Self-balancing trees

Bohdan Yeromenko
Faculty of Mathematics and Computer Science, University of Lodz

1. Introduction

We consider, implement and compare Binary Search Tree and AVL-tree.

2. Self-balancing trees

Let's review two self-balancing trees:

2.1 Binary Search Tree

- **Searching:** For searching element 1, we have to traverse all elements (in order 3, 2, 1). Therefore, searching in binary search tree has worst case complexity of O(n). In general, time complexity is O(h) where **h** is height of BST.
- **Insertion:** For inserting element 0, it must be inserted as left child of 1. Therefore, we need to traverse all elements (in order 3, 2, 1) to insert 0 which has worst case complexity of O(n). In general, time complexity is O(h).
- **Deletion:** For deletion of element 1, we have to traverse all elements to find 1 (in order 3, 2, 1). Therefore, deletion in binary tree has worst case complexity of O(n). In general, time complexity is O(h).

2.2 AVL-tree

- **Searching:** For searching element 1, we have to traverse elements (in order 5, 4, 1) = 3 = $\log_2 n$. Therefore, searching in AVL tree has worst case complexity of $O(\log_2 n)$.
- **Insertion:** For inserting element 12, it must be inserted as right child of 9. Therefore, we need to traverse elements (in order 5, 7, 9) to insert 12 which has worst case complexity of $O(\log_2 n)$.
- **Deletion:** For deletion of element 9, we have to traverse elements to find 9 (in order 5, 7, 9). Therefore, deletion in binary tree has worst case complexity of $O(\log_2 n)$.

3. The methodology of tests

First we need to choose the size *n* of the tree will be created. Since the size of both trees is the same - it is set for two at once.

The next step is to choose *m* elements what we will search. It comes by the next formula:
m = 2 + floor(n/4);

Then program generates trees with a given size and inform us about it by a *Trees generated* message.

In the next step program starts iterations in trees and count the time what is needed for each operation (insertion and search).

When the program finishes it work we can see the message:
BS Tree: size = x1, average duration of search time = y1
AVL Tree: size = x2 average duration of search time = y2
BS Tree: size = x1, average duration of insertion time = y3
AVL Tree: size = x2 average duration of insertion time = y4

Where *x1*, *x2* are the size of a tree and *y1*, *y2* are the time in seconds which program needs to search *m* chosen elements. And *y3*, *y4* are time time in seconds which program needs to insert *n* elements.

4. Result & Graph

When randomly filling a tree in the interval calculating with a formula on the line 280.

Where *interval_gomothety* was equal to 0.5, we had following results for searching and inserting:

```
BS Tree: size = 10000, average duration of search time = 1.45627e-005
AVL Tree: size = 10000, average duration of search time = 1.18305e-005
BS Tree: size = 10000, average duration of insertion time = 4.47399e-005
AVL Tree: size = 10000, average duration of insertion time = 7.97929e-005
```

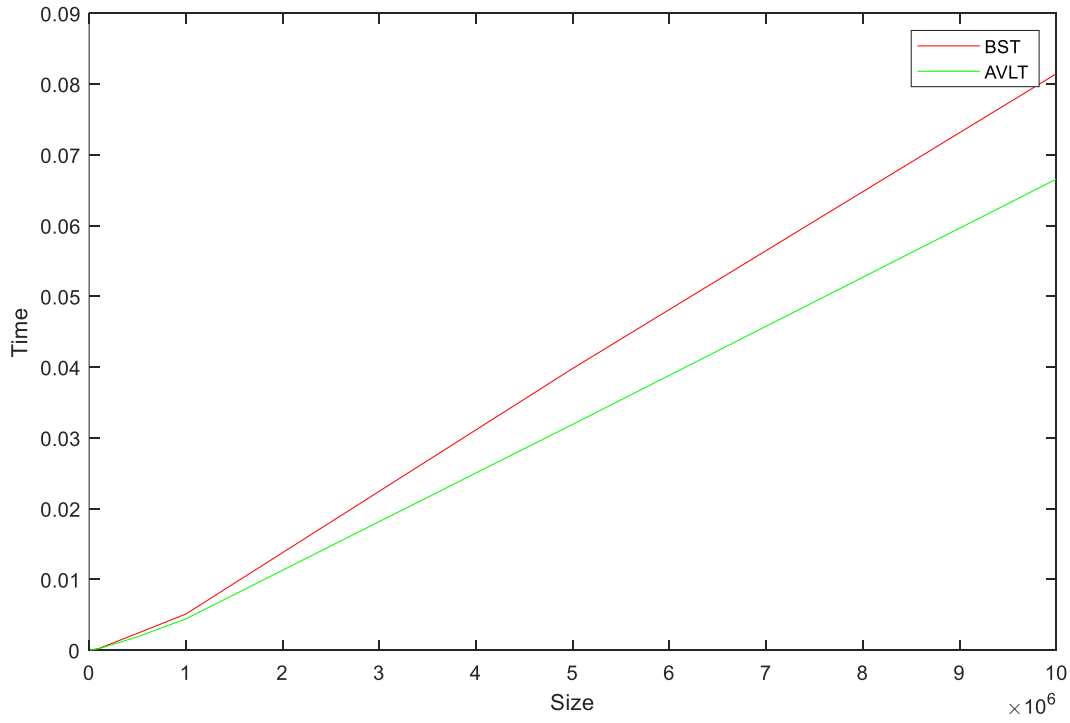
```
BS Tree: size = 50000, average duration of search time = 8.11086e-005
AVL Tree: size = 50000, average duration of search time = 7.61241e-005
BS Tree: size = 50000, average duration of insertion time = 0.000301217
AVL Tree: size = 50000, average duration of insertion time = 0.000535265
```

```
BS Tree: size = 100000, average duration of search time = 0.00020006
AVL Tree: size = 100000, average duration of search time = 0.00018532
BS Tree: size = 100000, average duration of insertion time = 0.000724773
AVL Tree: size = 100000, average duration of insertion time = 0.00117273
```

```
BS Tree: size = 500000, average duration of search time = 0.00164889
AVL Tree: size = 500000, average duration of search time = 0.00143314
BS Tree: size = 500000, average duration of insertion time = 0.00620251
AVL Tree: size = 500000, average duration of insertion time = 0.010469
```

```
BS Tree: size = 1000000, average duration of search time = 0.00420408
AVL Tree: size = 1000000, average duration of search time = 0.00358262
BS Tree: size = 1000000, average duration of insertion time = 0.0162432
AVL Tree: size = 1000000, average duration of insertion time = 0.0260711
```

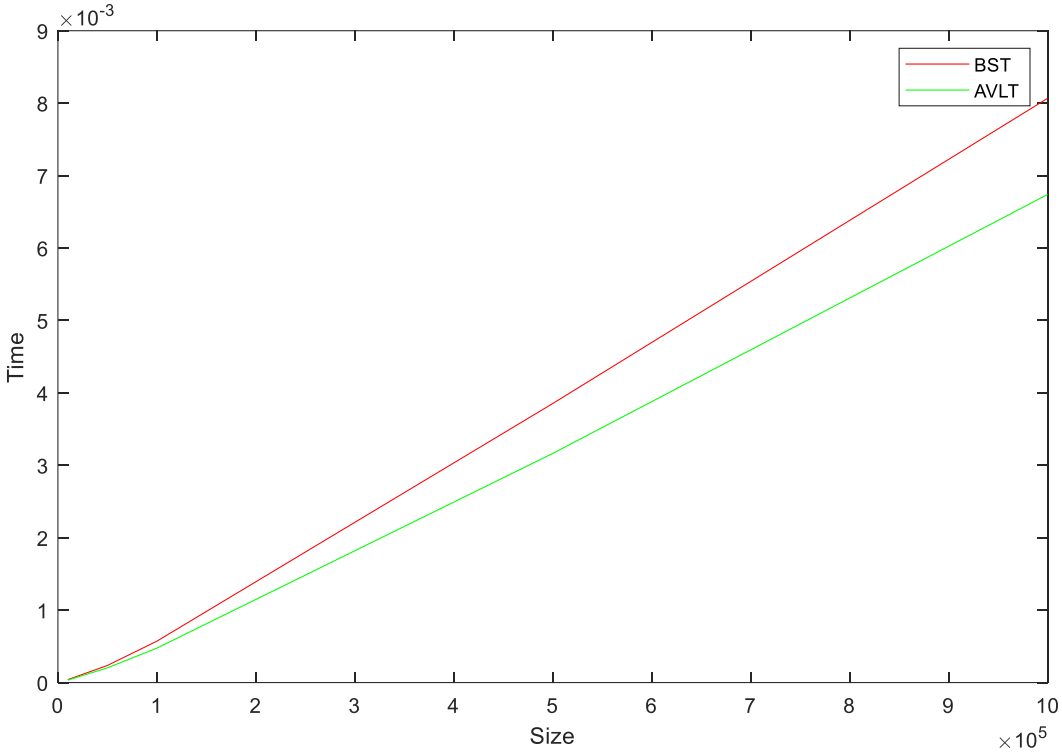
This graph shows the searching time when *interval_gomothety* equals to 0.5



Let’s increase the range of random numbers choosing: now the value of the *interval_gomothety* variable is equals to 8. We have following results for searching and inserting:

```
BS Tree: size = 10000, average duration of search time = 3.21302e-005
AVL Tree: size = 10000, average duration of search time = 2.48615e-005
BS Tree: size = 10000, average duration of insertion time = 9.96966e-005
AVL Tree: size = 10000, average duration of insertion time = 0.000148752
BS Tree: size = 50000, average duration of search time = 0.000176193
AVL Tree: size = 50000, average duration of search time = 0.00015382
BS Tree: size = 50000, average duration of insertion time = 0.000702435
AVL Tree: size = 50000, average duration of insertion time = 0.00104637
BS Tree: size = 100000, average duration of search time = 0.000425687
AVL Tree: size = 100000, average duration of search time = 0.000369738
BS Tree: size = 100000, average duration of insertion time = 0.00169319
AVL Tree: size = 100000, average duration of insertion time = 0.00255667
BS Tree: size = 500000, average duration of search time = 0.00303285
AVL Tree: size = 500000, average duration of search time = 0.00254768
BS Tree: size = 500000, average duration of insertion time = 0.0121386
AVL Tree: size = 500000, average duration of insertion time = 0.0183598
BS Tree: size = 1000000, average duration of search time = 0.00687165
AVL Tree: size = 1000000, average duration of search time = 0.0057507
BS Tree: size = 1000000, average duration of insertion time = 0.0273761
AVL Tree: size = 1000000, average duration of insertion time = 0.0400482
```

And the graph of searching time when *interval_gomothety* variable is equals to 8.



4.1 Questions & answers

Q: Is there a situation in which BST will be faster than AVL?
A: During the tests, there was not a single case of superiority of BST in speed

Q: What is the relation of the amount of elements inserted into the tree to insertion time?
A: We clearly see the relationship between elements inserted and insertion time. So ex example when we increase the number of elements by 5 times, we see an increase in search time by 12 times.

Q: What is the relation of the amount of elements inserted into the tree to search time?
A: As we can see on the graph, when increasing the number of tree elements, the amount of time for searching linearly increases too.

5. Conclusion

We implemented both trees, noted and measured the speed of various operations, such as: inserting elements into the tree and searching for a certain number of elements.

In conclusion we can see that AVL tree provides a bit faster operations than BS tree. Its connected with fact that ALV tree is self-balanced so it gives faster access to its elements.

And these are some words about the theoretical part of these two trees using: AVL trees are intended for in-memory use, where random access is relatively cheap. BS-tree are better suited for disk-backed storage, because they group a larger number of keys into each node to minimize the number of seeks required by a read or write operation.