- Summary:
  - Overall

    lib.rs defines a trait called Set, which is implemented by each of the four data structures in an effort to allow me to avoid having to write separate bench and main functions for each type. Each Set must define its own find, insert, and remove functions. Each Set is generic. It also defines an enum called SetType which can be passed into said functions to specify the type of the set being passed in.

    main.rs uses Clap to parse command line arguments. It works exactly as laid in the instructions. It takes the number of operations (i), the max key value (k), the read-only ratio (r), and the data structure (d). It will create and populate the requested Set. It will then run i operations on the set. Roughly r% of the operations will find(), and the remaining will be split between insert() and remove(). It will then report the elapsed time between the conclusion of the set population and the end of the operations.

    test.rs runs several simple unit tests on each Set to check correctness. It uses the default rust testing framework.

    The project uses the Criterion crate as its benchmark harness. Criterion is much more widely used, better supported, more stable, and more feature rich than the default benchmarking harness. I highly recommend it. It will run each benchmark as many times as it can in a user-defined amount of time and sample the data it collects before producing the final results. This means that the statistics collected by Criterion are more accurate because the result is closer to the "true" average. It also compares the performance of the current state of the benchmark to the prior state. It even generates summaries and plots for each benchmark because it logs each run. I did not end up using these features, but they would be quite useful in many cases.

    The pipeline fails because the runner has no space. I tried creating my own runner, but I was unable to verify it because I do not have sufficient permissions. If you look back at my first pipeline, you can see that it ran perfectly well, however it failed because the test took over one hour.
  - ArraySet

    The ArraySet is backed by a standard vector. find() uses the Vector::contains() function. insert() matches on find and pushes the value on false. remove() iterates over the vector and removes the index of the found value in place.
  - HashSet

    The HashSet is backed by the standard HashSet. find() uses the HashSet::contains() function. insert() uses the HashSet::insert() function. remove() uses the HashSet::remove() function. This set has the thinnest wrapper over its underlying data structure.
  - ListSet

    List set is backed by the standard LinkedList. find() uses the LinkedList::contains() method. insert() matches on find() and uses the

push_back() method on false. remove() iterates over the linked list and uses the unstable LinkedList::remove() function to remove it.

- TreeSet

  TreeSet is backed by the standard BTreeMap structure. find() uses BTreeMap::contain_key(). insert() uses BTreeMap::insert(). remove() uses BTreeMap::remove().

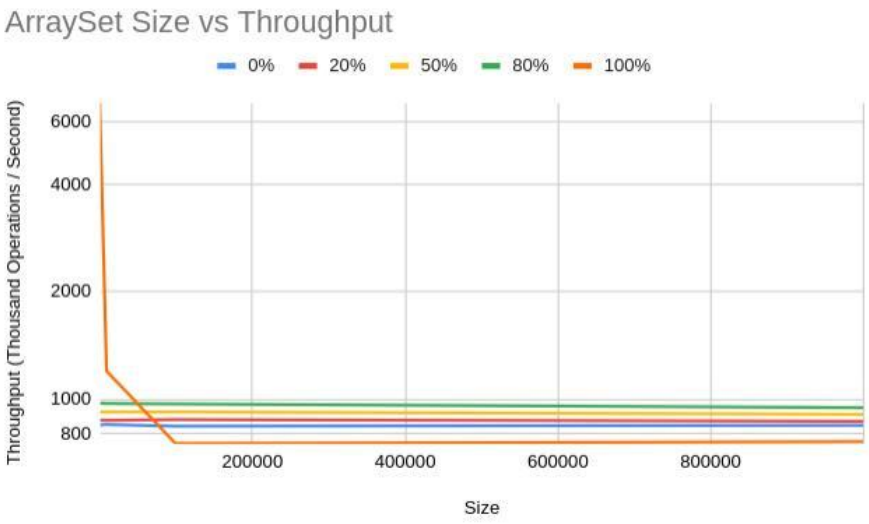
- Benchmark Setup

  set_bench.rs uses the Criterion testing harness. I and K are fixed at 10000. For each data structure for each r, one benchmark is run with the size (s) set to 1000, 10000, 100000, and 1000000. For each data structure for each s, one benchmark is run with r set to 0, 20, 50, 80, and 100. That makes 80 benchmarks. Each run of each benchmark will create and populate the requested Set. It will then run i operations on the set. Roughly r% of the operations will find(), and the remaining will be split between insert() and remove(). Criterion runs each benchmark as many times as it can in 1-5 seconds and then takes a random sample of 10, averages them, and reports back. Criterion is awesome. In a production testing environment, it would be better to take a larger sample, run each benchmark for a greater duration, increase i and increase k. However, it is clear from my data that these numbers suffice.

- Plots and Conclusions

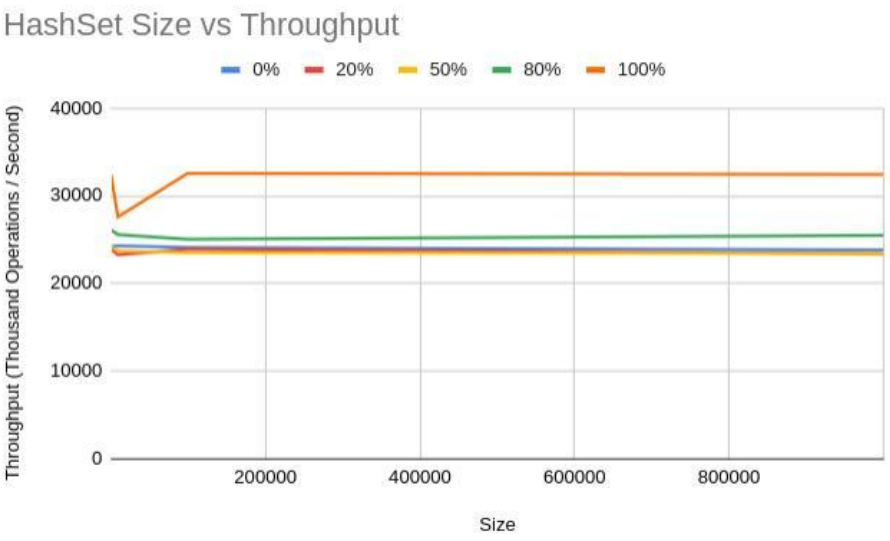
  Each plot shows size vs throughput at each readonly ratio. i and k are fixed at 10000.

  - ArraySet



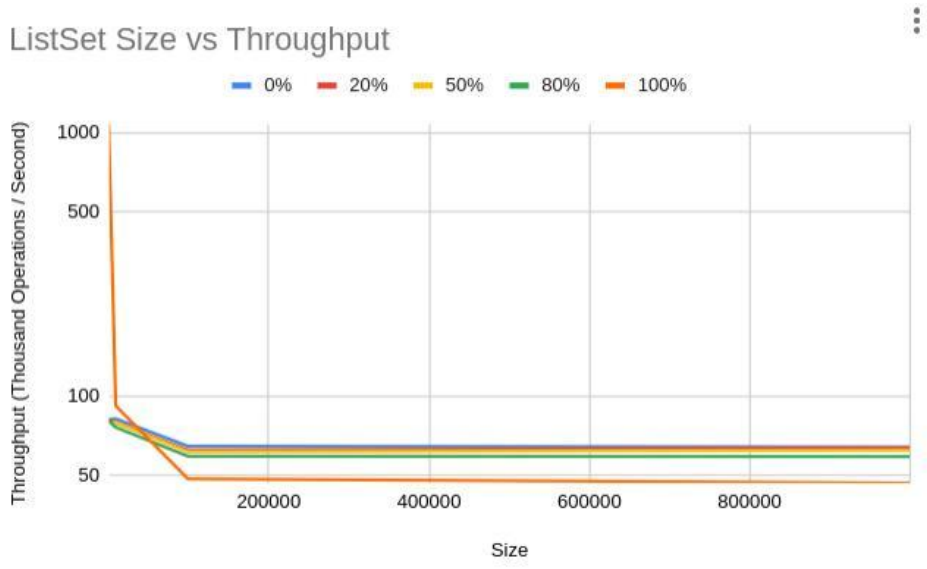
The ArraySet throughput is higher than ListSet but lower than HashSet and TreeSet. The plot shows that performance slightly improves as the read-only ratio increases until it reaches 100%. Size appears to have no impact on throughput until r reaches 100%. At 100%, the throughput starts off being significantly higher for small sizes, but as size increases, performance drops dramatically.

- HashSet



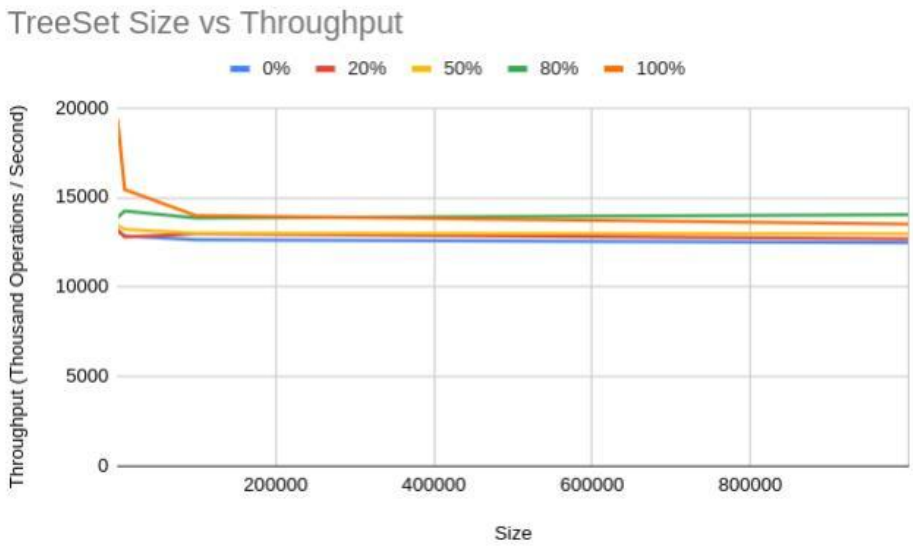
The HashSet throughput is higher than ListSet and ArraySet, but lower than TreeSet. The throughput difference between the read-only ratios improves consistently as it increases. The size appears to make little to no difference in throughput.

- ListSet



ListSet is easily the worst performing and also the most inconsistent data structure. Performance across read-only rations is fairly consistent, which makes sense due to the nature of linked lists. Size appears to have little effect on throughput.

- TreeSet



Tree set is the most performant data structure. Size has little effect on throughput across read-only ratios. The throughput goes up as read-only ratio increases in most cases.