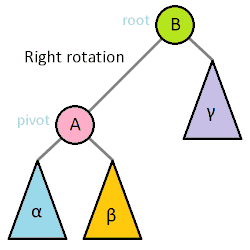
Data Structures 2

Lab 2:

***Implementing AVL Trees***



Dahlia Chehata Mahmoud 28

Mohamed Ayman farrag 61

Joumana Ahmed

* **Problem Statement:**

***It is required to implement AVL tree***, a binary search tree with a balance condition. The balance condition is easy to maintain, and it ensures that the depth of the tree is O(log n). An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1. (The height of an empty tree is defined to be -1.)

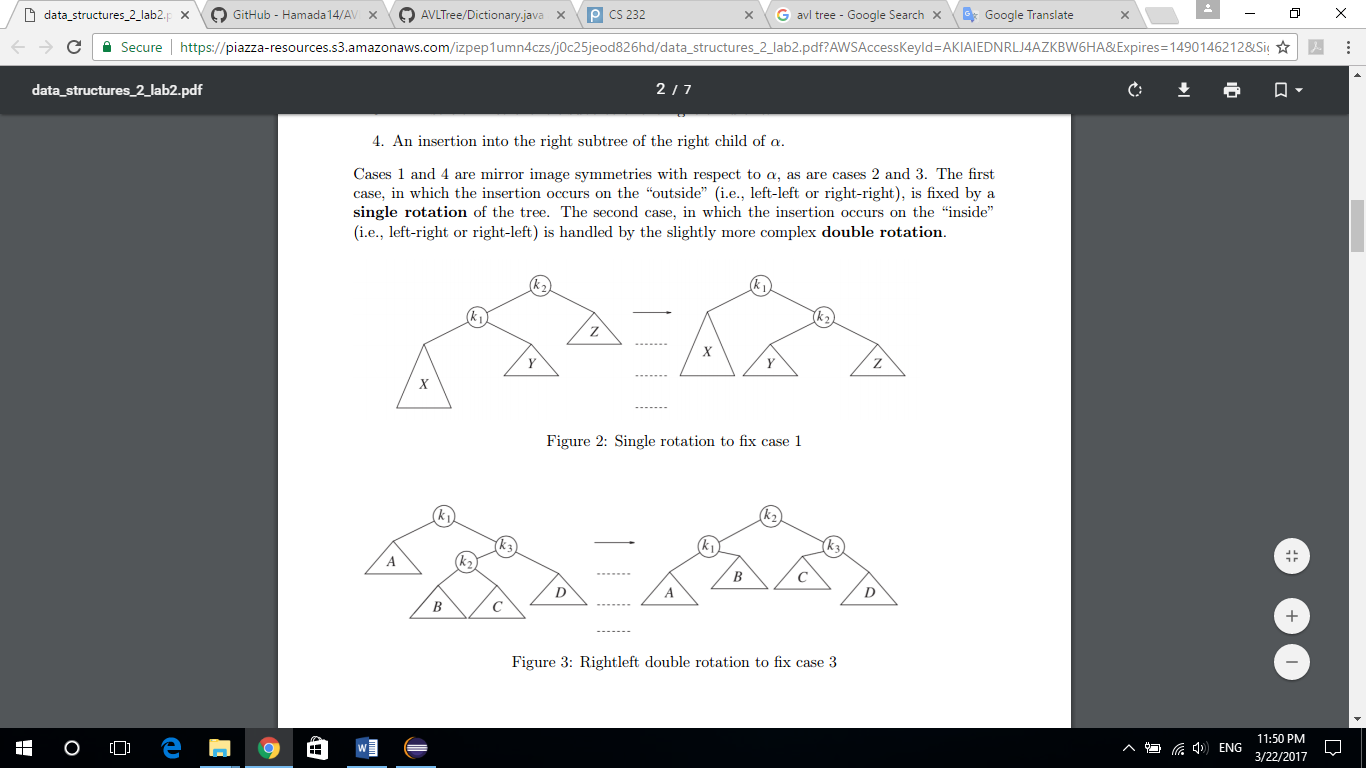
***The implementation of the basic operations: insert, delete, search and get the tree height are required***. All the tree operations can be performed in O(log n) time, except possibly insertion and deletion. The reason that insertions and deletions are potentially difficult is that the operation could violate the AVL tree property.1 The balance of an AVL tree can be maintained with a simple modification to the tree, known as a ***rotation with its four combinations (left left, left right, right left, right right).***

***A dictionary application is implemented using the AVL tree*** with the operations:

Load file, delete, insert a word, check the existence of a word, get height and get size.

***Given the three interfaces INode, IAVLTree, IDictionary,***the classes node,tree and dictionary were implemented

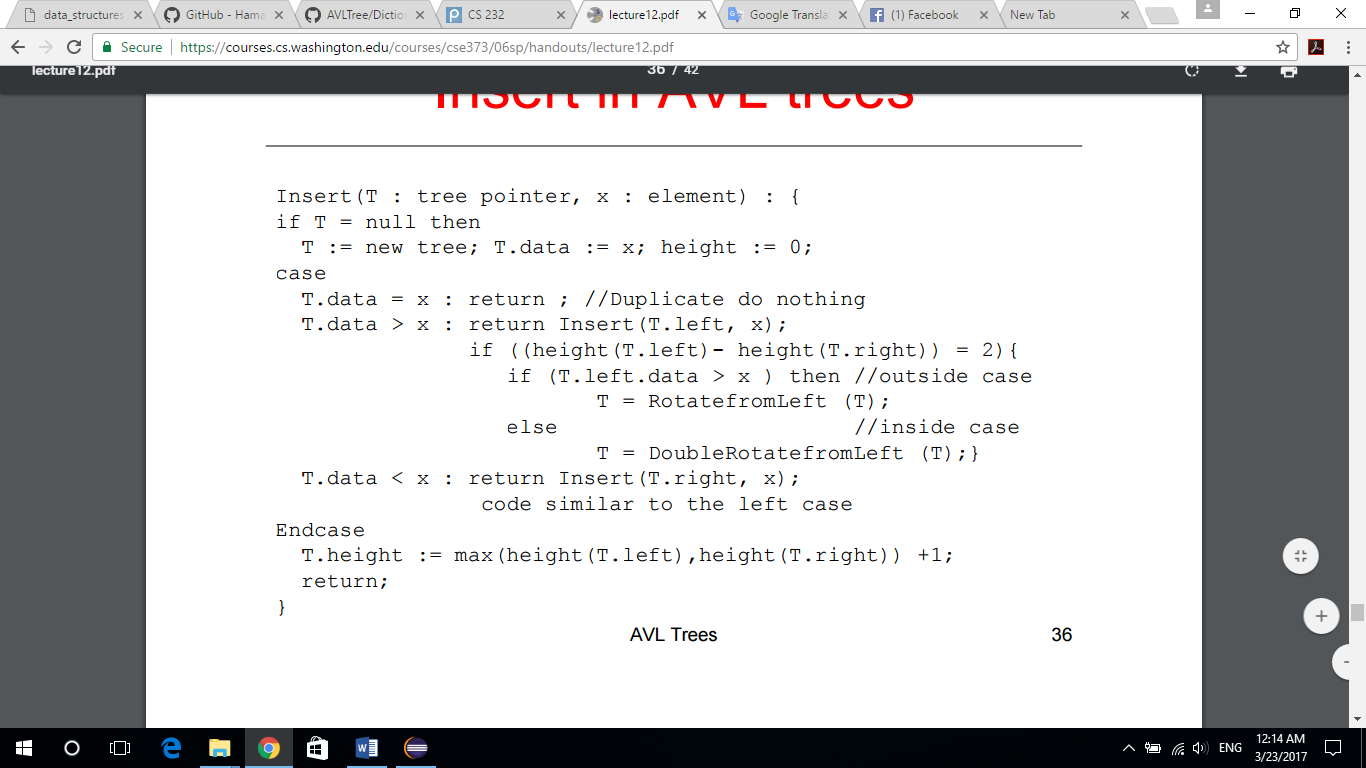
***Some of the major methods***: the balance factoring method, the rebalancing method, update the height, rotate left, rotate right.



* **Algorithms:**

**AVL operations:**

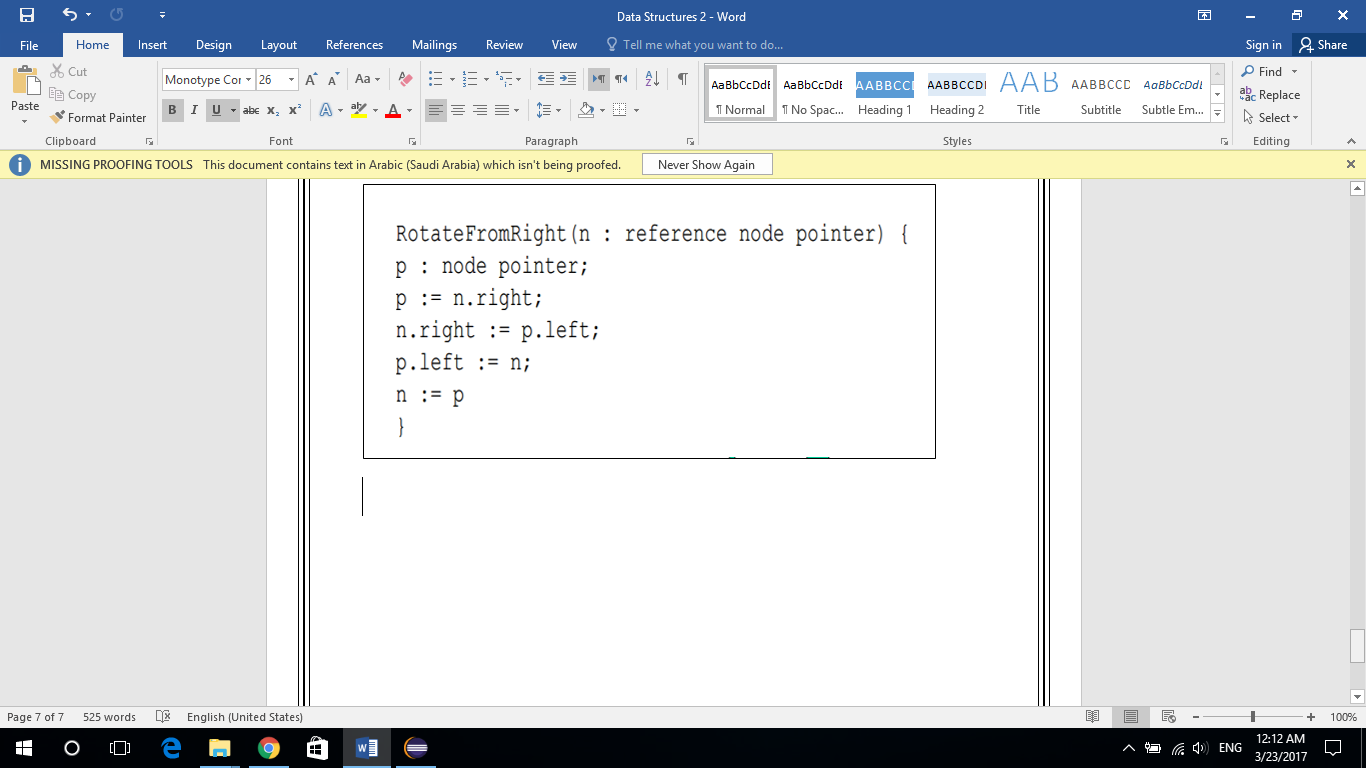
**1.Insert:**



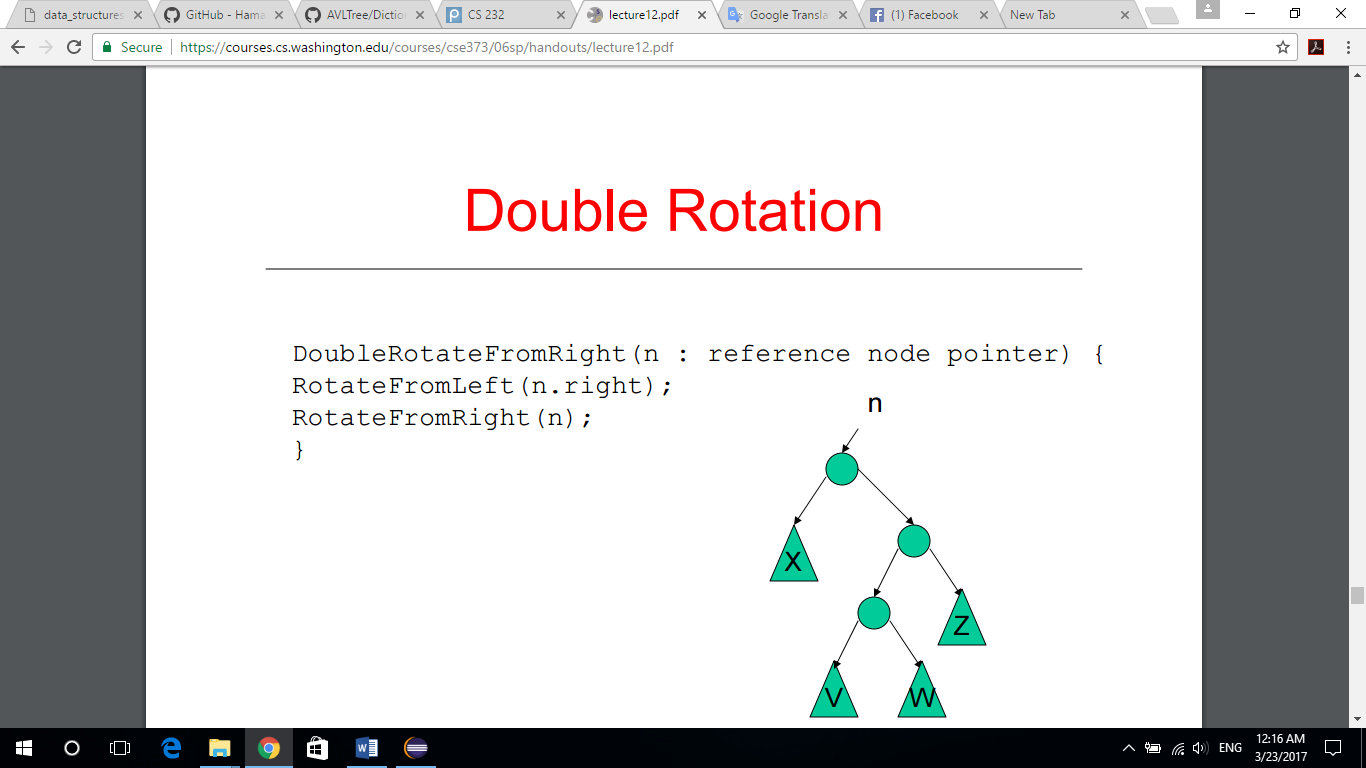
**2.deletion:**

|  |
| --- |
| **for**(X=parent(N);X!=**null**;X=G){  G=parent(X);  **if**(N==left\_child(X)){  **if**(BalanceFactor(X)>0){  Z=right\_child(X);  b=BalanceFactor(Z);  **if**(b<0)  N=rotate\_RightLeft(X,Z);  **else**  N=rotate\_Left(X,Z);  }  **else**{  **if**(BalanceFactor(X)==0){  BalanceFactor(X)=+1;  **break**;  }  N=X;  BalanceFactor(N)=0;  **continue**;  }  }  **else**{  **if**(BalanceFactor(X)<0){  Z=left\_child(X);  b=BalanceFactor(Z);  **if**(b>0)  N=rotate\_LeftRight(X,Z);  **else**  N=rotate\_Right(X,Z);  }  **else**{  **if**(BalanceFactor(X)==0){  BalanceFactor(X)=–1;  **break**;  }  N=X;  BalanceFactor(N)=0;  **continue**;  }  }  parent(N)=G;  **if**(G!=**null**){  **if**(X==left\_child(G))  left\_child(G)=N;  **else**  right\_child(G)=N;  **if**(b==0)  **break**;  }  **else**{  tree->root=N;  **continue**;  }  } |

**3. Single Rotation:**



4**.Double Rotation:**



* **Data structures Used:**

**AVL Tree:**

To avoid the vast memory usage we use reference between nodes where each node has two reference to the left and the right sub tree avoiding the use of external arrays to data structure.

Also the array model of the AVL tree doesn't have any advantages over the reference model because whether in search or addition of new nodes, Iterator must traverse from the root till the leaf or the needed node.

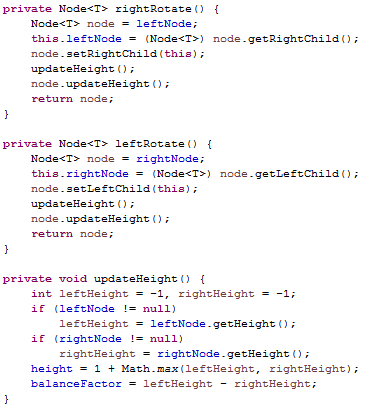
**Dictionary:**

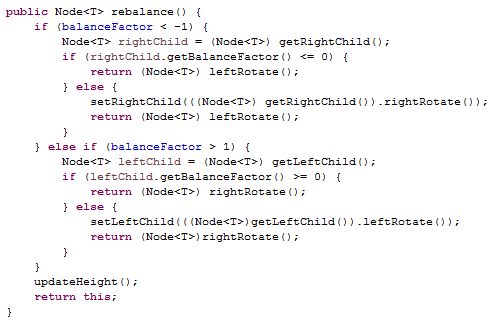
in case of the Dictionary we need a data structure supporting two operations insertion and search for words and it has to support these operations fast enough to be user friendly in case of large number of words >= 100,000 word.

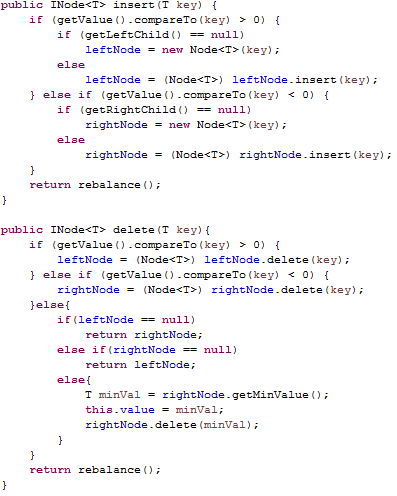
AVL Tree supports these operations in O(log n) because of its property where the difference of height between the left and the right sub tree must not exceed 1 or we'll have to rotate the nodes to preserve this property.

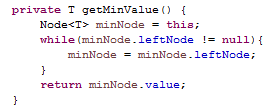
* **Code snippets:**

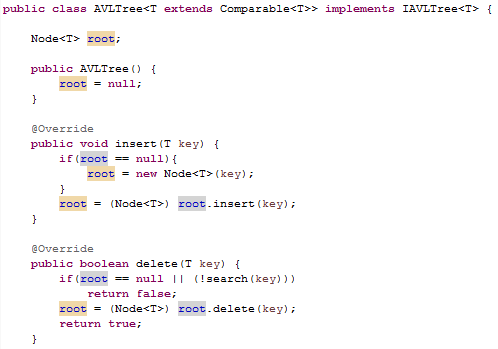
**1.Class Node:**

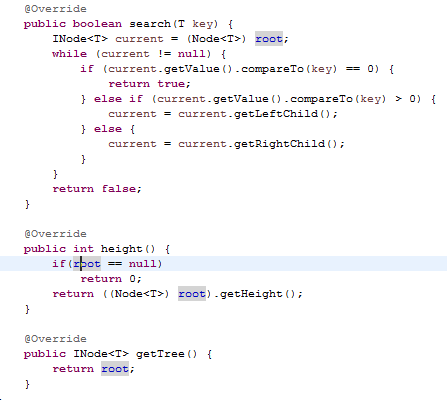
****

****

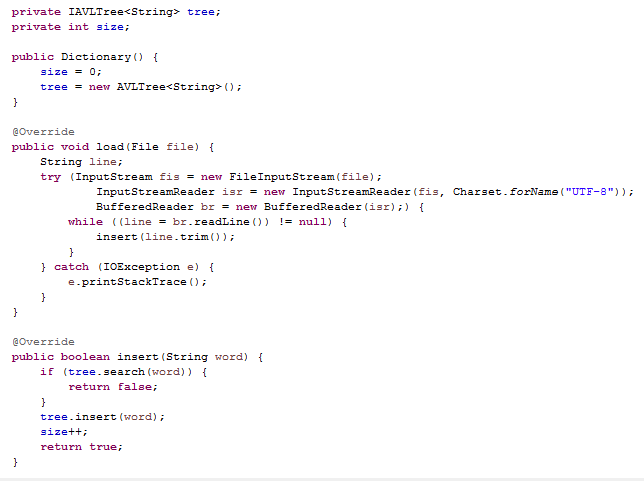
****

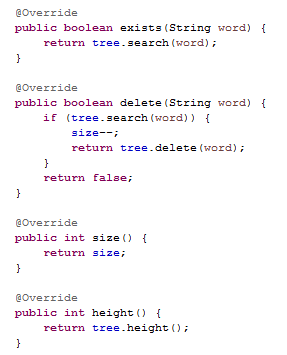
****

**2.Class AVLTree: **

****

**3.Class Dictionary:**

****

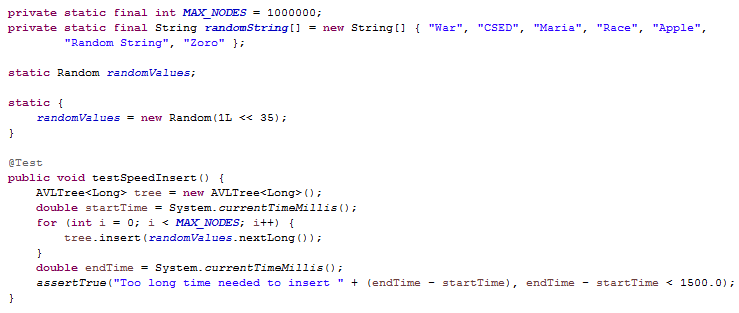
****

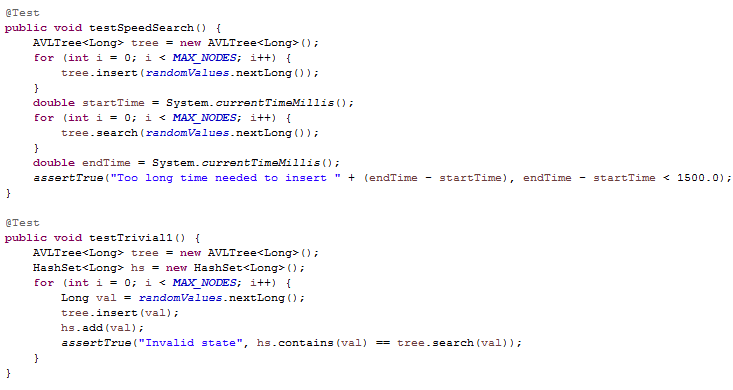
* **Sample Runs:**

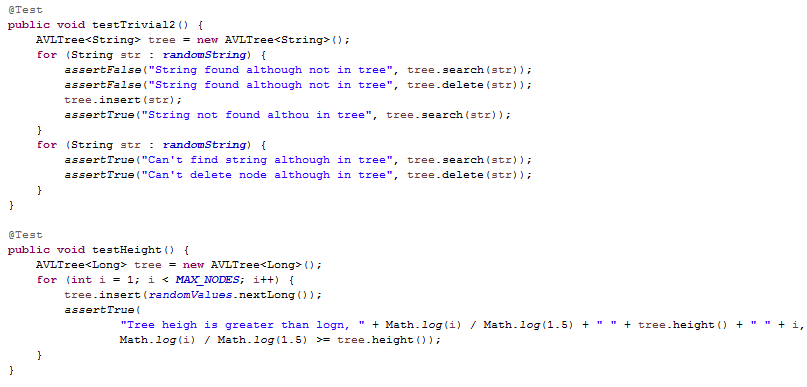
Sample Runs were made using JUnit testing in java to support checking and validating the code after version iterations and changing anything in the code to avoid the appearance of new bugs during the fixes in between versions.

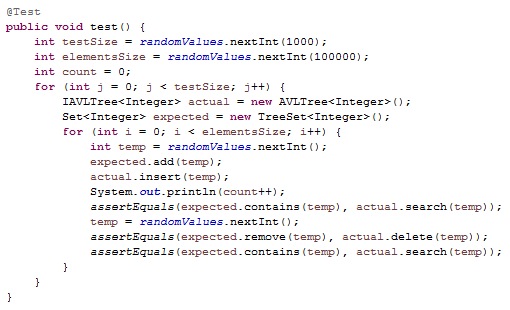
tests covered the d

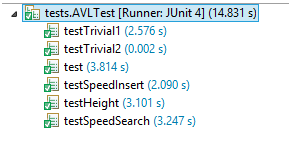
uplication of keys, checking the height of the tree and checking the speed of insertions, search and deletions of large numbers of keys.



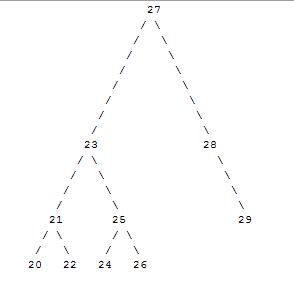


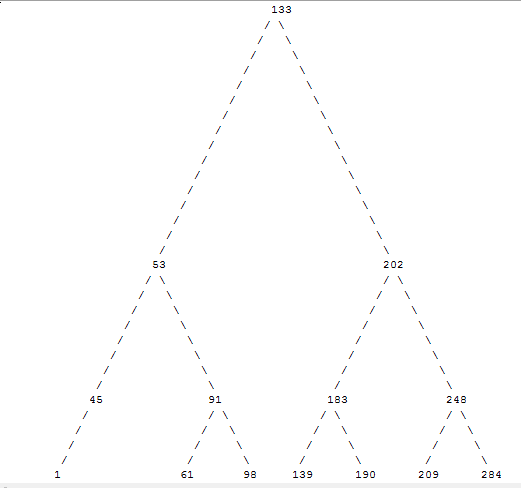


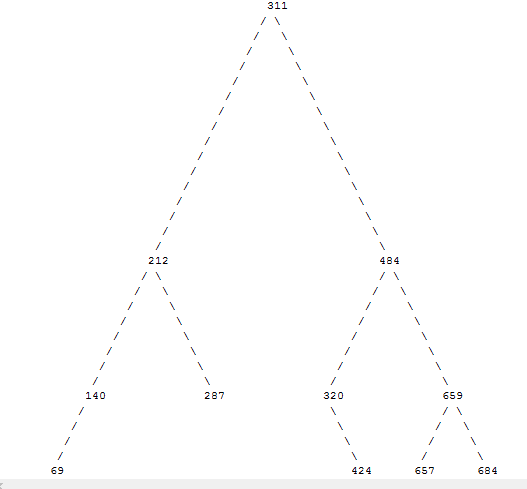




Sample Run using animated ASCII







Code used for Animation:

