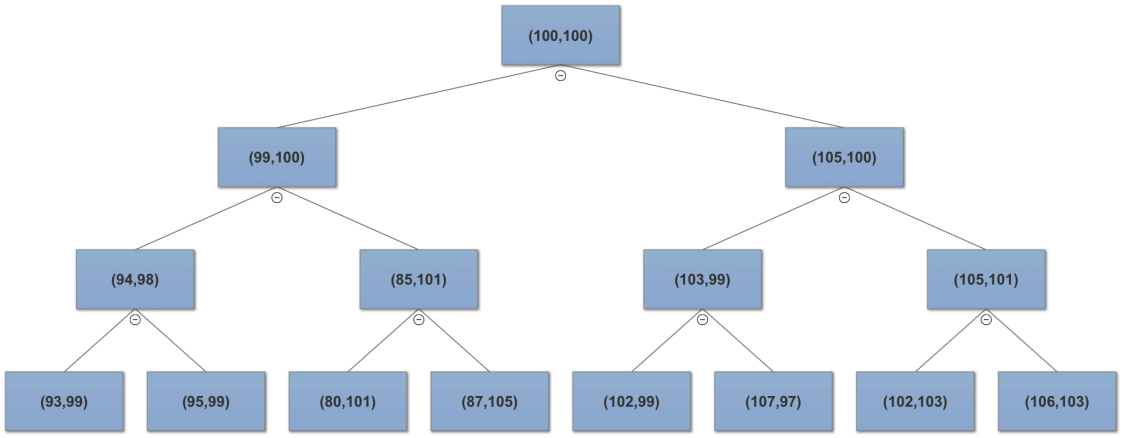
Jason Schenck
CSC-385
Module 12 – 2D Binary Search Tree Analysis

Test Case 1: Balanced BST



For my first test case, by hand I constructed the above binary tree following the assignment's specification of alternating comparisons of x and y coordinates based on level. The above tree is perfectly balanced.

- **Test1:** Level order traversal output

  Expected output: (100,100), (99,100), (105,100), (94,98), (85,101), (103,99), (105,101), (93,99), (95,99), (80,101), (87,105), (102,99), (107,97), (102,103), (106,103)

  Actual output: (100,100), (99,100), (105,100), (94,98), (85,101), (103,99), (105,101), (93,99), (95,99), (80,101), (87,105), (102,99), (107,97), (102,103), (106,103)

  Passed

- **Test 2**: Add and Contains methods

  All items successfully added, no errors. Level order traversal prints correctly after adding.

  Expected output:
  Contains (95,99) → true,
  Contains (10,13) → false,

Contains (102, 99) → true

Actual output:
Contains (95,99) → true,
Contains (10,13) → false,
Contains (102, 99) → true

Passed

- **Test 3**: Duplicate item add throws exception

  Expected output after "*tDtree.add(102,99);*":
  IllegalArgumentException: Cannot add duplicate item to the tree!

  Actual output:
  IllegalArgumentException: Cannot add duplicate item to the tree!

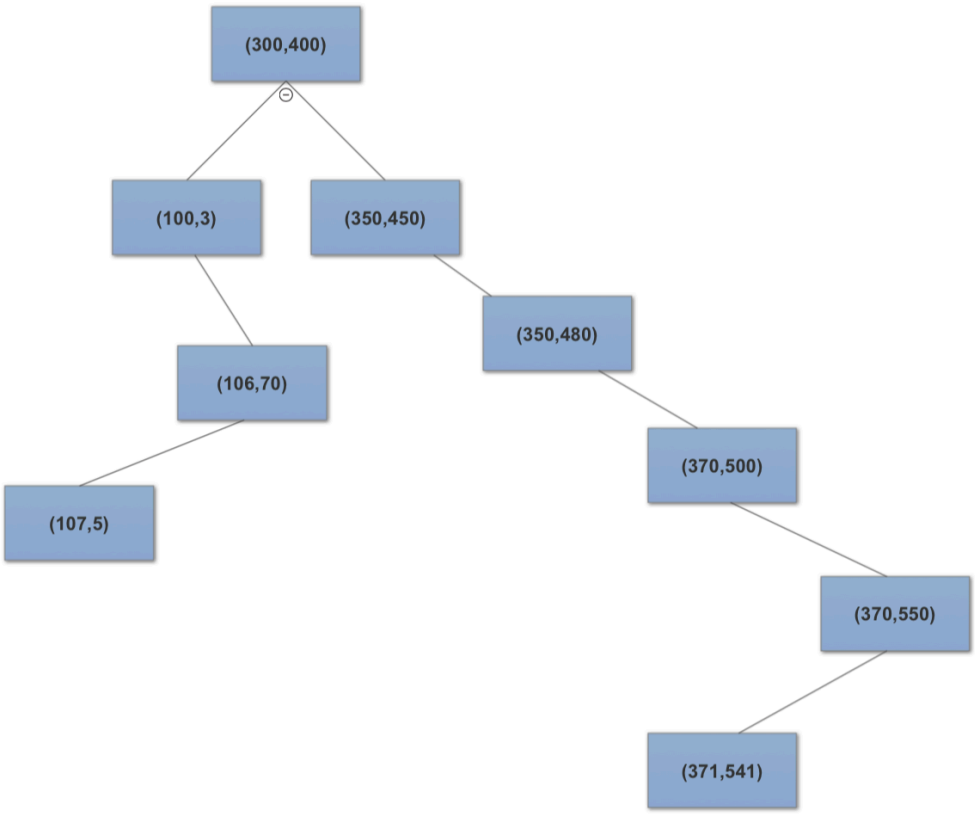
  Passed

- **Analysis of Time Efficiency:**
  Generally speaking, my contain and add methods will run on average case of O(logn), when the tree has some branching, and in the worst case O(N), when the tree has no branching at all.

  In general, my methods will run at O(1) time in the best case scenario when either a node is being added to an empty tree or the node we are searching for happens to be in the root.

  However, balance is very important in BST's because it improves the worst case performance of the recursive add and contains algorithms. Since this tree is a perfectly balanced binary search tree it will always perform at O(logn) time efficiency due to the ability to evenly eliminate branches from inspection during recursion, similar to the efficiency of a mergesort algorithm.

Test Case 2: Unbalanced BST



For the second required test case, I constructed a very unbalanced binary tree posted above.

- **Test1:** Level order traversal output

  Expected output: (300,400), (100,3), (350,450), (106,70), (350,480), (107,5), (370,500), (370,550), (371,541)

  Actual output: (300,400), (100,3), (350,450), (106,70), (350,480), (107,5), (370,500), (370,550), (371,541)

  Passed

- **Test 2**: Add and Contains methods

  All items successfully added, no errors. Level order traversal prints correctly after adding.

  Expected output:

Contains (106,70) → true,
Contains (4,1) → false,
Contains (4,90) → false
Contains (371,541) → true
Contains (101,2) → false
Contains (100,3) → true
Contains (300, 400) → true

Actual output:
Contains (106,70) → true,
Contains (4,1) → false,
Contains (4,90) → false
Contains (371,541) → true
Contains (101,2) → false
Contains (100,3) → true
Contains (300, 400) → true

Passed

- **Test 3**: Duplicate item add throws exception

  Expected output after "*tDtree.add(106,70);*":
  IllegalArgumentException: Cannot add duplicate item to the tree!

  Actual output:
  IllegalArgumentException: Cannot add duplicate item to the tree!

  Passed

- **Analysis of Time Efficiency:**
  This BST is very unbalanced with minimal branching. This means that during recursion there will be less work to eliminate and much of the data will end being processed similar to a linkedlist of nodes. For example the path from root node (300,400) to leaf node (371,541) has no branching and is simply in memory as a linkedlist of connecting nodes. In order to traverse, find, add etc this nodes recursively would have similar linear running time to a linkedlist.

  I would say that for this unbalanced tree, on average it would perform pretty close to O(N).. depending on the input value it maybe rarely perform at a time faster than O(N), but on the average it will be O(N).

  And most certainly the worst case is going to be linear O(N).

For any tree with zero branching, aka a linkedlist, it will always perform at O(N).