Part 2

Algorithm and Data Structure

Aneesh Garg - agarg6@uncc.edu - 800815619
4/10/2014

1.0 How to run program ... 3
2.0 Input Sizes ... 3
3.0 Data Structures ... 3
3.1 Node Class ... 3
3.1.1 Methods and Parameters ... 3
3.2 BinarySearchTree Class ... 5
3.2.1 Methods and Parameters ... 5
3.2 AVLTree Class ... 6
3.2.1 Methods and Parameters ... 6
3.2 ImplementationPartTwo Class ... 7
3.2.1 Points to remember ... 7
4. Output and Conclusion ... 8

1.0 How to run program

Prerequisite: Proper installation of java

- Go to Part2\src folder and open command window there.
- Compile ImplementationPartTwo.java file
  `javac ImplementationPartTwo.java`
- Run the compiled java file
  `java ImplementationPartTwo`

2.0 Input Sizes

I have tested all algorithms on following input sizes.

[100, 500, 1000, 5000, 10000]

3.0 Data Structures

3.1 Node Class

This class represents the basic node which stores the key. Combination of these nodes constitute a tree.

3.1.1 Methods and Parameters

```
/**
 * Stores the key in node.
 */
private int key;
/**
 * Stores reference to left child of the node.
 */
private Node left;
/**
 * Stores reference to right child of the node.
 */
private Node right;
/**
 * Stores reference to parent of the node.
 */
private Node parent;
```

```java
    /**
     * Whether the node is root node or not
     *
     * @return boolean true or false
     */
    public boolean isRoot() {...

    /**
     * Whether the node is leaf node or not
     *
     * @return boolean true or false
     */
    public boolean isLeaf() {...

    /**
     * Whether the node is internal node of not
     *
     * @return boolean true or false
     */
    public boolean isInternal() {...

    /**
     * Whether any children of this node is a leaf node or not
     *
     * @return boolean true or false
     */
    public boolean isAboveExternal() {...

    /**
     * Whether the node is right child of its parent or not.
     *
     * @return boolean true or false
     */
    public boolean isRightChild() {...

    /**
     * This method calculates the balance factor of this particular node.
     */
    public int calculateBalanceFactor() {...


  /**
   * Gets height of the tree starting at calling node.
   *
   * @return int height
   */
  public int getHeight() {...
```

3.2 BinarySearchTree Class

This class represents the Binary Search tree.

3.2.1 Methods and Parameters

```
/**
 * Stores the reference to the root node of the tree.
 */
private Node root;
/**
 * Stores the number of nodes in a tree.
 */
private int size;


/**
 * Inserts a key into a binary search tree.
 *
 * @param key
 *            : Key to be inserted.
 * @return Reference to the node storing recently inserted item.
 */
public Node insertItem(int key) {...

/**
 * Finds and Return Node object in the tree with a given key.
 *
 * @param key
 *            : Key of the item to be searched
 * @return Node with key equal to input key of null if node is not found
 */
public Node findItem(int key) {...

/**
 * This function will delete single specified node from binary search tree.<br>
 * Deletion of any node from binary search tree can be done under following
 * cases:
 * <ol>
 * <li>Deleting a leaf node</li>
 * <li>Deleting a node having only right subtree (empty left subtree)</li>
 * <li>Deleting a node having only left subtree (empty right subtree)</li>
 * <li>Deleting a node having non-empty left and right subtree</li>
 * </ol>
 *
 * @param key
 *            : Key of the node to be removed
 * @throws EmptyTreeException
 *             : thrown if tree is empty
 * @throws KeyNotFoundException
 *             : thrown if key is not present in the tree
 * @throws InvalidTreeOperation
 *             : thrown if any invalid operation takes place in the tree
 * @return reference to the node that was just deleted.
 */
public Node removeItem(int key) throws EmptyTreeException,...
```

```java
/**
 * This method finds the node which comes next in inorder traversal of the
 * binary search tree.
 *
 * @param node
 *            : Node whose next inorder is to be found
 * @return Node : next inorder node
 */
public static Node nextInorder(Node node) {

/**
 * This method prints the binary search tree according to inorder traversal
 */
public void printTree() {
```

3.2 AVLTree Class

This class extends the Binary Search tree and represents AVL Trees.

3.2.1 Methods and Parameters

Below are the new methods that are not there in its super class.

```java
/**
 * Performs left rotation with input node as top node in rotation.
 * @param imBalancedNode : reference to the node that is the top node.
 */
private void rotateLeft(Node imBalancedNode) {

/**
 * Performs right rotation with input node as top node in rotation.
 * @param imBalancedNode : reference to the node that is the top node.
 */
private void rotateRight(Node imBalancedNode) {

/**
 * Performs restructuring and re-balancing of the tree after a node is deleted.
 * @param z: node that is imbalanced due to removal.
 */
private void restructure(Node z) {
```

3.2 ImplementationPartTwo Class

This class is used test the tree and perform required operations.

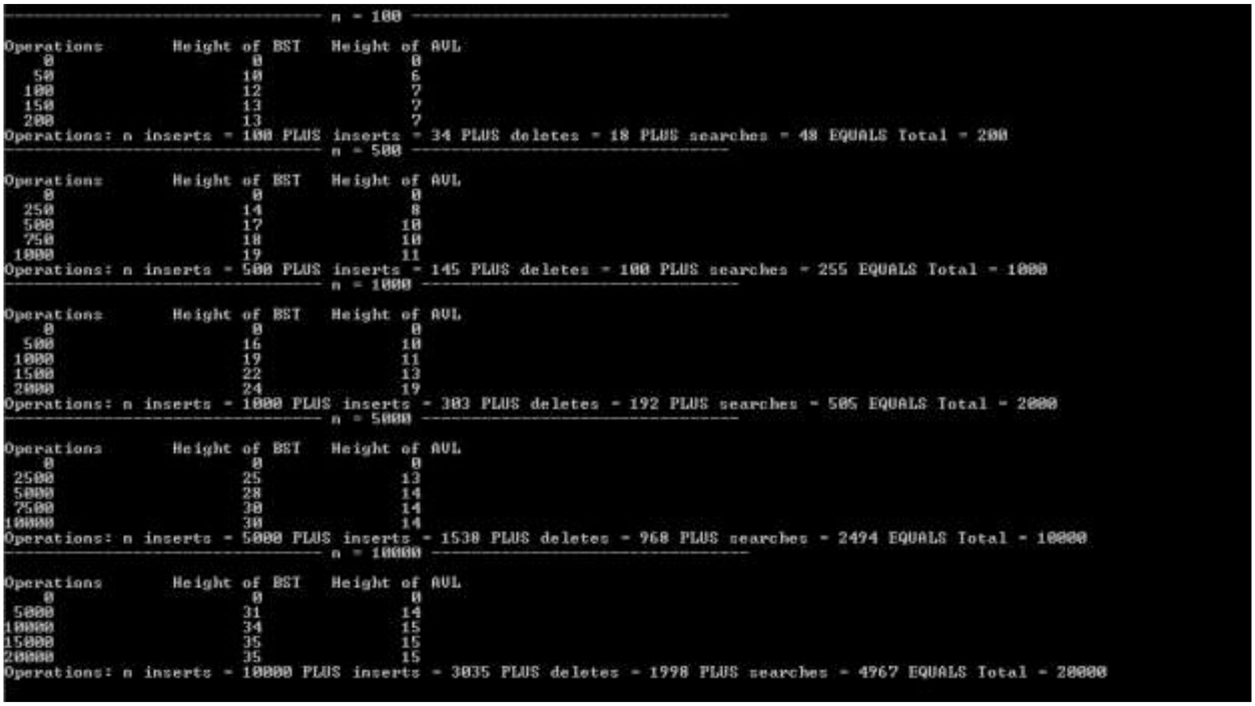
3.2.1 Points to remember

1. I am performing each operation with an element (same) on both BST and AVL Tree. This will help us in actual comparison of heights after same operations.
2. I am first performing n insertions then after that n operations with a probability of 50% Search, 30% Insertion and 20% deletion.
3. While making insertions I am taking a random number which is already not there in the tree. Having duplicate keys can create a lot of issues. I was facing issues with following scenario:

   > I want to make my `avl-tree` support duplicate keys but there is a problem with the default behavior of the `binary search tree` with duplicates that the rotation could make nodes with equal key be on the left and the right of the parent.
   >
   > For example when adding three nodes all with key A will cause the tree to do a rotation to be something like this:
   >
   > ```
   >     A
   >    / \
   >   A   A
   > ```
   >
   > So getting all the entries with that key will be a problem...and searching in both direction is not good.

   [Ref: Similar problem on Stack Overflow]
   It can be handled using several ways but that would have made program too complex.
4. While searching and deleting I am taking any random number so it is highly possible that the key is not found during these operations. I am throwing the appropriate exception and printing that key is not found.

4. Output and Conclusion



```
----------------------------------- n = 100 -----------------------------------

Operations        Height of BST   Height of AVL
    0                  0               0
   50                 10               6
  100                 12               7
  150                 13               7
  200                 13               7
Operations: n inserts = 100 PLUS inserts = 34 PLUS deletes = 18 PLUS searches = 48 EQUALS Total = 200
----------------------------------- n = 500 -----------------------------------

Operations        Height of BST   Height of AVL
    0                  0               0
  250                 14               8
  500                 17              10
  750                 18              10
 1000                 19              11
Operations: n inserts = 500 PLUS inserts = 145 PLUS deletes = 100 PLUS searches = 255 EQUALS Total = 1000
----------------------------------- n = 1000 -----------------------------------

Operations        Height of BST   Height of AVL
    0                  0               0
  500                 16              10
 1000                 19              11
 1500                 22              13
 2000                 24              19
Operations: n inserts = 1000 PLUS inserts = 303 PLUS deletes = 192 PLUS searches = 505 EQUALS Total = 2000
----------------------------------- n = 5000 -----------------------------------

Operations        Height of BST   Height of AVL
    0                  0               0
 2500                 25              13
 5000                 28              14
 7500                 30              14
10000                 30              14
Operations: n inserts = 5000 PLUS inserts = 1538 PLUS deletes = 968 PLUS searches = 2494 EQUALS Total = 10000
----------------------------------- n = 10000 -----------------------------------

Operations        Height of BST   Height of AVL
    0                  0               0
 5000                 31              14
10000                 34              15
15000                 35              15
20000                 35              15
Operations: n inserts = 10000 PLUS inserts = 3035 PLUS deletes = 1998 PLUS searches = 4967 EQUALS Total = 20000
```

As the number of nodes in the tree increases, height of BST increases much faster than AVL Tree. We balance AVL Tree after every insertion or deletion keeping the height of the tree in range of log (n). Therefore in worst case scenario complexity of AVL Tree will be O (log n) while for BST worst case complexity can be O (n) if keys are inserted in sorted order.