CS 584 Research Project

Portland State University

Dylan Laufenberg

June 5, 2018

1 Introduction

This paper aims to illuminate various facets of the relationship between asymptotic complexity and empirical performance of a range of related data structures. The vehicle for this exploration is a series of benchmarks of data structures that perform similar jobs in very different ways. These include binary search trees, treaps, skip lists, and red-black trees. For many tasks, these data structures have the same asymptotic complexity, e.g. average complexity of $O(\log n)$ for search, insert, and delete operations, with some data structures having worst-case complexity of $O(n)$ for each.

The binary search tree in particular has well-known best- and worst-case inputs. Balanced trees like the red-black tree are designed to mitigate these worst-case input scenarios by ensuring a balanced tree structure. The randomized treap and skip list data structures, on the other hand, rely on different randomization techniques to maintain fast expected performance with a tree and a list structure, respectively. These different approaches to potential worst-case inputs will make for interesting comparisons across the board. This paper examines the performance characteristics of each data structure through careful benchmarks and asks how comparable the performance characteristics of these asymptotically equivalent data structures really are.

2 Testing Methodology

Since the analysis in this paper is based on benchmark data, the accuracy of the analysis hinges on the accuracy of the benchmarks themselves. The benchmarks included in this paper utilize the following measures to help ensure their accuracy:

- To minimize the impact of timer error, CPU load spikes, and so on, each data point is the average running time of a large number of operations k, where $k \geq 1,000$.
- The difference in running times between the first and kth insert or delete operations can skew the benchmark results as well. To minimize the impact of such large values of k, the overall sample sizes are suitably large, such that $n \geq 100 \cdot k$, where n is the size of a data structure when the k timed operations begin. In other words, the number of operations being timed is no larger than 1% of the overall data structure at any point.
- When necessary for accuracy, a benchmark may be repeated multiple times, and the results may be combined by taking the median or mean of the running times for each value of n being plotted.
- Since these benchmarks are highly sensitive to fluctuating operating conditions (e.g. CPU scheduling, RAM availability, and Python interpreter behavior), each benchmark sequence is re-run to produce multiple graphs. The most representative graph among the set is chosen for inclusion in this paper.
- To prevent human error in transcribing graphs or plot data, all graphs are generated programmatically using the same functions and included without modification (except to specify each graph's presentation in the report).
- All tests are performed on the same computer, an Intel i7-4930k with 16 GB of DDR3 running at 2133 MHz. The CPU is overclocked beyond its Turbo frequency, which should further stabilize the benchmark results.
- Random samples are chosen by shuffling integers in the range $[0, n)$, where n is the number of samples required. This prevents repeated values, which means that there are no repeated keys within any data structure. This

is important for benchmarking searches and deletes, since they will act on the first matching key they find: multiple keys would skew these benchmarks in potentially unpredictable ways.

Because the data structure implementations considered vary widely in their handling of potential error conditions, the inputs are carefully sanitized. In particular, no test will insert an element that already exists in the data structure, search for an element that is not present in the data structure, or delete an element that is not the data structure. This is particularly important since some implementations raise exceptions in these cases, and catching these exceptions would very likely skew the resulting benchmarks.

3 Project Files

The Python 3 code included with this report is structured as follows:

- datastructures/ — contains the implementations of data structures benchmarked below, one per file, as well as some that were cut from the benchmarks.
- plots/ — contains the output LaTeX figures that the benchmark system produces, ready to \include.
- pgfplot.py — contains the PgfPlot class, which represents one LaTeX figure to be produced. This class receives and stores parameters that affect the figure, including plots to be produced.
- plot.py — contains the Plot and BenchmarkPlot classes, which represent plots in a PGFPLOT graph. The Plot base class may be used to produce arbitrary plots, whereas the BenchmarkPlot subclass receives benchmark parameters for one function and produces a corresponding plot.
- sidgraphset.py — contains the SIDGraphSet and SIDBenchmark classes. The SIDBenchmark class performs a sequence of search, insert, and delete operations on one data structure, producing three corresponding plots. The SIDGraphSet class coordinates a set of data structures, producing a SIDBenchmark for each one and collating the results into three PgfPlots.
- mixedsidgraph.py — contains the MixedSIDPgfPlot and MixedSIDBenchmarkPlot classes. The MixedSIDBenchmarkPlot class performs a sequence of search, insert, and delete operations for one data structure, as specified by the input parameters and produces, one plot of the results. The MixedSIDPgfPlot class runs a MixedSIDBenchmarkPlot for each data structure it is given and produces a PgfPlot of the results.
- benchmark.py — sets up and runs the benchmarks used in this document. See its table of contents for details.
- report.tex — the LaTeX used to produce this report.

The repository should be readily runnable in PyCharm 2018.1 with Python 3. To run custom benchmarks, simply follow the examples in benchmark.py. All classes are thoroughly documented and commented.

4 Data Structure Implementations

The data structures in question are well-known, so many implementations exist. This paper examines the data structure implementations below, marked by the files or subfolders they occupy within datastructures/ in the project files. All credit for each implementation goes to the author of that implementation. All data structures used are cited here. Modifications are annotated in source comments.

- binarysearchtree.py — Dylan Laufenberg (written as a naive reference for these benchmarks)
- toastdriven_pyskip.py — Credit: Daniel Lindsley. Modified from `https://github.com/toastdriven/pyskip`.
- stromberg_treap.py — Credit: Dan Stromberg. Modified from `https://pypi.org/project/treap/`.
- jenks_treap.py — Credit: Grant Jenks. Modified from `http://www.grantjenks.com/wiki/random/python_treap_implementation`. Converted from Python 2 to Python 3.
- pyskiplist/ — Credit: Geert Jansen. Modified from `https://pypi.org/project/pyskiplist/`.
- redblacktree.py — Credit: Dmitry Sysoev. Modified from `https://github.com/dsysoev/fun-with-algorithms/blob/master/trees/redblacktree.py`. (Excluded from benchmarks.)

- avltree.py — Credit: marehr. Modified from `https://github.com/marehr/binary-tree`. (Excluded from benchmarks.)
- enether_rbtree.py — Credit: Stanislav Kozlovski. Modified from `https://github.com/Enether/Red-Black-Tree`.
- sortedlist.py — Credit: Grant Jenks. Modified from `http://www.grantjenks.com/docs/sortedcontainers/`.
- list.py — wrapper around the built-in list data type.

The above data structures have been modified as needed to standardize their interfaces for insert, search, and delete operations and to add `legend_text` class data members to assist with automated legend creation.

5 Choosing Implementations to Test

Note: benchmarks in this section necessarily stray from the testing methodology stated above. The benchmark sizes are (and must be) too small to apply the stated methodology.

The first step in analyzing these data structures is to establish a baseline for their performance. This provides an opportunity to rule out any implementations that are too slow to consider (as well as a disappointing number of incorrect implementations found along the way—not included here). First, a benchmark on element insertion with only a few hundred elements reveals an outlier:

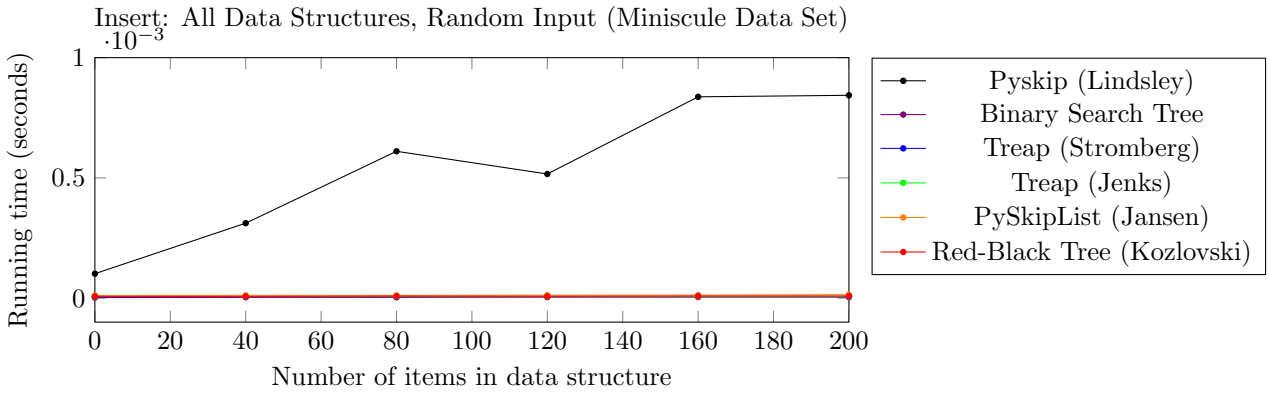

Figure 1: Average of 10 operations, benchmarked every 40, starting at 0.

Clearly Pyskip is too slow for further consideration. In fact, I began with a 105-million-element benchmark that my reference BST implementation handled in about a minute. Pyskip took so long that I eventually had to cancel the benchmark. I had to pare the initial benchmark down to below 1,000 elements to achieve a running time that would not dwarf the rest of my benchmarks combined. With Pyskip eliminated, a graph with larger (but still quite modest) data sets presents a clearer picture of the comparability of the various data structures with random inputs:

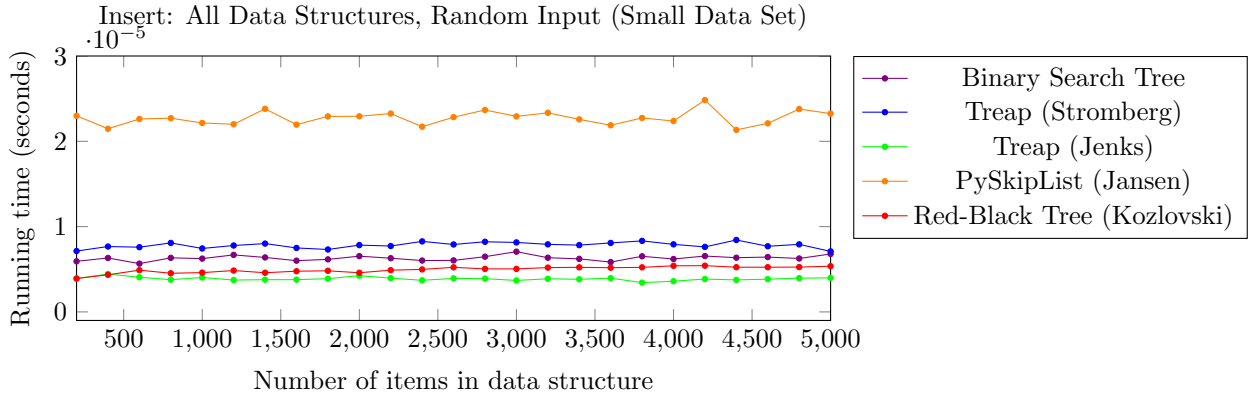

Figure 2: Average of 20 operations, benchmarked every 200, starting at 200.

Based on these benchmark results, the only remaining skiplist implementation is again the clear outlier, but in this case, the difference is much less pronounced. Compared to the binary search tree reference implementation, PySkipList benchmarks on insertion at only about 2.5 times slower, and this, of course, is under ideal conditions for a binary search tree.

Comparing the Stromberg and Jenks treaps, the latter appears to be two or three times faster in this particular benchmark. Interestingly, both are comparable or superior to the binary search tree, and the Jenks treap is faster than both the binary search tree and the Kozlovski red-black tree! Because the treaps provide such interesting counterpoints, both to one another and to the deterministic data structures used here, I will keep both. Thus, the data structures are finalized.

6 Choosing Inputs & Benchmarks

The remainder of this paper examines the above implementations' performance on the insert, search, and delete operations with large data sets (10^6 or larger). The two essential cases are randomized input and ordered input (the binary search tree's worst-case input). Beyond these, I will add as many more novel configurations as space allows.

The SIDGraphSet and SIDBenchmark classes in sidgraphset.py perform the first two benchmark sets. Their sequence, in short, is to perform insertions up to each data point, benchmark a number of searches on selected elements that have already been inserted, benchmark a number of insertions, and repeat. Once all insertions are complete, the sequence continues with deletions: SIDBenchmark will delete elements until it reaches the last graph point plus the sample size, benchmark deletions down to the graph plot point, and repeat this cycle until it benchmarks the first plot point.

The SIDBenchmark class implements this benchmark sequence, and the SIDGraphSet class collates the three plots (search, insert, and delete) of a set of SIDBenchmarks into three final graphs. The full pseudocode for the SIDBenchmark sequence is below, updated for presentation here, along with opening notes copied from the source comments.

```
Instantiates a Plot containing the results of a full suite of search, insert, and
delete benchmarks. Runs these benchmarks immediately and synchronously. Saves the
results as instance variables. Note that this benchmark differs substantially from
BenchmarkPlot, even though its interface is similar!

The benchmark accepts 3 sample sets and maintain 3 indices: one apiece for search,
index, and delete. The data structure on which search, insert, and delete are called is
presumed to be empty prior to this benchmark; the plot points count from 0 elements.
Each index starts at 0.

The sample sets should have the following minimum sizes:
    len(search_samples) >= the number of searches requested
        (likely bm_length times some integer)
    len(insert_samples) >= stop
    len(delete_samples) >= stop

SIDBenchmark(search, insert, delete, search_samples, insert_samples, delete_samples,
             stop, bm_start, bm_length, bm_interval):
0. Set next_benchmark as bm_start.
1. Call insert on the items in insert_samples with indices [0, bm_start).
2. While insert_index < stop:
    a. Set last_benchmark as next_benchmark.
    b. Set next_benchmark as min(next_benchmark + bm_interval, stop).
    c. Benchmark: time bm_length search calls on items in search_samples[search_index],
       incrementing search_index each time. Divide the total time by bm_length
       and save (data structure size, time) as a plot point for insert.
    d. Benchmark: time bm_length inserts on items in insert_samples[insert_index],
       incrementing insert_index each time. Divide the total time by bm_length
       and save (data structure size before timed insertions, time) as a plot point
```

```
       for insert.
    e. Insert subsequent items from insert_samples, incrementing insert_index, until
       insert_index == next_benchmark. These insertions are not timed.
3. Set next_benchmark = last_benchmark + bm_length.
4. While data structure size > bm_start:
    a. Delete items from delete_samples[delete_index], decrementing delete_index,
       until data structure size == next_benchmark.
    b. Benchmark: time bm_length deletions on items in delete_samples[delete_index],
       decrementing delete_index each time. Divide the total time by bm_length
       and save (data structure size after timed deletions, time) as a plot point
       for delete.
    c. Decrease next_benchmark by bm_interval.
```

7 Benchmarks: Randomized Inputs

Random inputs are, of course, the best-case scenario for binary search tree insertions. The other data structures should perform well on randomized inputs, too. As a result, this benchmark serves as a useful baseline for later comparisons. The average-case asymptotic performance for all four data structures is $O(\log n)$ for search, insert, and delete, and I expect the benchmarks to reflect this time complexity on all five implementations with this input set.

Search (Figure 3) is perhaps the most interesting graph, because, surprisingly, all five plots curve to plausibly match the graph of $c \log n$ for various constant multiples c. Clearly, all five do in fact work in logarithmic time! Both treaps are the clear winners for search. PySkipList, the slowest implementation in the small data set benchmark above, is neck-and-neck with the binary search tree. Finally, the red-black tree, the second-fastest on insertion in the small data set benchmark, is the standout slowest here.

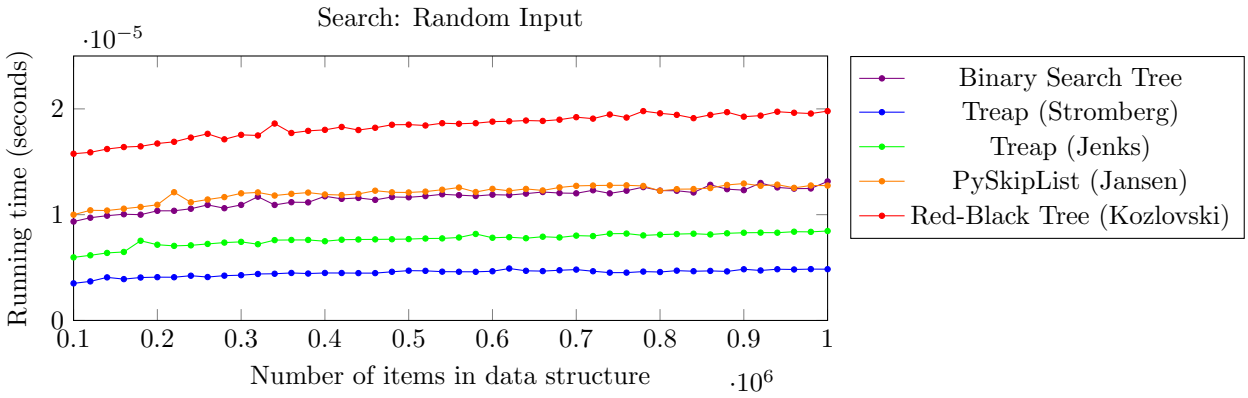

Figure 3: Average of 1000 operations, benchmarked every 20000, starting at 100000.

Next up is insertion (Figure 4). Here, the logarithmic graph is faintly visible at best. Compared to the small data set insertion benchmark, almost everything is the same: the treaps and red-black tree are all tightly grouped, and PySkipList takes about twice as long as the rest. However, in this best-case input for the binary search tree, it does indeed pull ahead of the pack.

Finally, delete performance (Figure 5) is almost entirely flat across all five graphs. It is stratified into two groups. On closer inspection, these are easily explained. Binary search trees and treaps both enjoy relatively simple insertions and deletions, whereas red-black trees and skip lists may require much more work for both.

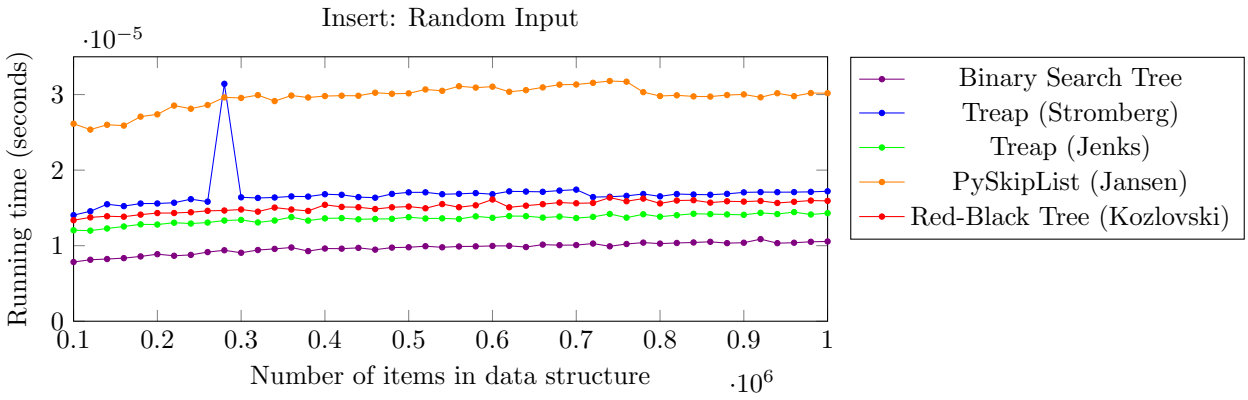

Figure 4: Average of 1000 operations, benchmarked every 20000, starting at 100000.

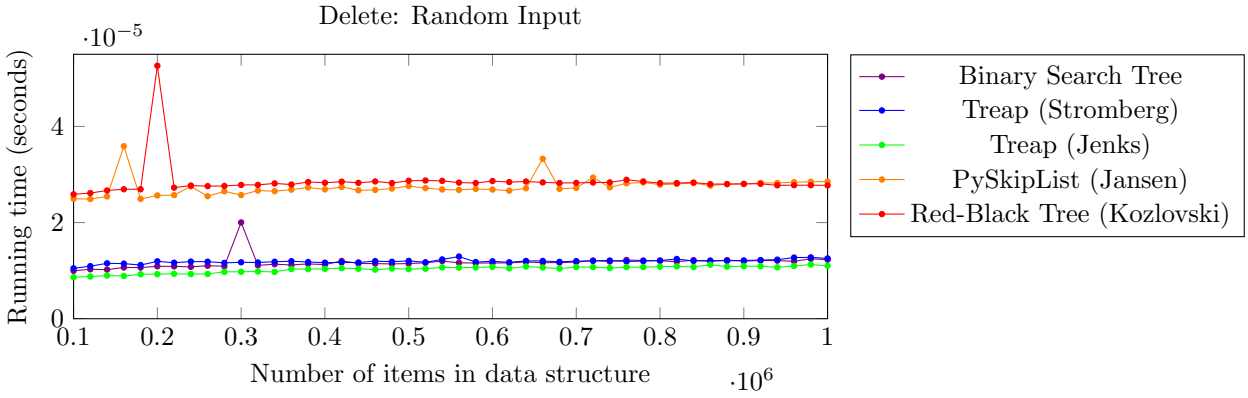

Figure 5: Average of 1000 operations, benchmarked every 20000, starting at 100000.

8 Benchmarks: Ordered Inputs

The worst-case scenarios for binary search trees are inserting ordered data, deleting the farthest leaf, and searching for the farthest leaf in the tree. In any of these cases, the worst-case performance is $O(n)$. Since treaps, skiplists, and red-black trees all claim to run in $O(\log n)$ expected time in these scenarios, these worst-case inputs are the next logical benchmarks. To briefly illustrate the performance difference between the binary search tree and all the other data structures, see the mini-graphs (Figure 6 in which the binary search tree clearly runs in linear time. This is expected and needs no further discussion. (Note that the benchmark parameters for these graphs are scaled down.)

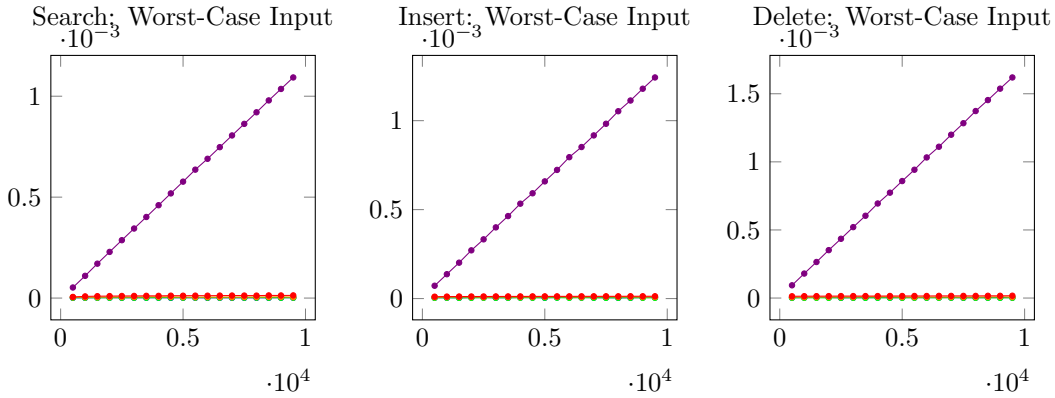

Figure 6: Average of 100 operations, benchmarked every 500, starting at 500.

The more interesting graphs appear when the binary search tree is removed from the data set, as appears below. Here, same worst-case inputs for binary search trees create minor challenges for the various data structures, as they

cope in their own ways with the ordered input data. By removing the binary search tree, the scale of the graphs' y-axes move from 10^{-3} to 10^{-5}, which already demonstrates their shared advantage over the binary search tree.

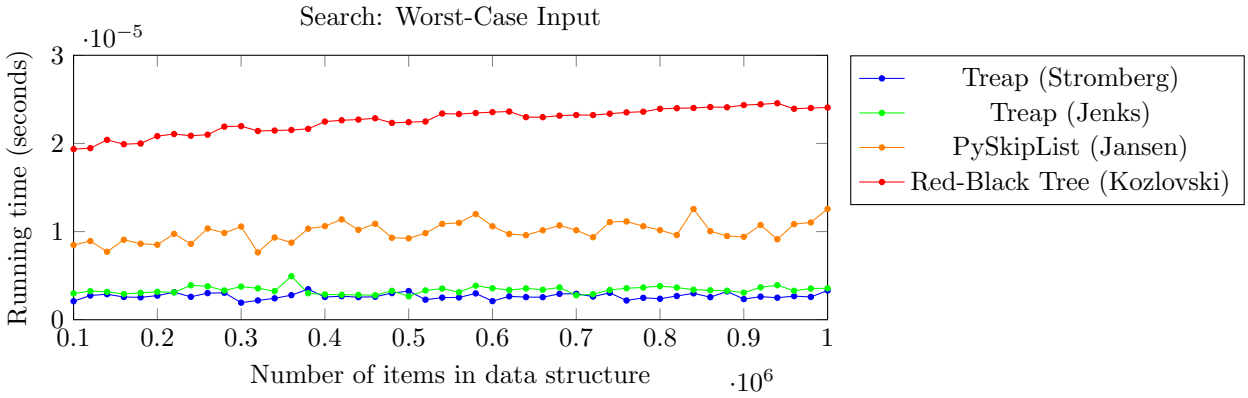

Figure 7: Average of 1000 operations, benchmarked every 20000, starting at 100000.

The first graph, search, presents a great deal more noise than the randomized input search benchmarks. This, I attribute simply to the challenges and inefficiencies inherent in coping with sorted inputs in these data structures. The specifics are surprising and don't seem to have simple explanations. The two treaps perform almost identically, in contrast with the marked performance difference on search with randomized inputs. This suggests that ordered inputs to insert may cause the trees to develop more similar geometries than random inputs. PySkipList is by far the noisiest on search, which suggests a high degree of unpredictability in its performance, even as it maintains logarithmic complexity overall. Lastly, the red-black tree is consistent but much slower than the rest.

The relative speeds of the three data structures remain unchanged: the treaps are fastest, the skiplist is significantly slower, and the red-black tree is slower still. While these are not provable general statements about the three data structures, they do provide insight into the impact of ordered insertions on the performance of searches, which is to say that the data structures are only very modestly affected. Finally, it is worth noting that, compared to search with randomized inputs, the red-black tree and Stromberg treap are both noticeably slower, whereas the Jenks treap and PySkipList remain about the same in their overall speeds.

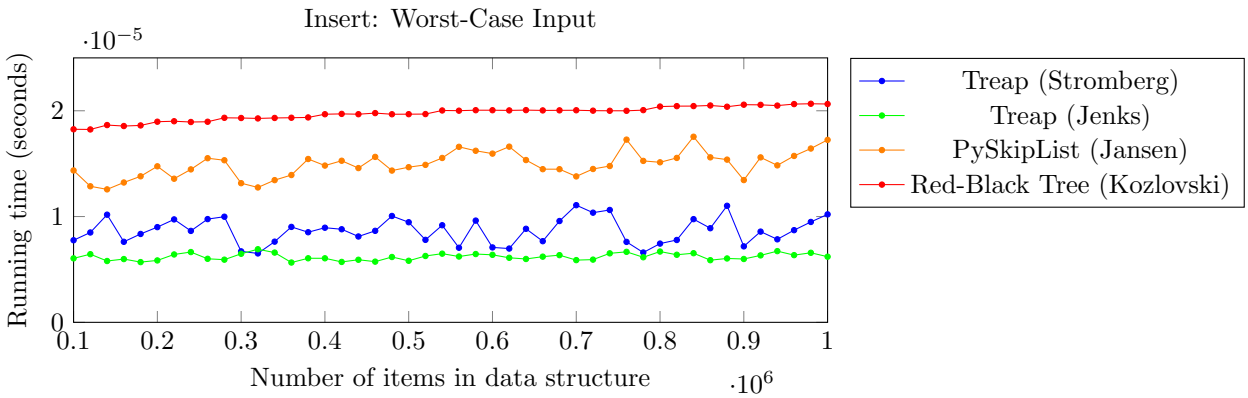

Figure 8: Average of 1000 operations, benchmarked every 20000, starting at 100000.

Insertion of ordered data presents a more interesting picture still. First, the red-black tree, whose performance on random data sat squarely between the two treaps, is now in last place by a wide margin. This makes sense: sorted input is a worst-case scenario for red-black trees, too, because they will need to rotate almost constantly to maintain balance. Yet, it remains clearly and visibly $O(\log n)$ even in this case—an impressive feat. Being deterministic, it exhibits very smooth worst-case performance, but it is indeed far slower than its randomized counterparts.

The three randomized data structures exhibit differing levels of noise, but their relative performance is essentially unchanged compared to the randomized insert benchmark. The most noteworthy aspect of their performance is perhaps the consistency of Jenks relative to Stromberg. While presents wide and erratic variations in running time,

its overall performance remains consistent with its performance on randomized inserts. Meanwhile, Jenks' treap remains quite consistent.

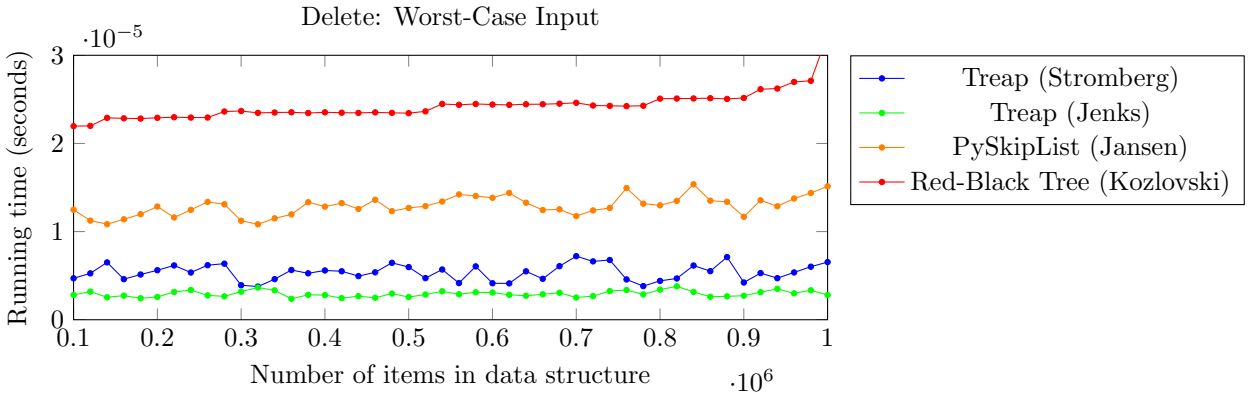

Figure 9: Average of 1000 operations, benchmarked every 20000, starting at 100000.

Delete is much the same story: worst-case but very consistent performance for the red-black tree, while the randomized data structures experience some noise but are still relatively consistent as well. Again, Jenks is much more stable than Stromberg in its running times.

9 Benchmark: Comparing to SortedList with Random Operations

In searching for data structure implementations in Python, I found a plethora of references to the Sorted Containers package and its SortedList class in particular. The advice seemed to be: if you need a sorted data structure, forget trees and use SortedList. It seems fitting, after investigating the performance of these well-known data structures relative to one another, to benchmark them all against the Python-specific data structure commonly used in practice.

The SortedList class leverages Python's very fast implementation of lists (for sizes up to 10,000) by storing a list of sorted sublists that can each be individually managed very quickly. To locate elements, SortedList performs a binary search on the lists' maxima and then performs a second binary search on the appropriate sublist. SortedList is designed with modern hardware in mind: the creator, Grant Jenks, observes that "a lot of time has been spent optimizing mem-copy/mem-move-like operations both in hardware and software."[1]

This final benchmark is structured somewhat differently than the rest. It aims to provide a different perspective on the data structures examined so far. For this benchmark, each data structure is subjected to the same, randomly generated sequence of search, insert, and delete operations. As before, no duplicates are inserted, no absent elements are searched for, and no absent elements are deleted. This benchmark does not aim to mimic a real-world workflow;

[1] `http://www.grantjenks.com/docs/sortedcontainers/implementation.html`

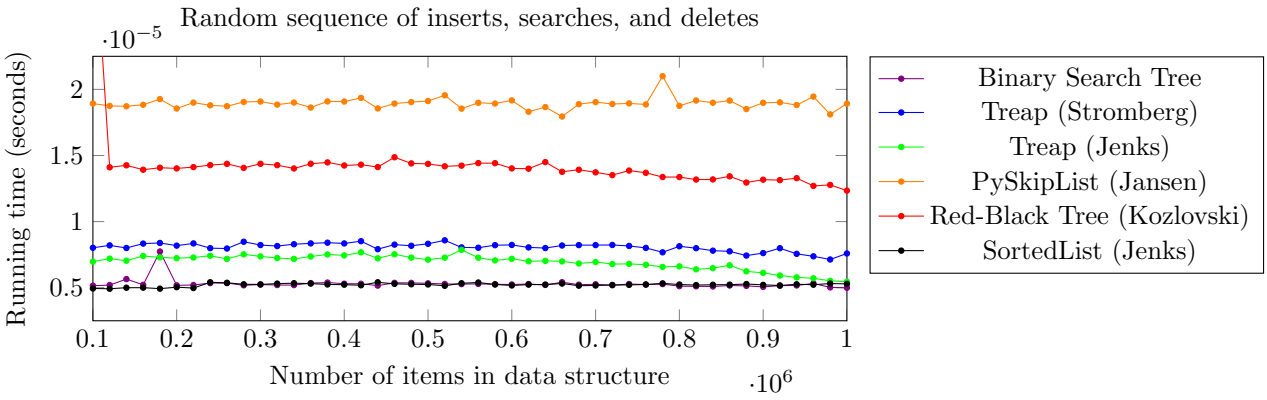

Figure 10: Average of 1000 operations, benchmarked every 20000, starting at 100000.

it simply aims to put the data structures through their paces in a less static way and observe their performance under varied conditions.

First, perhaps the most obvious result is that SortedList is tied for top performance in terms of running time across the board. What shocks me is that the naive binary search tree, which I wrote without any attempt to optimize it, runs just as fast under this randomized workload! (SortedList very likely has superior space complexity, and this is a considerable advantage in practice. However, it is beyond the scope of this paper.)

One of CS 584's core messages is that algorithms with better asymptotic performance will generally beat any other optimizations by leaps and bounds. The worst-case benchmarks certainly and trivially verified binary search trees' vulnerability to $O(n)$ worst-case performance and, more interestingly, the relative ease with which all other tested data structures handle said input in $O(\log n)$ time. For random data, however, the algorithmic simplicity of the binary search tree seems to beat out the treaps, skiplist, and red-black tree. Meanwhile, despite inferior asymptotic complexity on insert and delete, SortedList ties for lowest running time on mixed operations. This seems to vindicate Jenks' approach and demonstrates the tremendous power of designing data structures to take advantage of modern hardware.

Looking back briefly at the random input benchmarks for search, insert, and delete, the only constants are that the treaps consistently perform well and PySkipList consistently runs substantially more slowly than the treaps. The red-black tree and binary search tree both vary in their rankings relative to the other data structures. The three graphs for single operations on random inputs foreshadow some, but certainly not all, of the characteristics of this combined benchmark. Most clearly: the treaps again come in with respectable running times but sit squarely behind the fastest data structures. PySkipList and the red-black tree each suffer greatly on certain operations in prior benchmarks, so it is unsurprising to find that they are much slower than the treaps (and, since these data are randomized, the binary search tree).

In practice, SortedList does appear to be a highly competitive data structure. It undeniably bests all but the binary search tree in this test. Were I to continue testing, I would wish to investigate SortedList's worst-case performance, however. It will very likely suffer under one of the same worst-case inputs as binary search trees, reverse-sorted input data: it will presumably copy all the data in the target list on every insert. Unlike the binary search tree, however, SortedList is designed so that its worst-case behavior is highly optimized on modern hardware. My open question: how much will hardware optimization cushion SortedList's performance hit in this worst-case?

10 Concluding Remarks

The reality of how slow $O(n)$ can be compared to $O(\log n)$ hit me clearly during the worst-case benchmarking of the binary search tree. In CS 584, we consider any polynomial time algorithm to be feasibly runnable, but for these admittedly very large input sizes, even linear-time algorithms brought my very powerful workstation to a grinding halt with a single binary search tree benchmark sequence that, under ideal circumstances, had taken about a minute to complete. In reflecting on this—which I had ample time to do during a two-hour, worst-case benchmark involving Python's beloved built-in lists—I recognized that by inserting n items into these data structures, I was effectively multiplying the overall theoretical complexity of my benchmarks by n. This explains why the blazingly fast $O(\log n)$ operations nonetheless took around a minute to complete. It further explains why going from 10^{-5} to 10^{-3} seconds for a given operation increased my overall benchmarking time by a factor of 30 or more—a frustrating reality I had to manage throughout my testing. In effect, my worst-case tests had time complexity $O(n^2)$ with $n = 1,000,000$.

I find all of the above results interesting for academic purposes and for certain applications. However, I can't ignore the 11 idle logical cores on my CPU during each of the benchmarks I ran! There is a place for single-threaded data structures, to be sure, but I would very much like to explore these same questions using data structures or methods that would allow me to more fully utilize the actual computational power of my hardware to more quickly complete the tasks at hand.

It's worth noting that an implementation by Dmitry Sysoev of the textbook's red-black tree algorithm would have made for an even more interesting counterpoint: on insert, with random data, it had proved to be the fastest algorithm. Unfortunately, its deletion method was fatally flawed, partly due to design decisions by the textbook authors that made their algorithm very difficult to implement correctly. I was not able to fix the Sysoev implementation in the time available for this project, so I excluded it from the analysis instead. My partially-fixed implementation is available in my repository as redblacktree.py in the data structures folder.