Binary Search Trees and Complexity

Jack Zezula | Student No.: 808867

Project Repository: http://bit.do/ev3pA

Introduction

This experiment involved testing the theory of the binary search tree (BST) and its processing complexities in practice by utilising them to implement a dictionary, and thereby process a simplified dataset of Olympian records from Kaggle[1].

The dictionary was implemented in two ways, taking Olympian names as keys in each case:

1. With a standard binary search tree, composed of linked nodes, each pointing to a 'left' and 'right' node. Duplicate keys were always inserted to the left of a given node.
2. With a linked-list binary search tree, where each linked node pointed to a 'left', 'right' and 'next' node. The left and right nodes served the same function as in the first implementation. However, duplicate keys were stored in a linked list that stemmed from original node containing a given key.

To test the computational complexity of these dictionary implementations, subsets of the original Kaggle records were parsed through the programs dict1 (standard implementation) and dict2 (linked-list implementation). The number of key comparisons required to search for all instances of a given key in the dictionary was then used to consider the efficiency of each structure.

Method

Data subsets to be processed by each of the programs were generated by taking various sample sizes ranging from 10,000 to 250,000 entries, and creating randomised and ordered (by Olympian name) datasets.

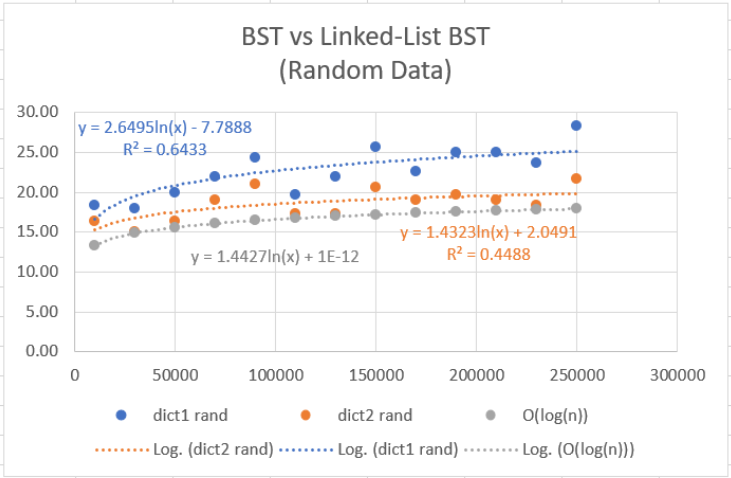
The trends determined between the sample size of a dataset and the number of comparisons required to search for a given key were performed after averaging the number of key comparisons across the three initial datasets[2].

The subsets used to test dict1 and dict2 can be generated with *Inputs/jz_data.py* in the project repository.

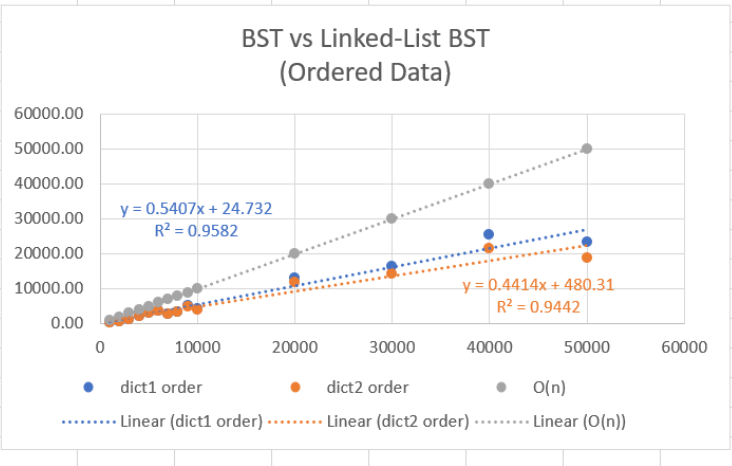
Data

The complete comparison-count datasets used to compute the average comparisons are available in 'Testing/comparisons_summary.xlsx' under the project repository.

Dataset	Average Comparisons		Theoretical
Size	dict1 rand	dict2 rand	O(log(n))
10000	18.33	16.33	13.29
30000	18.00	15.00	14.87
50000	20.00	16.33	15.61
70000	22.00	19.00	16.10
90000	24.33	21.00	16.46
110000	19.67	17.33	16.75
130000	22.00	17.33	16.99
150000	25.67	20.67	17.19
170000	22.67	19.00	17.38
190000	25.00	19.67	17.54
210000	25.00	19.00	17.68
230000	23.67	18.33	17.81
250000	28.33	21.67	17.93



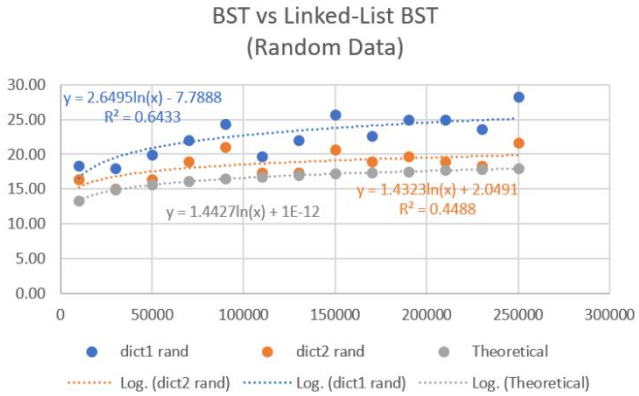
Dataset	Average Comparisons		Theoretical
Size	dict1 order	dict2 order	O(n)
1000	493.00	488.67	1000
2000	766.00	755.67	2000
3000	1403.00	1374.67	3000
4000	2114.00	2068.67	4000
5000	3181.00	3093.00	5000
6000	3701.67	3591.00	6000
7000	2862.33	2748.00	7000
8000	3497.00	3347.67	8000
9000	5132.33	4876.67	9000
10000	4185.33	3975.00	10000
20000	13133.67	11925.67	20000
30000	16414.67	14283.67	30000
40000	25623.33	21450.33	40000
50000	23280.00	18824.67	50000



Comparison and Discussion

1. BST vs Linked-List BST (Randomised Data)

The average complexities would theoretically be in $\Theta(\log n)$), which was supported by the tests of the number of key comparisons required for a given dataset. Using randomised datasets and searching for random keys, both the BST implementations followed a logarithmic comparison complexity, as was expected by the theory.



The standard implementation naturally required more key comparisons than the linked-list version; even when a match had been found between the search key and a given node, the entire left subtree of that node had to be searched exhaustively for duplicate keys.

2. Linked-List BST (Ordered Data)

In theory, the worst-case key complexity should be in $O(n)$, and would occur when each node is only inserted to one side (i.e. left or right) of any given node across the entire tree; the worst-case scenario resembles an expensive (space) array.

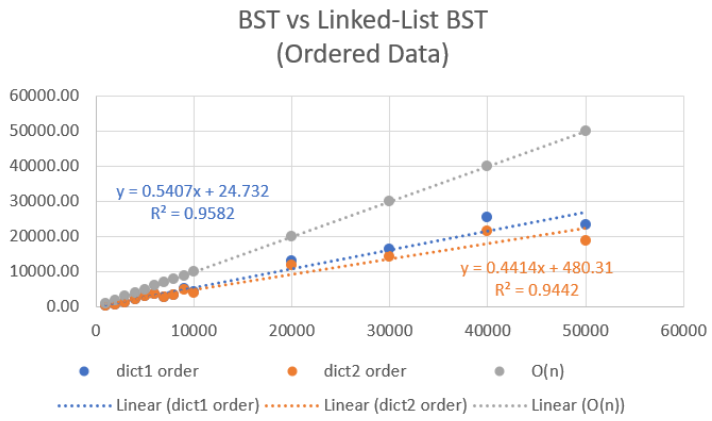
The worst-case complexities of the BST structures were considered by searching for a key inserted last in an ordered dataset. For example: with a dataset containing 10,000 entries ordered by name, searching for the last entry's key "va Kiss" resulted in exactly 10,000 comparisons in dict1, but only 9519 comparisons in dict2.

```
$ ./dict1 Inputs/Data/athlete_events_filtered_order_10000.csv output.txt < "va Kiss"
va Kiss --> 10000

$ ./dict2 Inputs/Data/athlete_events_filtered_order_10000.csv output.txt < "va Kiss"
va Kiss --> 9519
```

The fewer key-comparisons performed by dict2 can be explained by the way the linked-list BST handles duplicates; once a given key is found, all its duplicates can be found without further key comparisons by exhaustively printing the associated linked-list[3].

For ordered datasets, on average both programs tended towards the worst-case complexity of a binary search tree, $O(n)$. This was suggested by the high coefficient of determination ($R^2 > 0.90$) for both dict1 and dict2's regression to a linear trendline.



Summary of Discussion

Comparing the graphs for Ordered Data and Random Data key-comparison tests, it can be seen that:

- Using randomised datasets produces complexities that are lower-bounded by $\log(n)$, suggesting that the search operation on the associated BSTs is in $\Omega(\log(n))$. This supports the theory that complete BSTs have a guaranteed complexity of $\log(n)$, as these randomised datasets produce more complete trees than do the ordered datasets.
- Ordered datasets, on the other hand, produce comparison complexities that are (on-average) bounded by n. These 'average' worst-case scenarios, and the absolute worst-case scenario (i.e. searching for the last key from an ordered dataset) affirm the theory that the search operation is in $O(n)$.
- Using linked lists to store duplicate keys within a BST provides a consistent decrease in search complexity, as the graph of dict1 is lower-bounded by that of dict2 for both types of data (i.e. ordered and randomised). Unrelated to the graphs, however, it is worth considering that this more efficient comparison complexity comes with the cost of requiring more memory; every node must hold one extra 'next' pointer.

Conclusion

Implementing a dictionary with a binary search tree, and testing it with randomised and ordered data was useful in determining the complexity (number of key-comparisons) in average-case and worst-case scenarios respectively. These tests, along with explicit cases of the absolute worst-case scenario, affirmed the theory that binary search trees have associated complexities $\Omega(\log(n))$ and $O(n)$.

---

[1] Original dataset taken from Kaggle: '120 Years of Olympic History: athletes and results'.
[2] These data sets were the files provided on the LMS: *athlete__events_filtered.csv, athlete__events_filtered_alternative.csv, athlete__events_filtered_alternative2.csv*.
[3] This assumes that comparisons to NULL keys are not 'full' comparisons, as suggested by the assignment specification.