CMPT 225
Lecture 20 – Self-balancing Binary Search Trees
Last Lecture

- We saw how to ...
  - Implement a binary search tree in a variety of ways
    - Array-based
    - Link-based
  - Expressed time efficiency of a BST’s operations for various case scenarios
Learning Outcomes

- At the end of the next few lectures, a student will be able to:
  - Define the following data structures:
    - Binary search tree
    - Balanced binary search tree (AVL)
    - Binary heap
    as well as demonstrate and trace their operations
  - Implement the operations of binary search tree and binary heap
  - Implement and analyze sorting algorithms: tree sort and heap sort
  - Write recursive solutions to non-trivial problems, such as binary search tree traversals
Today’s menu

- Describe how a self-balancing binary search tree balances itself
- Perform insertion into a self-balancing binary search tree, applying appropriate rotations when necessary
- Express time efficiency of self-balancing binary search tree’s operations for various case scenarios
Review: What is a Balanced BST

- From our Lecture 15: Introduction to Trees
  - **Balanced Tree:**
    - A $N$-ary tree in which all $N$ subtrees of any nodes have height that differ by at most 1
  - Adapting the above definition:
    - **Self-Balancing BST:**
      - A BST in which the height of its left subtree and the height of its right subtree differ by at most 1
BST and the *balance* property

- BST does not have the *balancing* property
- BST *insertion* and *deletion* algorithms *only* preserve the *sort ordering* property of the BST
- So, these algorithms do not guarantee the BST will be balanced once these operations have been performed
Self-Balancing Binary Search Trees

- Let’s consider a variation of BST that self-balances itself when insertion, deletion operations are performed ...
AVL

- An AVL tree is a binary search tree with an additional property: a balancing property which ensures that ...
  - Considering any node in the tree, the height of its left subtree and the height of its right subtree differ by at most 1
  - Worst case scenario of insertion, deletion and search is $O(\log n)$
- Named for its inventors: Adelson-Velskii and Landis
- It was the first self-balancing binary search tree and was invented in 1962
Our objective in this lecture is ...

- to learn how to re-balance an AVL self-balancing tree during an **insertion** using the conceptual representation of an AVL self-balancing tree
AVL Tree Balancing - 6 Steps

1. Is the tree a BST?
2. Is the tree an AVL self-balancing tree, i.e., is it balanced? If not, which node is out of balance?
3. Which type of rotation should be applied to re-balance the tree?
4. Apply the rotation, i.e., balance AVL tree
5. Is the tree now an AVL self-balancing tree i.e., is it now balanced?
6. Is the tree still a BST?
AVL Balancing Algorithm

Step 2:
- We need to figure out where (at which node) the AVL tree becomes unbalanced
- We start considering nodes at level $H - 1$ and ascertain whether or not they are balanced nodes, i.e., their subtrees have heights differing by at most 1
- We can skip nodes at level $H$ - Why?
- We work our way up to the root, until we find the node that is out of balance (N.O.O.B.)
- From this out of balance node, we apply the AVL balancing algorithm
AVL Balancing Algorithm – cont’d

Step 3 and Step 4:
- We balance the out of balance node (N.O.O.B.) and its subtrees by performing a tree rotation
  - There are 4 cases of tree rotations
    - Case 1: RR rotation
    - Case 2: LL rotation
    - Case 3: RL rotation
    - Case 4: LR rotation
- We select the appropriate tree rotation and apply it to the N.O.O.B.
Case 1: RR Single rotation

- Before rotation

![Diagram of tree structure before rotation]
Case 1: RR Single rotation – cont’d

- Before rotation

- After rotation
Case 1: Example #1
Case 1: Example #2
Case 2: LL Single rotation

- Before rotation

- After rotation
Case 2: Example #1
Case 2: Example #2
Case 3: RL Double Rotation
Case 3: Example #1
Case 3: Example #2
Case 4: LR Double Rotation
Case 4: Example #1
Case 4: Example #2
Observation About AVL Tree Rotation

- Cases 1 and 2 are mirror images of each other

- Cases 3 and 4 are mirror images of each other
AVL Tree Insertion - 5 Steps

1. Is the tree a BST?
2. Is the tree an AVL self-balancing tree, i.e., is it balanced?
   If not, **balance it first before inserting an element into the tree!**
3. `insert(newElement)` into AVL self-balancing tree
4. Is the tree still an AVL self-balancing tree, i.e., is it still balanced? If not, **balance it!**
5. Is the tree still a BST?
Insertion: Step 4

4. Is the tree still an AVL self-balancing tree, i.e., is it still balanced? If not, balance it!

1. We start at the insertion point and work our way up to the root, ascertaining whether each node is balanced
   ➤ Section of tree unaffected by insertion is still balanced

2. We stop when ...
   ➤ we encounter the first N.O.O.B. node
      ➤ In this case, the tree is declared unbalanced and we follow the 6 steps of AVL Tree Balancing algorithm to rebalance it

   OR

   ➤ we have reached the root without encountering a N.O.O.B.
      ➤ In this case, the tree is declared balanced and we go to Step 5
Insertion: Let’s try!
Note about Step 4

- We always balance the first node out of balance (N.O.O.B.) we encounter as we proceed from insertion point to the root.
- Often, balancing the lowest subtree, which has become unbalanced due to the insertion, is all we need to do in order to balance the whole tree.
In Single Rotation cases 1 and 2, we say that the insertion occurs on the outside of the tree either ... 
- in the right subtree of the right child (which we call the RR case – case 1)

Or
- in the left subtree of the left child (which we called the LL case – case 2)
Note about Double Rotation

- In Double Rotation cases 3 and 4, we say that the insertion occurs on the inside of the tree either
  - in the left subtree of the right child (which we call the RL case – case 3)

Or

- in the right subtree of the left child (which we call the LR case – case 4)
Time Efficiency of AVL’s operations

<table>
<thead>
<tr>
<th>Operations</th>
<th>$H &lt;&lt; n$</th>
<th>$H = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retrieve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>traverse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- It can also be shown that balancing is required only for less than 50% of insertions
A peek at AVL self-balancing tree implementation

class AVLTreeNode {

private:
    ElementType element; // element stored in node
    AVLTreeNode* leftChild; // link to left subtree
    int heightLeft; // height of left subtree
    AVLTreeNode* rightChild; // link to right subtree
    int heightRight; // height of right subtree

    ...

}
Learning Check

- We can now ...
  - Describe how a self-balancing binary search tree balances itself
  - Perform insertion into a self-balancing binary search tree, applying appropriate rotations when necessary
  - Express time efficiency of self-balancing binary search tree's operations for various case scenarios
Next Lecture

- Tree sort and heap