Augmenting AVL trees

- We want to augment AVL trees to support rank/select operations in $O(\log n)$ time?
  - $\text{rank}(x)$ reports the index of key $x$ in the sorted array of keys
  - $\text{select}(i)$ returns the key with index $i$ in the sorted array of keys
- At each node, store the size (no. of nodes) of the subtree rooted at that node.
Selection in Augmented AVL trees

- Selection on an AVL tree augmented with size data is similar to quickselect, where the root acts as a pivot.

- Select(i): compare i with the rank of the root r (size of left subarray).
  - If equal, return the root r
  - if \( i < \text{rank}(\text{root}) \), recursively find the same index \( i \) in the left subtree
  - if \( i > \text{rank}(\text{root}) \), recursively find index \( i - \text{rank}(\text{root}) - 1 \) in the right subtree

E.g., select(5,12) \( \rightarrow \) select(5,7) \( \rightarrow \) select(2,9) \( \rightarrow \) select(0,11) \( \rightarrow \) 11 is returned
Augmenting AVL trees

- To find \( \text{rank}(x) \) on an AVL tree augmented, search for \( k \).
- On the path from the root to \( x \), sum up sizes of all left sub trees
  - When searching for \( x \), when you recurs on the right subtree, add up the size of the left subtree plus one (for the current node).
  - When the node was found, add up the size of its left subtree to the computed rank.

\[
\text{rank}(16,20) \xrightarrow{\text{left}} \text{rank}(16,12) \quad \text{res} + = 12 + 1 \quad \xrightarrow{\text{right}} \quad \text{rank}(16,17) \xrightarrow{\text{left}} \\
\text{rank}(16,14) \quad \text{res} + = 1 + 1 \quad \xrightarrow{\text{right}} \quad \text{rank}(16,16) \quad \text{res} + = 1 \quad \text{rank}(25,20) \\
\text{res} + = 20 + 1 \quad \xrightarrow{\text{right}} \quad \text{rank}(25,28) \xrightarrow{\text{left}} \quad \text{rank}(25,25) \quad \text{res} + = 4.
\]
Augmenting AVL trees

\[
\text{rank}(\text{searchKey})
\]

- return \( \text{rank}(\text{searchKey}, \text{root}) \)

\[
\text{rank}(\text{searchKey}, \text{node})
\]

- If \( \text{node} = \emptyset \) then return \(-\infty\) (node doesn't exist).
- If \( \text{searchKey} = \text{node.key} \) then return \( \text{node.left.size} \).
- If \( \text{searchKey} < \text{node.key} \) then return \( \text{rank}(\text{searchKey}, \text{node.left}) \).
- If \( \text{searchKey} > \text{node.key} \) then return \( 1 + \text{node.left.size} + \text{rank}(\text{searchKey}, \text{node.right}) \).
Updating Augmented AVL trees

- After an **insertion**, the sizes of all ancestors of the new node should be incremented.
- After a **deletion**, the sizes of all ancestors of the deleted node should be decremented.
- The 2 nodes involved in each **single rotation** must have their sizes updated. (recall that double rotation involves two single rotations)
  - Only sizes of A and B should be updated!

![Augmented AVL tree diagrams](image)
Updating Augmenting AVL trees

- \text{insert}(2): first insert the new node and update sizes of ancestors.

- After the insertion, node 3 is unbalanced, since it is left-heavy and its left child (1) is right heavy, first apply a left rotation; update the sizes of the two involved node (1 and 2).

- Now 3 is left-heavy and its left child (2) is not right-heavy; apply a single rotation between them and update their sizes.
Augmenting AVL trees

Theorem

It is possible to augment an AVL tree by storing the sizes of each subtree so that select and rank operations can be supported in $\Theta(\log n)$ time. The time complexity of other operations (search, insert, and delete) remain unchanged.

- In fact, we can merge such AVL tree with a doubly linked list to support predecessor and successor operations.
Augment Structures Summary

Steps to Augmenting a Data Structure

- Specify an ADT (including additional operations to support).
- Choose an underlying data structure.
- Determine the additional data to be maintained.
- Develop algorithms for new operations.
- Verify that the additional data can be maintained efficiently during updates.