Tal Derei

CSE 411: Assignment 0
Set Implementations

1. Brief summary of competitor: only include important details that distinguish each one.

[1] Set -- associative containers of the C++ standard library which stores elements in a sorted-manner. All elements contained within a set are unique and cannot be modified, but can be inserted and/or removed. Sets are typically implemented as balanced binary search trees (BST), like red-black trees. BSTs do NOT support random access, and require a custom searching algorithm std::find() as a targeted solution for sets.

Time Complexity: Search Time: O(log n) | Insertion Time: O(log n) + rebalancing | Deletion Time: O(log n) + rebalancing

[2] List -- sequence containers that allow non-contiguous memory allocation. Compared to a vector, list-traversals are slow (linear-time), but once a position has been found, insertions and deletions are quick. Lists are typically implemented as doubly-linked lists. And unlike sets, lists can contain duplicate elements.

Time Complexity: Insertion Time: O(1) | Deletion Time: O(1) | Search Time: O(n)

[3] Sorted Vector -- dynamic arrays with the ability to resize itself automatically when elements are inserted and deleted. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators, and their storage is handled automatically by the container. Vectors can contain duplicate elements. Sorted vectors accelerate lookups with binary search as opposed to linear search employed in unsorted vectors, but add extra cost on the insert/remove operations.

Time Complexity: Insertion Time: O(1) + rebalancing | Deletion Time: O(1) + rebalancing | Search Time: O(log n)

[4] Unsorted Vector -- same as sorted vectors, except the unsorted version makes insert/remove operations faster, but pays an extra cost of “linear search” during lookups.

Time Complexity: Insertion Time: O(1) | Deletion Time: O(1) | Search Time: O(n) unsorted

2. Experimental Setup

[1] Fixed Parameters: The number of iterations (i.e. operations) remained constant at 1M for each trial run, independent of the data structure. The number of trials per data structure (set, list, sorted vector, unsorted vector) was 100: 5 average/variance trial runs per key size for each

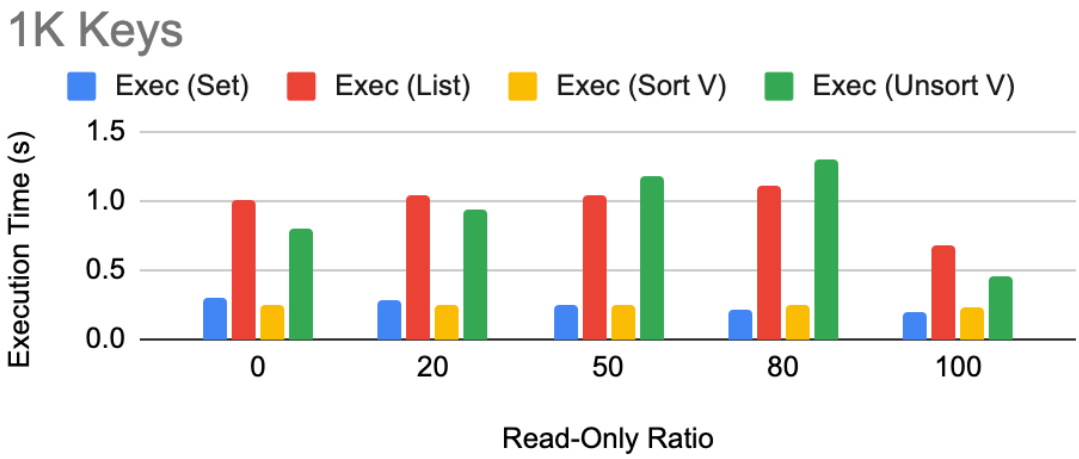
read-only ratio. For example, 5 trials for a key size of 1k with read-only-ratio of 0%, 5 trials for a key size of 1k with read-only-ratio of 20%, etc.

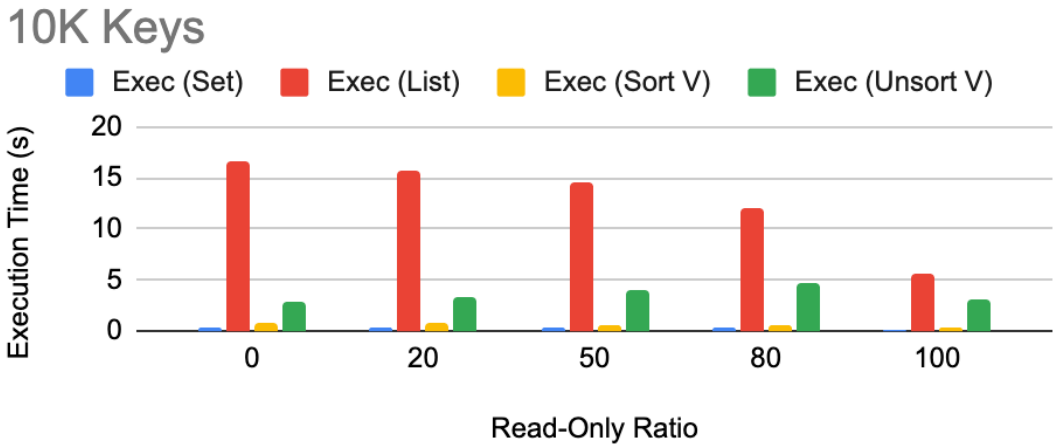
[2] Test Configurations: key sizes (1k, 10k, 100k, 1M), read-only-ratios (0%, 20%, 50%, 80%, 100%)

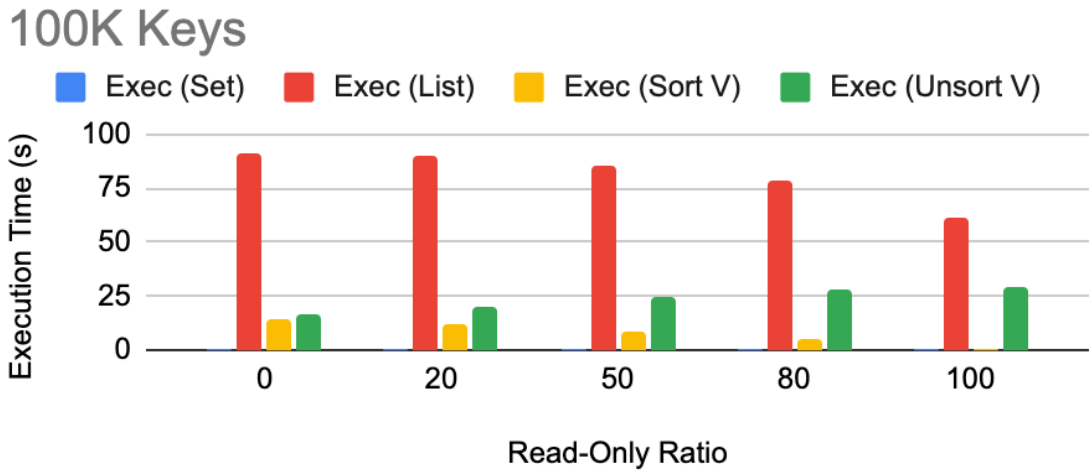
3. Plots

Read-Only	Exec (Set)	Exec (List)	Exec (Sort V)	Exec (Unsort V)	Key-Size	Operations
0	0.301679	1.01231	0.247	0.797222	1K	1M
20	0.287069	1.03364	0.252514	0.944899		
50	0.254782	1.04732	0.249607	1.17312		
80	0.224207	1.10294	0.243879	1.2965		
100	0.204481	0.688326	0.225663	0.457613		
Read-Only	Exec (Set)	Exec (List)	Exec (Sort V)	Exec (Unsort V)	Key-Size	Operations
0	0.34776	16.5387	0.771928	2.77436	10K	1M
20	0.332721	15.7715	0.738823	3.23808		
50	0.29657	14.5581	0.653484	4.02061		
80	0.262867	11.9477	0.524224	4.77138		
100	0.241836	5.62429	0.255869	3.09313		
Read-Only	Exec (Set)	Exec (List)	Exec (Sort V)	Exec (Unsort V)	Key-Size	Operations
0	0.421592	90.5783	14.0306	16.5522	100K	1M
20	0.393709	90.5271	12.4089	19.902		
50	0.350784	85.6801	9.16367	24.7413		
80	0.324044	79.0187	5.02796	28.7294		
100	0.306661	61.1496	0.287495	28.9682		

[SEE NEXT PAGE]


1K Keys
Exec (Set)
Exec (List)
Exec (Sort V)
Exec (Unsort V)
Execution Time (s)
1.5
1.0
0.5
0.0
0
20
50
80
100
Read-Only Ratio


10K Keys
Exec (Set)
Exec (List)
Exec (Sort V)
Exec (Unsort V)
Execution Time (s)
20
15
10
5
0
0
20
50
80
100
Read-Only Ratio


100K Keys
Exec (Set)
Exec (List)
Exec (Sort V)
Exec (Unsort V)
Execution Time (s)
100
75
50
25
0
0
20
50
80
100
Read-Only Ratio

4. Analysis

In regards to the appropriate configuration for each competitor (i.e. the data structure), it seems like the set was particularly efficient for a large number of accesses (inserts and removals) and look up operations. My tree outperforms the others with tree size at 1,000,000 and the others with only size 1,000. In regards to the sorted and unsorted vectors, It seems like the unsorted vector had faster inserts/removals (amortized constant time), but longer lookups (linear) time complexities as compared with my sorted vector that had faster lookups (logarithmic) and slower inserts and removals. I also didn't implement any adaptors like stacks or queue/deque...could that be a potential bottleneck that's bringing down my performance, or is the performance degradation bounded by the linear searching of these data structures as the key sizes grow exponentially?