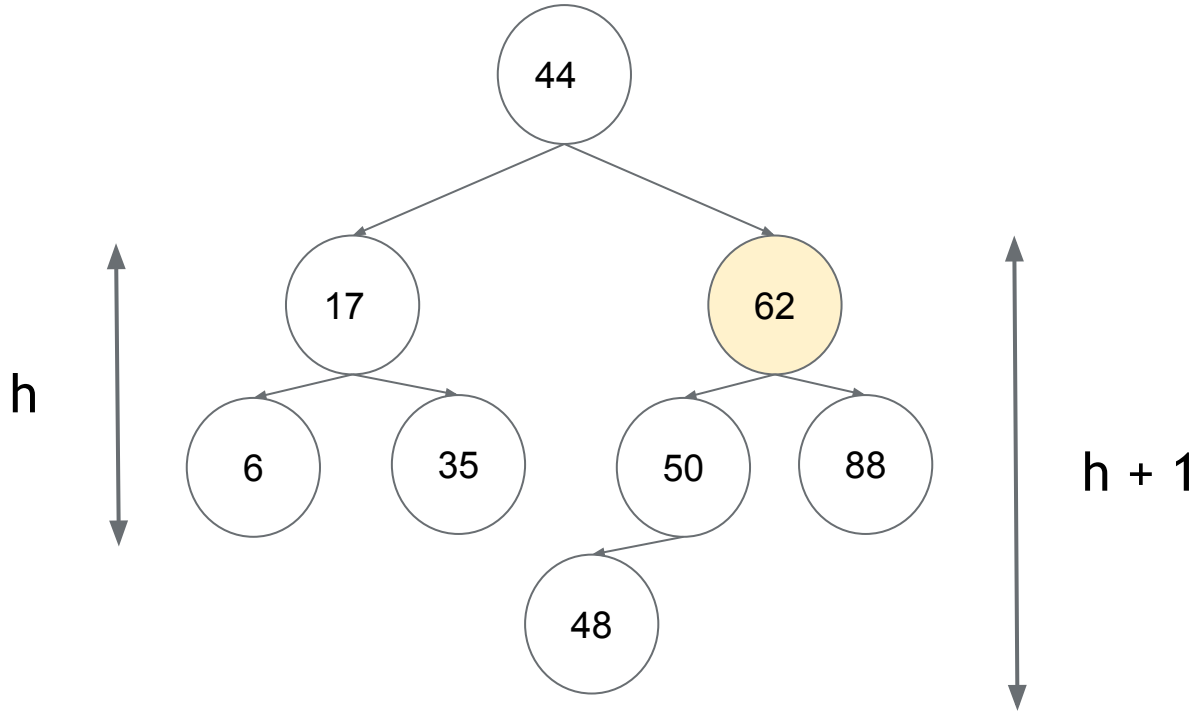
Replace it with successor or predecessor of 78

Successor: 88

Predecessor: 62

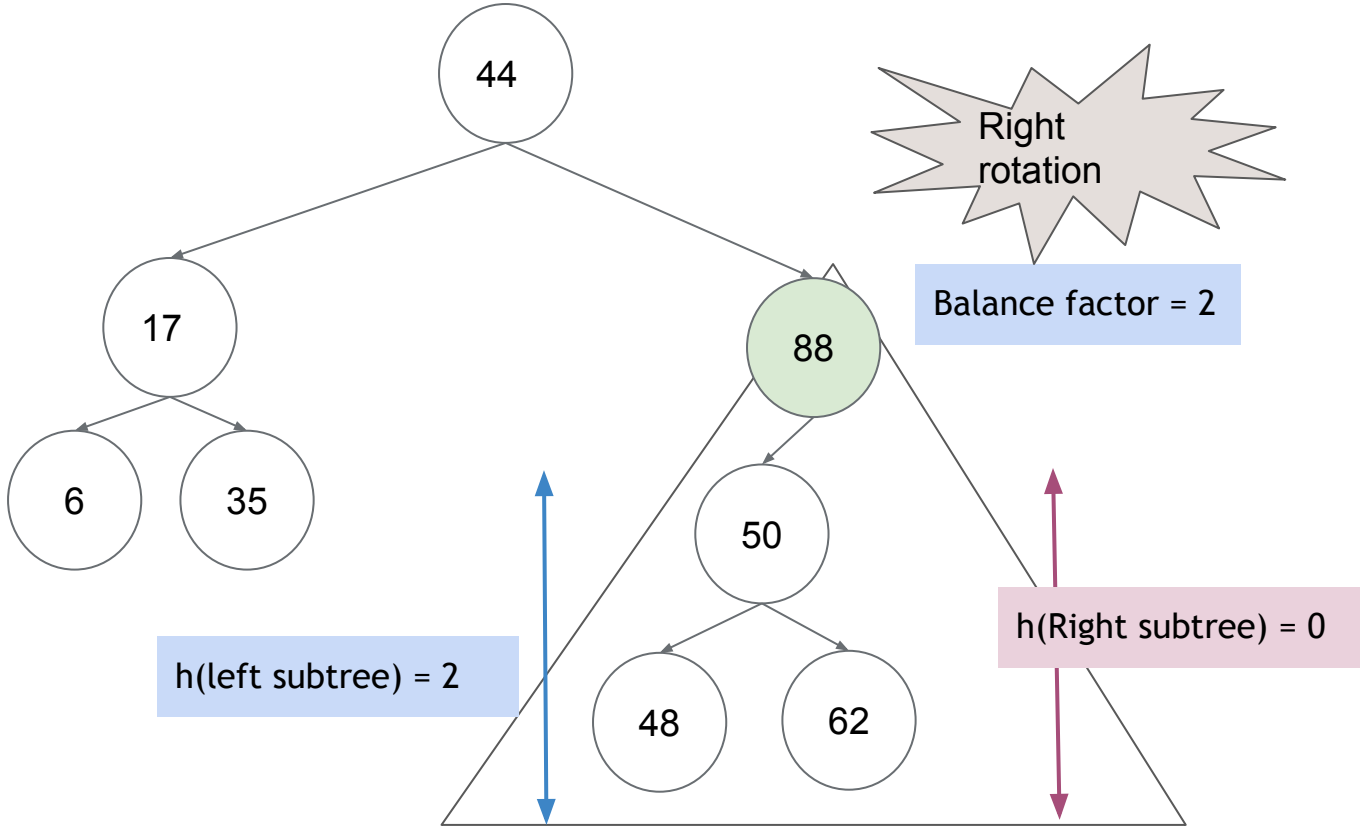
Example

Predecessor: 62

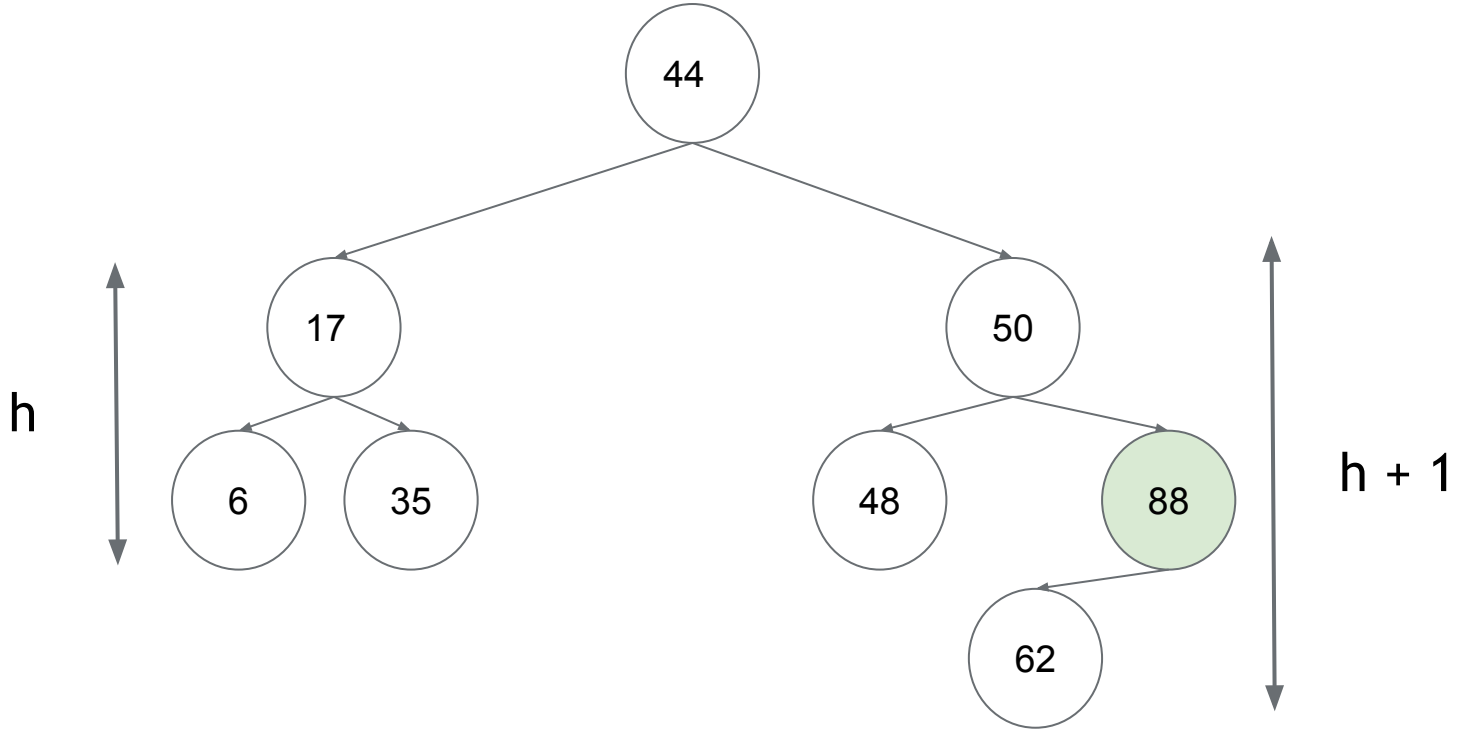


Example

Successor: 88



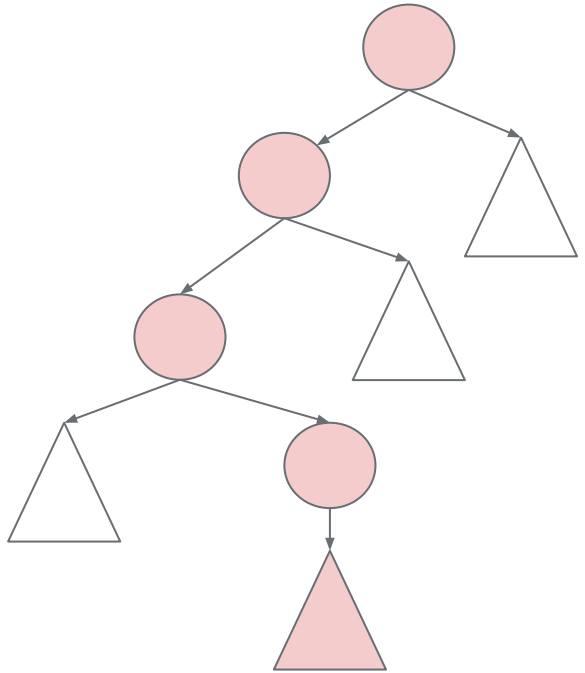
Example



Balanced Tree

Updating the balance factors

- Since the height of the "whole subtree" may change, then the BFs of some nodes outside the "whole subtree" need to be updated.
- Which nodes?
  - All ancestors of the subtree
- How many of them?
  - O(log n)
- Updating BFs take O(log n) time.



AVL trees Complexity

COMPLEXITY

- Since the tree is balanced the height of an *AVL* tree is *O(log n)* (we will prove this shortly).
- This means *insert*, *delete* and *search* are all *O(log n)*.
- Searching for the location to *insert* or *delete*, takes *O(log n)*.
- *Rebalancing* takes at most *O(log n)*.

AVL TREE HEIGHT

- If there are *n* nodes, what is the *maximum* possible *height h*?
- Q. What is *another* way to say this in terms of the number of nodes *n*?
- A. If the *height* is *h*, what is the *minimum* possible number of *nodes*?
- Let *minsize(h)* be the minimum number of nodes for an AVL tree of *height h*.

AVL TREE HEIGHT

- Let *minsize(h)* be the minimum number of nodes for an AVL tree of *height h*.

Then, we can say:

minsize(0) = 0

minsize(1) = 1

minsize(h+2) = 1+ minsize(h+1) + minsize(h)

Q. Does this remind you of anything?

A. Fibonacci...

AVL TREE HEIGHT

minsize(h+2) = 1+ minsize(h+1) + minsize(h)

and *Fibonacci*:

Can prove by *induction*:

minsize(h) = fib(h+2)−1

and given the golden ratio Φ = (√ 5+1)/2 = 1.618... that

.

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

AVL TREE HEIGHT

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

$$n \geq minsize(h) > \frac{\phi^{h+2}}{\sqrt{5}} - 1 - 1$$

Solving for h :

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

$$h < \frac{\lg(n+2)}{\lg\phi} + \frac{\sqrt{5}}{\lg\phi} + 2 \quad = 1.44\lg(n+2) + \text{constant}$$

$$\in O(\lg n)$$

Augmenting AVL trees

Reflect on AVL tree

- We “augmented” BST by storing additional information (the balance factor) at each node.
- The additional information enabled us to do additional cool things with the BST (keep the tree balanced).
- And we can maintain this additional information efficiently in modifying operations
  - (within O(log n) time, without affecting the running time of Insert or Delete).

Augmentation

Augmentation is an important methodology for data structure and algorithm design.

It's widely used in practice, because

- On one hand, textbook data structures rarely satisfy what's needed for solving real interesting problems.
- One the other hand, people also rarely need to invent something completely new.
- Augmenting known data structures to serve specific needs is the sensible middle-ground.

Augmentation: General Procedure

1. Choose data structure to augment

2. Determine additional information

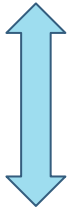
3. Check additional information can be maintained, during each original operation, hopefully efficiently.

4. Implement new operations

Example: Ordered Set

An ADT with the following operations

- Search(S, k) in O(log n)
- Insert(S, x) in O(log n)
- Delete(S, x) in O(log n)
- Rank(k): return the rank of key k
- Select(r): return the key with rank r



AVL tree would work

E.g., S = { 27, 56, 30, 3, 15 }

Rank(15) = 2 because 15 is the second smallest key

Select(4) = 30 because 30 is the 4th smallest key

Augmenting AVL trees

- AVL trees by itself are not very useful
- To support more useful queries we need more structure

An augmented data structure is simply an existing data structure modified to store additional information and/or perform additional operations

- Augmenting AVL trees:

  Rank

  Select

Example

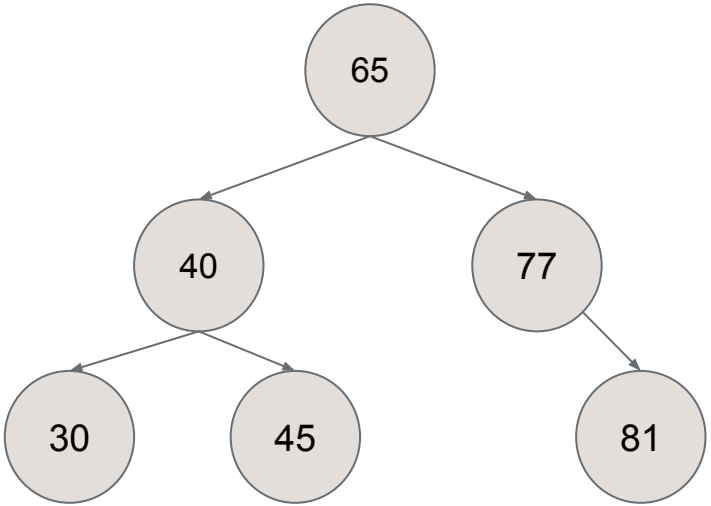
- We want a *data structure* that will allow us to do the standard set of operations(INSERT, DELETE, SEARCH) and:
  - Answer two types of "rank" queries on set of values
  - `RANK(k)`: Given a *key k*, what is its "rank" i.e., its *position* among the elements in the data structure?
  - `SELECT(r)`: Given a *rank r*, which key has this rank?
  - For example, if our set of values is 3, 15, 27, 30, 56, then
  - `RANK(15)` = 2 and
  - `SELECT(4)` = 30.
  - We will look at different ways to do this

Use AVL trees without modification

Rank and Select queries

- Simply do an *in-order* traversal of the tree, keeping track of the number of nodes visited, until the desired rank and key is reached

Q. what will be the *time* for a *query*?



Use AVL trees without modification

Rank and Select queries

- Simply do an *in-order* traversal of the tree, keeping track of the number of nodes visited, until the desired rank and key is reached

Q. what will be the *time* for a *query*?

A. At worst $\Theta(n)$ because we may have to visit every node if our rank = n

Q. Will the other operations (SEARCH/INSERT/DELETE) take any longer?

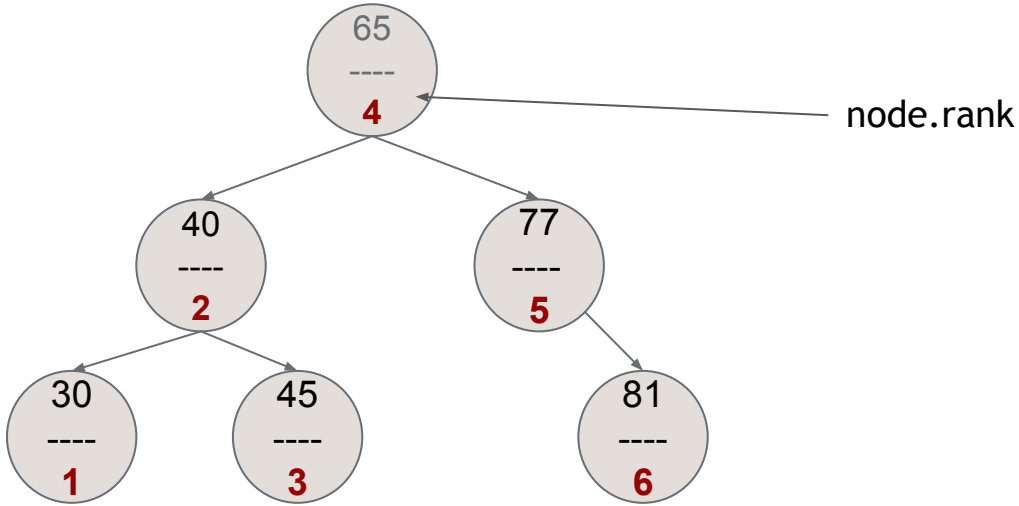
A. No

Q. What is the problem? Could we do *better*?

A. Very inefficient for *rank* and *select*

Augmenting the AVL tree - Take I

Augment each node with an additional field 'rank[x]' that stores x's rank in the tree



Augmenting the AVL tree - Take I

Augment each node with an additional field ‘rank[x]’ that stores x’s rank in the tree

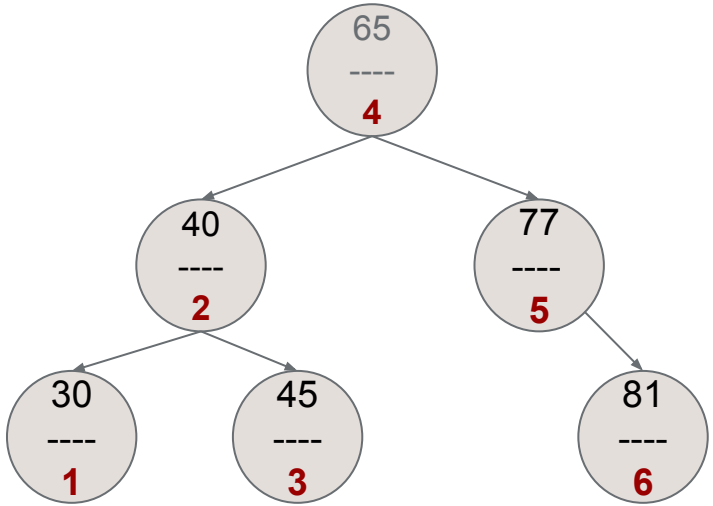
Q. What will be the time for a query?

For RANK(k), the same as SEARCH, or $\Theta(\log n)$

For SELECT(r), we can search on the rank field, so $\Theta(\log n)$

Augmenting the AVL tree - Take I

Q. Will the other operations(INSERT/SEARCH/DELETE) take any longer?



At what cost?

Q. Will the other operations (INSERT/SEARCH/DELETE) take any longer?

A. INSERT will now require the update of the rank field.

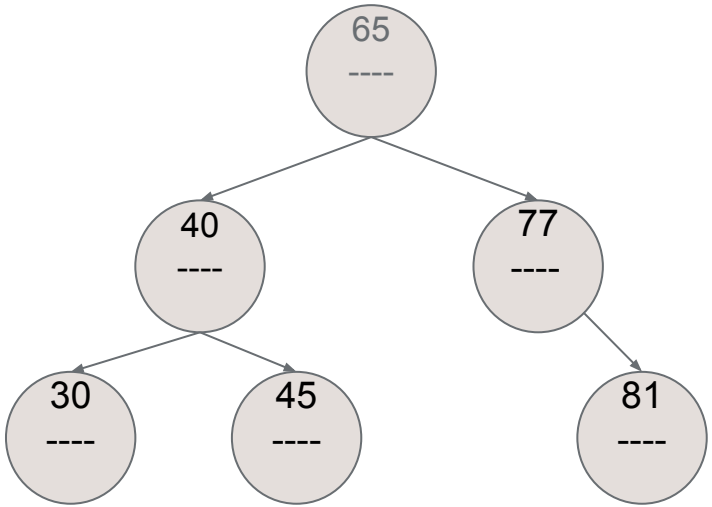
Problem:

We may need to update *all the other nodes'* rank field, so at worst, $\Theta(n)$

So, approach is efficient for RANK and SELECT but at the cost of increasing INSERT and DELETE complexity

Augmenting the AVL tree - Take II

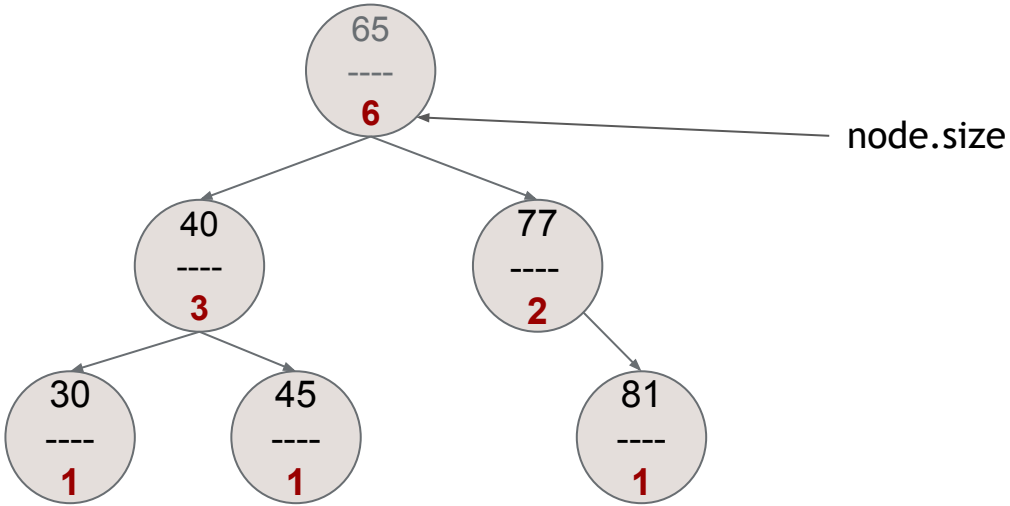
Q. How can we *augment the nodes* of AVL trees so that we can perform all our queries efficiently?



Augmenting the AVL tree - Take II

Q. How can we *augment the nodes* of AVL trees so that we can perform all our queries efficiently?

Q. What *property of subtrees* could help us with questions about *rank*?



Augmenting the AVL tree - Take II

Q. How can we *augment the nodes* of AVL trees so that we can perform all our queries efficiently?

Q. What *property of subtrees* could help us with questions about *rank*?

A. size of subtrees

Data Structure Augmentation

Methodology:

- Choose an underlying data structure (AVL trees).
- Determine additional information to be stored in the data structure (subtree sizes).
- Verify that this information can be maintained for modifying operations (INSERT, DELETE— don't forget rotations).
- Develop new dynamic-set operations that use the information (SELECT and RANK).
- These steps are guidelines, not rigid rule