Evaluation of Different Implementations of Set Data Structure via Benchmark on CI Pipeline

Bonor Ayambem

CSE-411: Advanced Programming Techniques

Prof. Corey Montella

September 14, 2023

1. Introduction

The selection of the right data structure can significantly impact the efficiency and performance of an application. Different data structures, such as hash sets, tree sets, array sets, and list sets, are designed to excel in specific scenarios, and understanding their behavior under various conditions is crucial for making informed engineering decisions.

In this benchmark analysis, the performance of four distinct data structures: HashSet, TreeSet, ArraySet, and ListSet, are explored and compared. These tests aim to shed light on how these data structures behave when subjected to different sizes of data sets and various ratios of find, insert, and remove operations. This examination covers a range of scenarios, from smaller data sets with a few thousand elements to more substantial collections exceeding one million entries. Moreover, the distribution of operations vary, ranging from read-heavy workloads to balanced and write-heavy tasks, to comprehend how each data structure responds to different usage patterns.

2. Implementation Summaries

2.1 Dynamic Array Implementation

The insert() and remove() methods in this set implementation make calls to the insert and remove functions in the Vec standard library. remove() and find() loop through the indices of the vector and call remove() from the standard library, in the case of the remove() implementation, and return true.

Overall, the biggest difference between this implementation and the others is the ease of accessing elements using their index.

2.2 Linked List Implementation

Listset is implemented using the LinkedList standard library.

insert() adds a new element to the set by calling the push_back() method in the standard library.

find() calls the contains() function

remove() makes use of an iterator since the linked list cannot access elements via index or value. If the current element is equal to the number being removed, the remove() method from the standard library is called on the for loop variable which acts as the current index. remove() from the standard library is a nightly-only experimental API and so it required nightly installation, and the addition of crate attribute #![feature(linked_list_remove)] to the root module (lib.rs)

Overall, the biggest difference between this set implementation and the others, is the insertion of elements only to the front and back of a linked list, limiting access to the elements not added either first or last.

2.3 Balanced Binary Search Tree Implementation

`Treeset` is implemented as a struct with two fields: `set` and `count`. Set is a BTreeMap instance with i32 key and value types. Because BTreeMap stores key-value pairs, count is instantized as 0 and is incremented with each set entry. It acts as the value in the map, while the entries themselves are the keys. The count field itself does not have an impact on any functionality.

insert() is implemented using the insert() method in std::collections::BTreeMap. BTreeMap.insert() returns an Option type—it returns None if the number being added is not present as a key in the map, and the old value associated with the key if the number is present in the map. Because all possibilities have to be handled with the Option type, the returned old value, which has no real consequence, just returns false from the function. None returns true, that is, the number was not already in the map.

remove() in the BTreeMap standard library, similarly returns an Option type which is matched to true or false depending on whether or not the entry already existed in the set as a value.

find() is simply a call to the contains_key() method in the BTreeMap standard library.

Overall, the biggest difference between this implementation and the other set implementations, is the storing of entries as key-value pairs.

2.4 Hash Table Implementation

The implementations of insert(), remove(), and contains() in std::collections::HashSet perform the same function as those required for these implementations of insert(), remove(), and find() respectively. Therefore, the functions in this assignment are simply calls to those contained in the standard library.

3. Testing Setup

The set implementations were tested using different configurations for their sizes and percentage of find operations. Each set was benchmarked for every combination of the sizes: 1K, 10K, 100K, 1M; and the read-only ratios 0%, 20%, 50%, 80%, 100%. The leftover operations from the read-only operations were split evenly between the insert and remove operations.

The number of operations for each bench was fixed at 1,000. At first, 100,000 operations were attempted on each bench, followed by 10,000. In each instance, the time required to complete all 80 benches (4 set implementations x 5 read-only ratios x 4 sizes) was greater than the scope of this project.

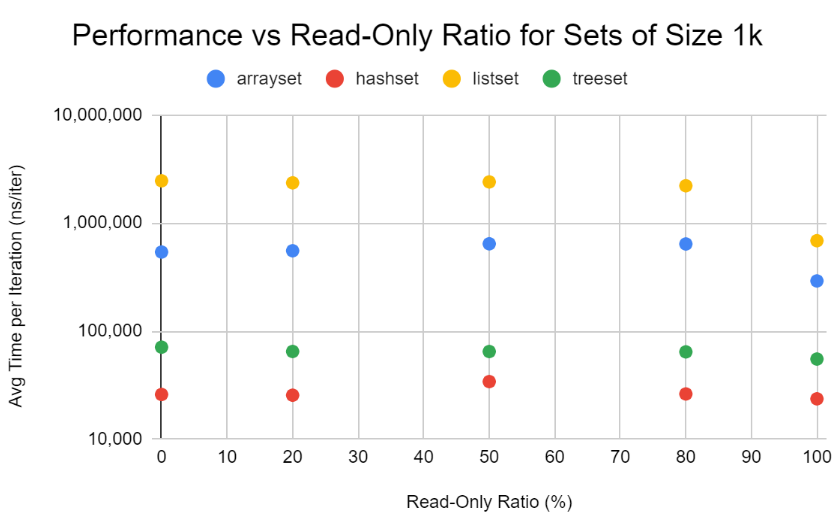
Each bench test began by filling up the set to 50% of its size, and then randomly selecting an operation between find, read, and remove, 1,000 times. Of the total randomly selected operations, find operations made up whatever the specified read-only ratio was for that bench.

4. Results

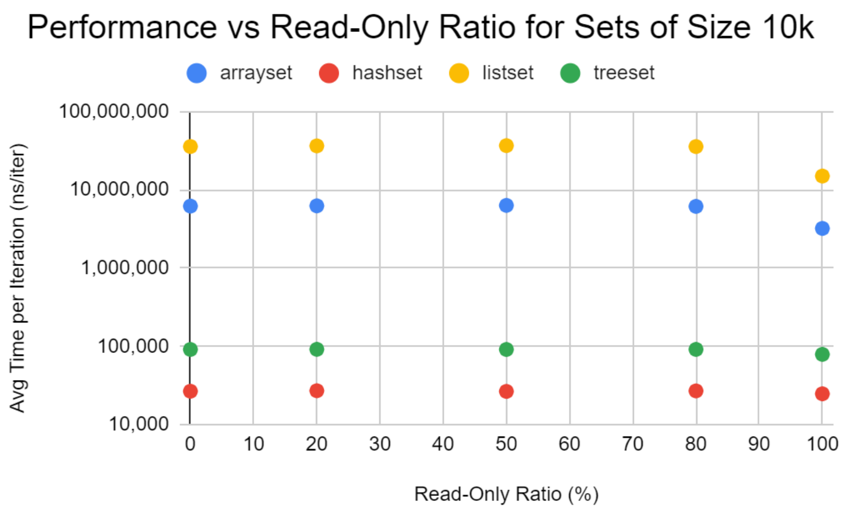
The results of the different implementations are compared based on their sizes and read-only ratios. When size is kept constant, we can observe how the different read-only ratios affect the performance of each set implementation. Similarly, with constant read-only ratios, we can see how the different sizes impact the performances of each implementation. Overall, we can make conclusions about which set implementations have the best and worst performances.

4.1 Constant Sizes

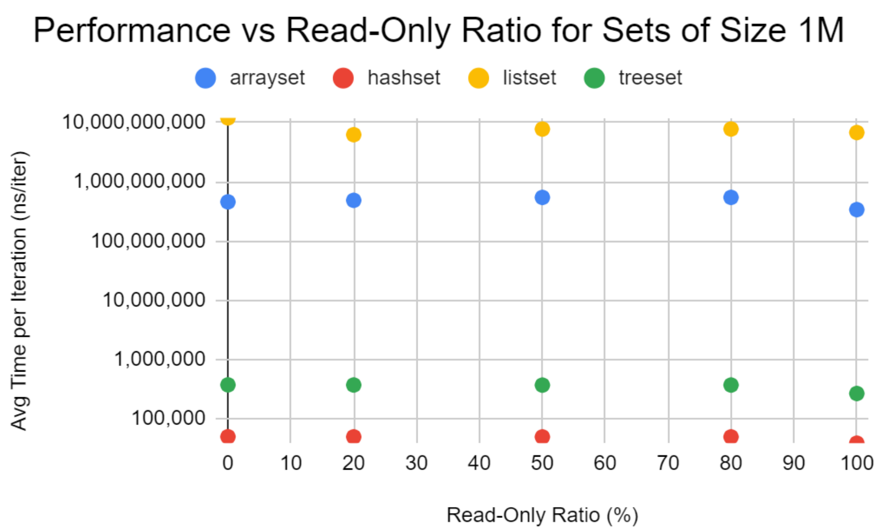
For a specific size, the performance of a set does not change much across the find-ratios for a set. The exception to this is the 100% find ratio which sees an increase in the speed per iteration.



Hashset generally performs the best, followed by treeset, then arrayset. Listset does the worst. The patterns remain for all the sizes, but the numbers themselves get larger to indicate that larger sets lead to higher average times per iteration, which aligns with the added complexity of managing larger data structures

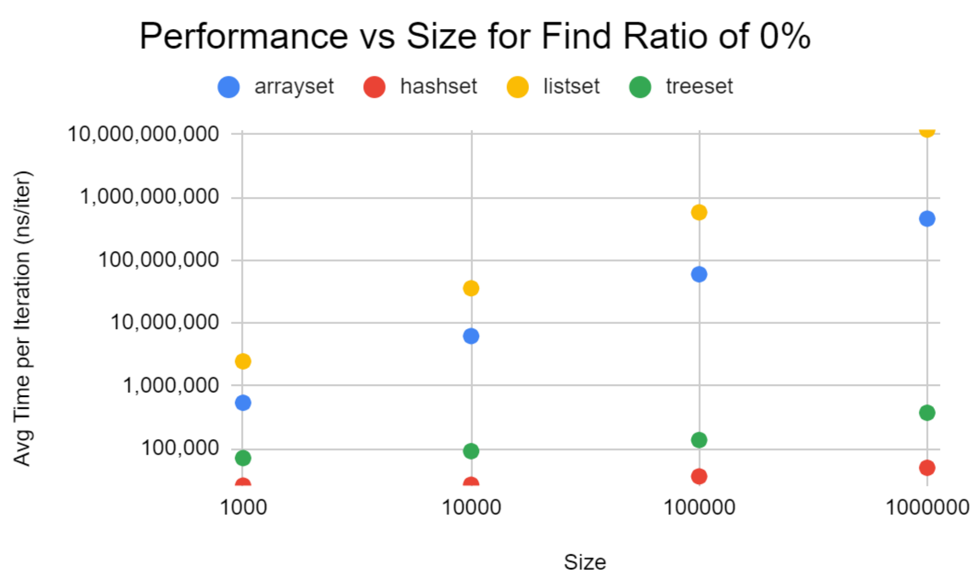


When the ratio is closer to 100, indicating a higher proportion of find operations, the average times per iteration tend to be lower. This suggests that data structures optimized for read-heavy workloads perform better in these cases. When the ratio is balanced (e.g., 50), where find, insert, and remove operations are evenly distributed, the performance is intermediate. When the ratio is lower (e.g., 0), indicating fewer find operations and more insert/remove operations, the average times per iteration are higher.

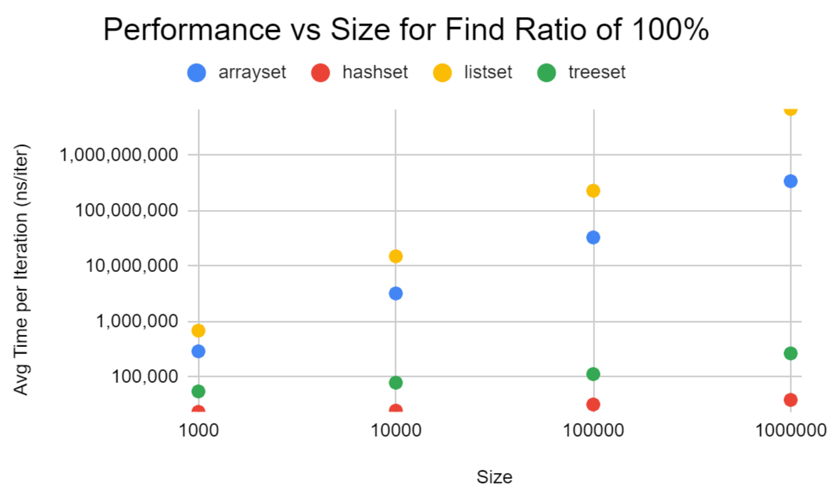


4.2 Constant Read-Only Ratios

For a specific read-only ratio, the performance of a set continues to do worse as the size increases.



Again, we can observe that hashset performs the best, followed by treeset, arrayset, then listset. Larger sets (e.g., 100k and 1m) tend to have higher average times per iteration compared to smaller sets (e.g., 1k and 10k). This is expected, as larger sets require more time to perform operations, and this trend holds true for all four data structures.



5 Conclusions

HashSet generally performs the best in these benchmark tests. It offers stable and competitive performance across various scenarios. ArraySet and ListSet perform competitively with HashSet, especially in smaller set sizes and read-heavy workloads. However, their performance may degrade with larger sets.

Regardless of the data structure, larger sets lead to higher average times per iteration, which aligns with the added complexity of managing larger data structures. The choice of data structure should consider the specific use case and workload. Some data structures may excel in certain scenarios but perform poorly in others.

```
running 80 tests
test arrayset_100k_0   ... bench:  59,672,350 ns/iter (+/- 18,529,272)
test arrayset_100k_100 ... bench:  32,817,540 ns/iter (+/- 6,238,121)
test arrayset_100k_20  ... bench:   6,279,920 ns/iter (+/- 884,205)
test arrayset_100k_50  ... bench:  63,418,000 ns/iter (+/- 8,968,709)
test arrayset_100k_80  ... bench:  63,512,950 ns/iter (+/- 18,761,556)
test arrayset_10k_0    ... bench:   6,203,910 ns/iter (+/- 581,921)
test arrayset_10k_100  ... bench:   3,217,590 ns/iter (+/- 627,444)
test arrayset_10k_20   ... bench:   6,247,830 ns/iter (+/- 2,302,586)
test arrayset_10k_50   ... bench:   6,331,200 ns/iter (+/- 1,375,191)
test arrayset_10k_80   ... bench:   6,150,585 ns/iter (+/- 1,499,426)
test arrayset_1k_0     ... bench:     539,416 ns/iter (+/- 78,596)
test arrayset_1k_100   ... bench:     291,657 ns/iter (+/- 28,210)
test arrayset_1k_20    ... bench:     554,510 ns/iter (+/- 160,743)
test arrayset_1k_50    ... bench:     642,971 ns/iter (+/- 145,847)
test arrayset_1k_80    ... bench:     640,585 ns/iter (+/- 35,287)
test arrayset_1m_0     ... bench: 457,144,120 ns/iter (+/- 207,224,031)
test arrayset_1m_100   ... bench: 338,286,880 ns/iter (+/- 76,742,999)
test arrayset_1m_20    ... bench: 485,449,870 ns/iter (+/- 174,563,538)
test arrayset_1m_50    ... bench: 543,508,220 ns/iter (+/- 179,986,632)
test arrayset_1m_80    ... bench: 544,280,290 ns/iter (+/- 206,540,841)
test hashset_100k_0    ... bench:      36,391 ns/iter (+/- 15,550)
test hashset_100k_100  ... bench:      32,135 ns/iter (+/- 6,688)
test hashset_100k_20   ... bench:      26,793 ns/iter (+/- 17,126)
test hashset_100k_50   ... bench:      34,979 ns/iter (+/- 8,179)
test hashset_100k_80   ... bench:      36,301 ns/iter (+/- 10,032)
test hashset_10k_0     ... bench:      26,827 ns/iter (+/- 8,774)
test hashset_10k_100   ... bench:      24,886 ns/iter (+/- 7,248)
test hashset_10k_20    ... bench:      27,228 ns/iter (+/- 7,099)
test hashset_10k_50    ... bench:      26,697 ns/iter (+/- 5,744)
test hashset_10k_80    ... bench:      27,095 ns/iter (+/- 7,272)
test hashset_1k_0      ... bench:      25,991 ns/iter (+/- 6,249)
test hashset_1k_100    ... bench:      23,689 ns/iter (+/- 7,420)
test hashset_1k_20     ... bench:      25,581 ns/iter (+/- 8,153)
test hashset_1k_50     ... bench:      34,192 ns/iter (+/- 17,945)
test hashset_1k_80     ... bench:      26,277 ns/iter (+/- 9,635)
test hashset_1m_0      ... bench:      49,872 ns/iter (+/- 17,633)
test hashset_1m_100    ... bench:      38,840 ns/iter (+/- 10,323)
test hashset_1m_20     ... bench:      49,654 ns/iter (+/- 14,451)
test hashset_1m_50     ... bench:      49,531 ns/iter (+/- 11,518)
test hashset_1m_80     ... bench:      49,573 ns/iter (+/- 14,053)
test listset_100k_0    ... bench: 576,761,350 ns/iter (+/- 94,098,871)
test listset_100k_100  ... bench: 227,879,530 ns/iter (+/- 55,788,615)
```

```
test listset_100k_20    ... bench:  37,220,250 ns/iter (+/- 12,851,753)
test listset_100k_50    ... bench: 566,239,450 ns/iter (+/- 94,990,547)
test listset_100k_80    ... bench: 542,935,970 ns/iter (+/- 61,397,952)
test listset_10k_0      ... bench:  35,750,790 ns/iter (+/- 6,170,164)
test listset_10k_100    ... bench:  15,000,510 ns/iter (+/- 2,704,045)
test listset_10k_20     ... bench:  36,430,230 ns/iter (+/- 3,654,001)
test listset_10k_50     ... bench:  36,711,720 ns/iter (+/- 4,789,674)
test listset_10k_80     ... bench:  35,701,770 ns/iter (+/- 5,636,227)
test listset_1k_0       ... bench:   2,459,102 ns/iter (+/- 317,912)
test listset_1k_100     ... bench:     686,770 ns/iter (+/- 188,754)
test listset_1k_20      ... bench:   2,354,762 ns/iter (+/- 594,300)
test listset_1k_50      ... bench:   2,401,025 ns/iter (+/- 385,229)
test listset_1k_80      ... bench:   2,212,852 ns/iter (+/- 1,070,855)
test listset_1m_0       ... bench: 11,925,911,250 ns/iter (+/- 5,054,306,178)
test listset_1m_100     ... bench: 6,755,817,290 ns/iter (+/- 3,222,293,495)
test listset_1m_20      ... bench: 6,210,838,590 ns/iter (+/- 262,797,443,430)
test listset_1m_50      ... bench: 7,735,651,510 ns/iter (+/- 2,406,660,634)
test listset_1m_80      ... bench: 7,749,860,790 ns/iter (+/- 2,982,693,058)
test treeset_100k_0     ... bench:     137,915 ns/iter (+/- 47,673)
test treeset_100k_100   ... bench:     113,298 ns/iter (+/- 6,063)
test treeset_100k_20    ... bench:      91,962 ns/iter (+/- 27,097)
test treeset_100k_50    ... bench:     136,139 ns/iter (+/- 50,395)
test treeset_100k_80    ... bench:     131,331 ns/iter (+/- 41,319)
test treeset_10k_0      ... bench:      91,656 ns/iter (+/- 4,020)
test treeset_10k_100    ... bench:      79,326 ns/iter (+/- 26,226)
test treeset_10k_20     ... bench:      91,994 ns/iter (+/- 4,417)
test treeset_10k_50     ... bench:      91,630 ns/iter (+/- 32,100)
test treeset_10k_80     ... bench:      91,812 ns/iter (+/- 10,244)
test treeset_1k_0       ... bench:      71,168 ns/iter (+/- 16,735)
test treeset_1k_100     ... bench:      55,270 ns/iter (+/- 1,960)
test treeset_1k_20      ... bench:      64,925 ns/iter (+/- 10,426)
test treeset_1k_50      ... bench:      64,867 ns/iter (+/- 1,623)
test treeset_1k_80      ... bench:      64,371 ns/iter (+/- 3,355)
test treeset_1m_0       ... bench:     374,240 ns/iter (+/- 46,856)
test treeset_1m_100     ... bench:     267,387 ns/iter (+/- 74,073)
test treeset_1m_20      ... bench:     372,176 ns/iter (+/- 47,817)
test treeset_1m_50      ... bench:     370,490 ns/iter (+/- 96,103)
test treeset_1m_80      ... bench:     371,450 ns/iter (+/- 95,214)
```