CS122 Algorithms and Data Structures

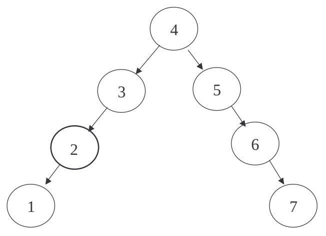
MW 11:00 am - 12:15 pm, MSEC 101

Instructor: Xiao Qin

Lecture 13: Balancing Trees

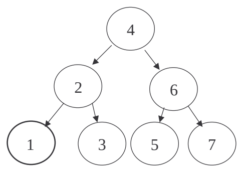
What does it mean to “balance” trees?

Rule #1



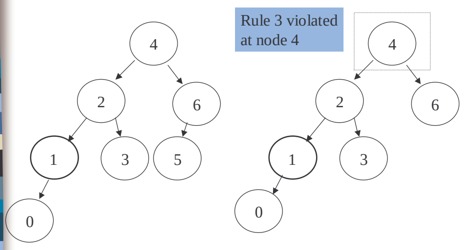
Rule #1: Require that the left and right subtrees of the root node have the same height. We can do better.

Rule #2



Rule #2: Require that every node have left and right subtrees of the same height. Too restrictive.

Rule #3



Rule #3: Require that, for every node, the height of the left and right subtrees can differ by most one.

Balancing Trees

- What does it mean to “balance” trees?

The difference in height of right and left subtrees of any node is either 0 or 1.

- When they are right-heavy or left-heavy, the trees need to be balance.

Approaches

- Reorder data and build a tree
- Restructure the tree when elements arrive and result in an unbalanced tree

Approach 1: Reorder data and build a tree

- Recursive implementation
- Data must be sorted in an array before the tree is created
- If the tree must be used while some elements are still coming, this approach is inefficient.

Approach 2: The DSW Algorithm

- Sorting is avoided
- An array is NOT required
- The basic functions are:

1. Right and left rotations
2. Backbone
3. Transfer the backbone into a balanced tree

Approaches 3: AVL Trees

- Rebalance trees globally or locally
- If a portion of the tree is affected, tree rebalancing can be done locally
- If the insertion or deletion of a node disturbs the balance of a tree, the balance can be immediately repaired.
- Balancing AVL does not guarantee that the resulting tree is perfectly balanced

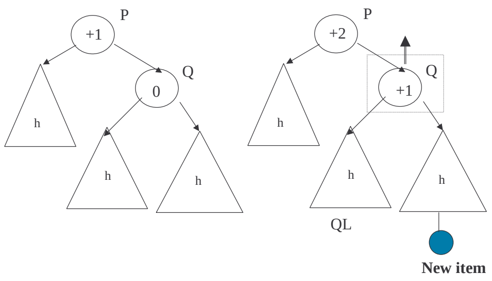
Repair

- Suppose the tree violates a balance condition. How and when can it be repaired?
  - Repair is accomplished via “tree rotations”.
  - Repair is done either during insertions, or after access of a node

Balance factor

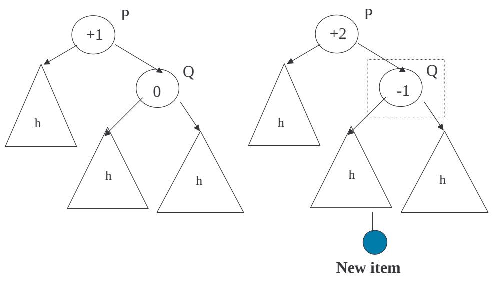
- Difference between the heights of left and right subtrees.
- Height of the right subtree minus the height of the left subtree.

Insertion of a Node



Case 1: Insert a node in the right subtree of the right child
Rotate Q about P

Insertion of a Node (cont.)



Case 2: Insert a node in the left subtree of the right child

Insertion of a Node (cont.)

- In these two cases, the tree P is a stand-alone tree.
- The tree P can be part of a larger AVL tree
- The central problem: Find a node P for which the balance factor is unacceptable after the insertion of a node.

Insertion of a Node (cont.)

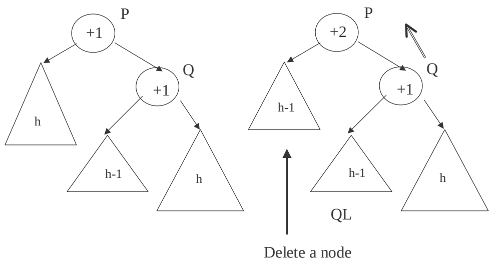
- Find the node P:

1. Move up toward the root from the position of the new node and update the balance factors of the nodes encountered.
2. The first node for which the balance factor becomes +/-2 is the root P of a subtree that needs to be rebalanced.

Deletion of a Node

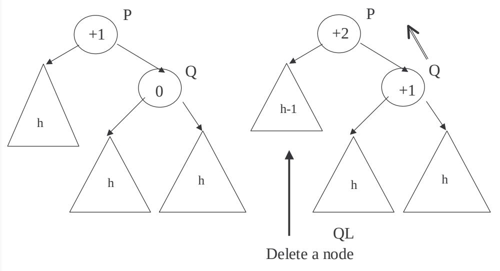
- Only consider the deletion of a node with at most one descendant.
- Balance factors are updated from the parent of the deleted node up to the root.
- It may improve the balance factor of its parent
- It may worsen the balance factor of its grandparent

Deletion of a Node (cont.)



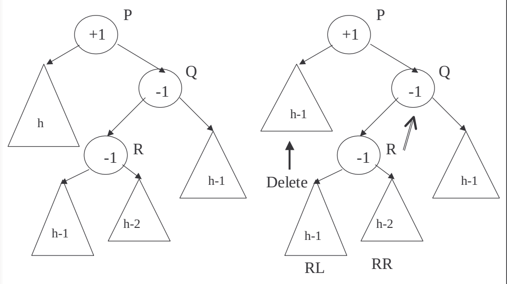
Case 1: Rotate Q about P

Deletion of a Node (cont.)



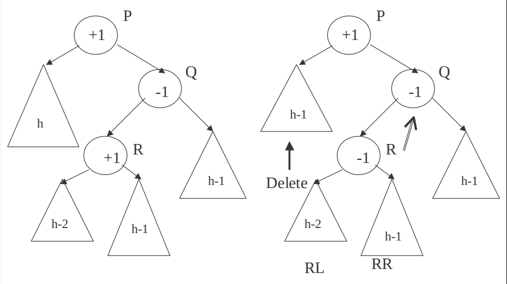
Case 2: Rotate Q about P

Deletion of a Node (cont.)



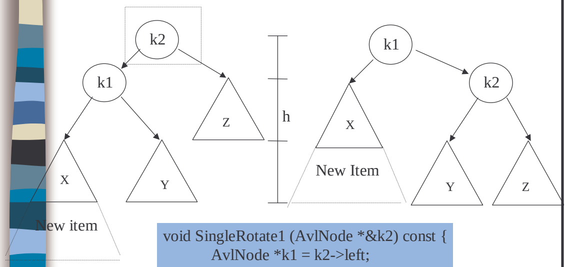
Case 3: Rotate R about Q, then about P

Deletion of a Node (cont.)



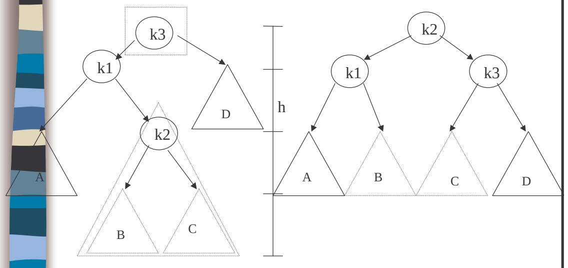
Case 4: Rotate R about Q, then about P

Using C++



```
void SingleRotate1 (AvlNode *&k2) const {
        AvlNode *k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2 = k1;
}
```

Using C++



```
void DoubleRotate1 (AvlNode *&k3) const {
        SingleRotate1( k3->left );
        SingleRotate1( k3 );
}
```

Conclusions

- AVL trees maintain balance of binary search trees while they are being created via insertions of data.
- An alternative approach is to have trees that readjust themselves when data is accessed, making often accessed data items move to the top of the tree. We won’t be covering these (splay trees).