Lecture 10: AVL TREES deletion
Case 2: node v storing entry to be removed has no leaf child
- replace entry at v with entry whose key k preserves binary tree order
  - either largest key entry in left subtree of v or smallest key entry in right subtree of v
  - we choose to take the smallest key entry in the right subtree of v
- copy entry at node u into node v
- remove node u and its leaf left child l with removeExternal(l)
- w is the parent of removed node u

Example: remove 3
Removal in AVL Tree

- Start by removing as in BST
- Let \( u \) be the removed node
  - may be different from the node which used to hold the removed entry
  - any ancestor of \( u \), starting with the parent, \( w \), may have an imbalance
- Example: delete 32

Before delete:

```
    44
   /|
  17 78
 /     |
32   88
```

After delete:

```
    44
   /|
  17 78
 /     |
50    88
```

```
Removal in an AVL Tree

- Follow the path up the tree from the parent of the removed node
- If an unbalanced node is encountered, rebalance the tree with TriNodeRestructure
  - unlike insertion, may have to perform several triNodeRestructure operations
Rebalancing After Removal

- Let $z$ be the first unbalanced node
- Height difference between left and right subtrees of $z$ is exactly 2
- One subtree has height $p$, the other $p+2$
  - $u$ was deleted from the smaller subtree
- 2 cases: right subtree is higher or left subtree is higher
- Similar to insertion, should be able to adapt trinode restructuring
  - get $y$ from the higher subtree of $z$ and $x$ from the higher subtree of $y$
- Unlike insertion, cannot say as much about the higher subtree of $z$
Rebalancing After Removal

- Let us expand the higher subtree of $z$

- At least one child of $y$ is of height $p+1$, both could be of height $p+1$.
- If one is of height $p$ and another of height $p+1$, take $x$ from the higher subtree, as before.
- From which tree to get $x$ if both subtrees have height $p+1$?
Rebalancing After Removal

- If both subtrees of $y$ have height $p+1$, take $x$
  - from the right subtree of $y$ if $y$ is the right child of $z$
  - from the left subtree of $y$ if $y$ is the left child of $z$

- Tree may not rebalance properly if this rule is not followed
Rebalancing After Removal
Performing Rebalance Improperly

- Taking \( x \) from the “same side” as \( y \) when both children of \( y \) have the same height \( p+1 \), we insure that children of \( x \) are not separated.
How Many Tribalance Procedures for Removal?

- Consider case when $y$ has one child higher than the other
- Height of $z$ before deletion is $p+3$, after trinode restructuring that position (now holding $x$) has height $p+2$
- Might cause imbalance among ancestors
  - must continue checking for imbalance
Example of AVL tree Deletion

before delete

after delete, before restructuring

after restructuring
Algorithm AVLtreeDelete(k)

Input: key k

w = TreeDelete(k, T.root) // w holds the parent of deleted node
z = w

while (z ≠ null) // traverse up the tree, checking for imbalance
    setHeight(z)
    if \(|\text{getHeight}(z.left) - \text{getHeight}(z.right)| > 1\) then
        z = TriNodeRestructure(tallerChild(tallerChild(z)), tallerChild(z), z)
        setHeight(z.left); setHeight(z.right); setHeight(z);
    z = parent(z)

- setHeight(z) = 1 + \max(z.left.height, z.right.height)
- tallerChild(v) returns child with larger height
  - if children have equal height, return the left child if \(v\) is a left child, or the right child if \(v\) is the right child
- complexity of AVLtreeDelete(k) is \(O(\log n)\)
Another Deletion Example

A binary search tree is shown, with nodes labeled from 10 to 65. The tree structure and labels are as follows:

- Root node: 30
- Left child of 30: 20
  - Left child of 20: 10
  - Right child of 20: 25
- Right child of 30: 55
  - Left child of 55: 45
  - Right child of 55: 60
  - Left child of 45: 35

The numbers next to the nodes indicate the depth of the nodes in the tree. The tree is a valid binary search tree with each node's value being greater than all the values in its left subtree and less than all the values in its right subtree.
Another Deletion Example
Another Deletion Example
Another Deletion Example
AVL Tree Complexity

- A single trinode restructure is $O(1)$
  - Using a linked-structure binary tree
- Find is $O(\log n)$
  - Height of tree is $O(\log n)$
- Insert is $O(\log n)$
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- Remove is $O(\log n)$
  - Initial find is $O(\log n)$
  - Restructuring up the tree, maintaining heights is $O(\log n)$
- Ordered dictionary implementation where main operations are as efficient as possible $O(\log n)$ in the worst case
“Tree” of Trees, so far

- **General Trees**
  - **Binary Tree**
    - **Complete Binary Tree**
    - **Full (Proper) Binary Tree**
  - **Heap**
    - **Binary Search Tree**
      - **AVL Tree**

**Binary Tree**
- Each level (except last one) has maximum number of nodes. Nodes as far to the left as possible.
- Arbitrary number of children
- At most 2 children
- 0 or 2 children
- Binary search tree order
- Height-balance property