Binary Search Tree Versus AVL Tree Performance
by Peter Master
12/01/2016

in order to compare the performance of the Binary Search Tree implementation provided and the AVL Tree that I implemented, I wrote test.cc which includes the following functions to test the run times of different operations with different quantities of items for both trees.

I ran the tests many times each for about an hour on CSIL and have listed estimated averages for the runtimes of each as well as provided graphs for visualization of the comparison.

These runtimes should only be considered relative to the other runtimes listed in this report; the runtimes themselves are not individually important or relevant outside the scope of this report.

Conclusion: AVL Trees seem to handle sorted data far, far better than BSTs, while BSTs are very slightly more effective at handling random data.

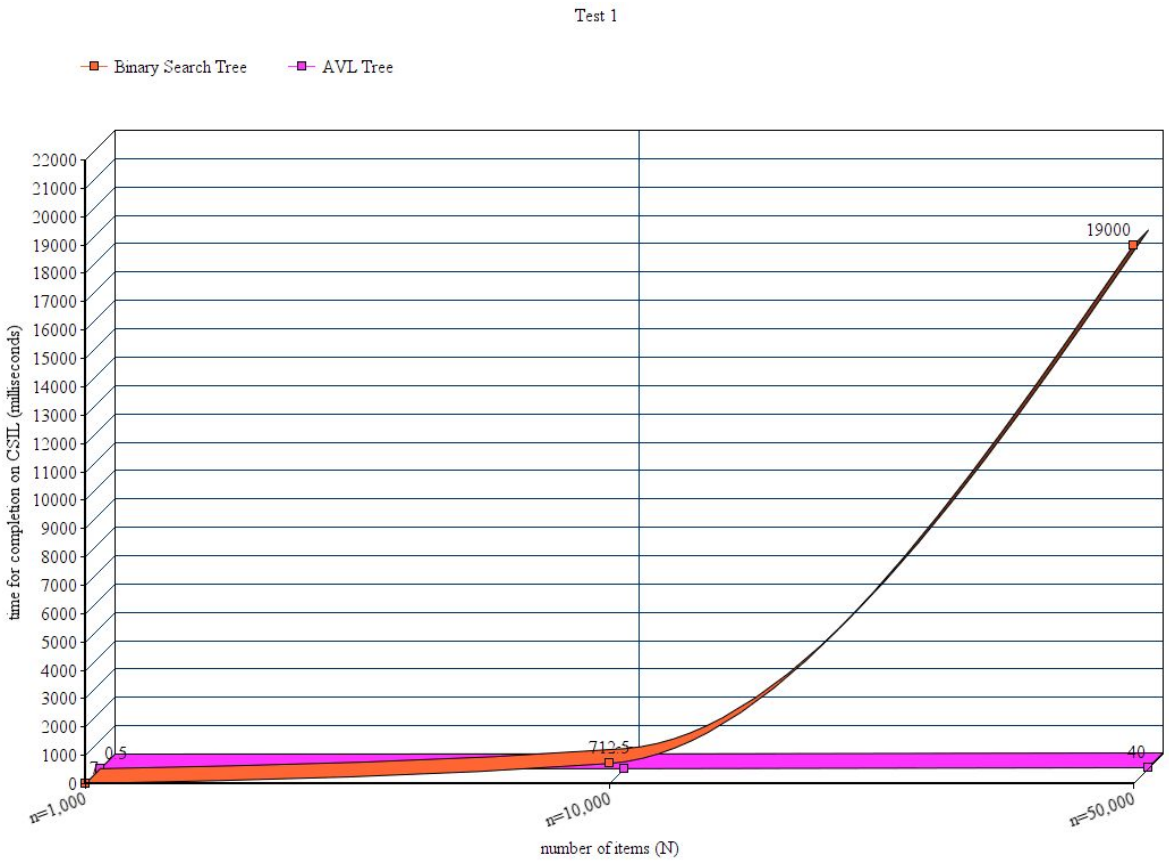
This makes sense, because AVL Trees require additional constant time per insert and delete in order to maintain a small height relative to BSTs, however this additional constant time is far outweighed when sorted (or semi-sorted) data is inserted, one after another, into each tree for large quantities of N because Insert, Delete, and Access functions scale linearly with height of trees which scales with N for BSTs under worst-case scenario (sorted data input) and log N for AVL Trees (no matter the input). This is because when sorted data is inserted, one after another, into an empty BST, that BST is equivalent to a Linked List (O(N) time complexity for insert, delete, and access) with worse space complexity.

One surprising thing that I found from this experiment is that AVL Trees are actually faster with sorted data input, accessing, and deletion than with random data input, accessing, and deletion (that is to say, AVL Trees fare better at Tests 1 and 2 than 3). This is because, with sorted data, the tree only has to rotate a maximum of one time per insert and delete, whereas with random data, the tree is likely to rotate by some factor times height -- a constant number of rotations versus a logarithmic (with respect to N; linear with respect to height) number of rotations.

TEST 1:
insert N elements in increasing order,
access N elements in increasing order,
delete N elements in increasing order.

```
for (int i = 0; i < N; i++)
  tree.Insert(i);
for (int i = 0; i < N; i++)
  tree.Access(i);
for (int i = 0; i < N; i++)
  tree.Delete(i);
```

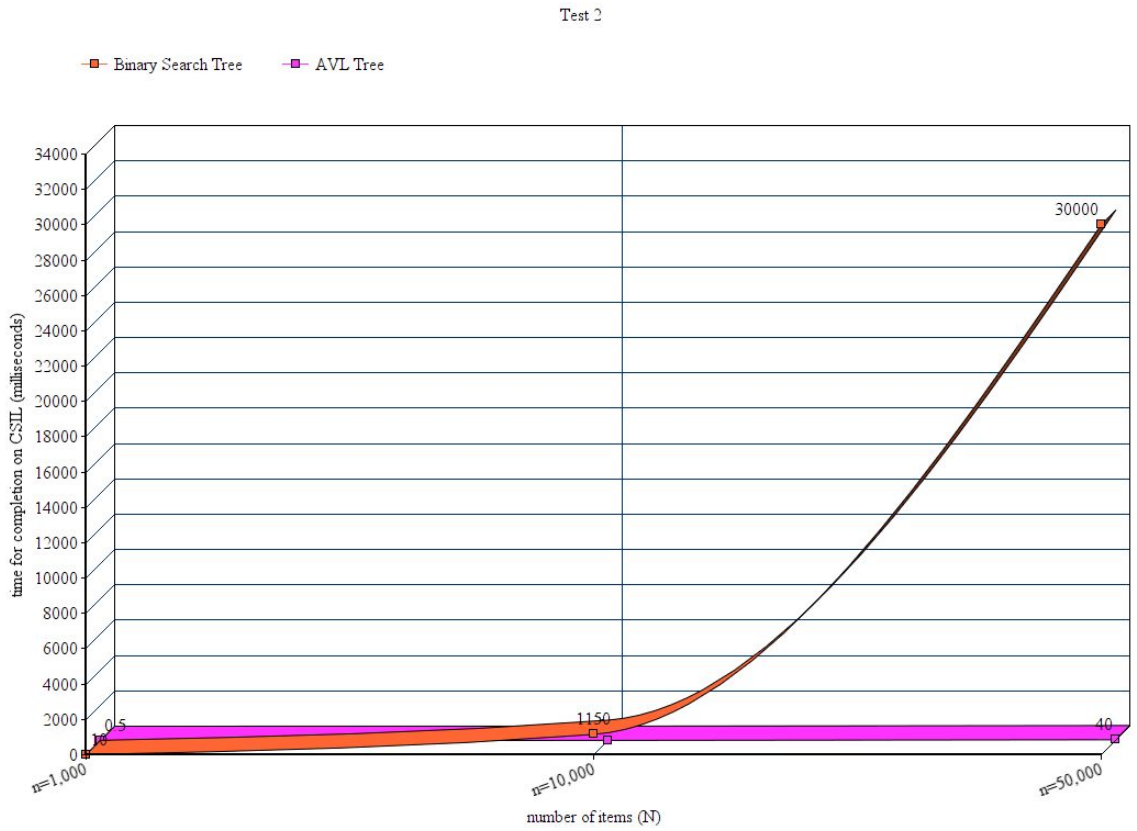
TEST 1 average time for execution on CSIL terminal	**Binary Search Tree**	**AVL Tree**
N=1,000	7ms	.5ms
N=10,000	7,125ms	6.3ms
N=50,000	19,000ms	40ms
approximate quadratic regression	t_ms(N) = (7.73E-6*N^2) - (.00665*N) + 6	t_ms(N) = (4.04E-9*N^2) + (6.00E-4*N)



TEST 2:
insert N elements in increasing order,
access N elements in decreasing order,
delete N elements in decreasing order.

```
for (int i = 0; i < N; i++)
  tree.Insert(i);
for (int i = N - 1; i >= 0; i--)
  tree.Access(i);
for (int i = N - 1; i >= 0; i--)
  tree.Delete(i);
```

TEST 2 average time for execution on CSIL terminal	**Binary Search Tree**	**AVL Tree**
N=1,000	10ms	.5ms
N=10,000	1,150ms	6.5ms
N=50,000	30,000ms	40ms
approximate quadratic regression	t_ms(N) = (1.21E-5*N^2) - (6.81E-3*N) + 4.7	t_ms(N) = (4.04E-9*N^2) + (6.00E-4*N)



TEST 3:
insert N elements in random order,
access N elements in random order,
delete N elements in random order.

```
for (int i = 0; i < N; i++)
  tree.Insert(randomSetOfNIntegers1[i]);
for (int i = 0; i < N; i++)
  tree.Access(randomSetOfNIntegers2[i]);
for (int i = 0; i < N; i++)
  tree.Delete(randomSetOfNIntegers3[i]);
```

all three randomSetOfNIntegers sets are generated at the beginning of test.cc, they are generated independently, and they each only contain all integers i such that 0 <= i < N

TEST 3 average time for execution on CSIL terminal	**Binary Search Tree**	**AVL Tree**
N=1,000	.4ms	.6ms
N=10,000	7ms	10ms
N=50,000	50ms	67.5ms
approximate quadratic regression	t_ms(N) = (6.97E-9*N^2) - (6.57E-4*N)	t_ms(N) = (8.02E-9*N^2) - (9.56E-4*N)