Comparisons of Ordered Set Implementations in C++

CMPT 353 - Fall 2024

Naoto Kuwayama, [redacted]
Mekdim Dereje, [redacted]
Denise Siu, [redacted]

Introduction

Motivation

The C++ ecosystem offers multiple implementations of ordered sets, each built upon different custom underlying data structures to suit specific needs. However, there is a lack of a comprehensive comparative analysis between different implementations of these ordered sets. This makes it challenging for developers to decide which implementation to use in their everyday code. We want to determine the optimal ordered set implementation in C++ across various use cases.

Purpose

The purpose of this project is to conduct a comprehensive performance analysis of a small sample of the most commonly used C++ ordered set implementations. We plan to test them under different operations and data characteristics, with the goal of providing empirical evidence for selecting the most efficient implementation based on specific use cases. For this project, we define efficiency in terms of the fastest execution for each case given we can reject the null hypothesis for each combination. See the data collection section for more details.

Problems

- Which ordered set would be the most efficient for insertion, lookup, and deletion?
- Which ordered set would be the most efficient with sorted, reverse sorted, or random data?

The main reason for considering the order of data is the behaviour of data structures may be different. For example, certain data structures, like balanced trees, require rebalancing, which can impact performance. However, the rebalancing frequency may vary based on the order in which data is inserted or removed. Similarly, cache efficiency can be influenced by data order as well. For instance, traversing data in order increases the likelihood that the next element is already cached.

Data Collection

We will generate our data by implementing a minimal version of an ordered set that accepts inputs, and we will benchmark the different ordered sets in different scenarios. Our analysis will examine performance across various data characteristics for each ordered set implementation. These characteristics include performance with sorted data, reverse-sorted data, and completely random data. We will also interact with several operations. The three operations we have chosen are insertion, value lookup, and deletion. These three are commonly used in the context of ordered sets thus making them the most valuable for us to compare in our analysis.

Configurations

We will be testing the following ordered set implementations:

Library	Data Structure	Source
std::set	Red-Black Tree	cppreference
boost::avl_set	AVL Tree	boost
absl::btree_set	B-Tree	abseil
boost::container::flat_set	Ordered Vector	boost
boost::splay_set	Splay Tree	boost

Test Scenarios

We have crafted ten unique test scenarios covering various interesting real-world use cases which we will use to run benchmarking tests, for each of the ordered set implementations. For consistency, the data we will be using to run these tests will be an array of randomly generated and uniformly distributed data containing `10,000` values called `random_data`. Each of these scenarios will be tested 100 times with each ordered set implementation to get a fair sample size of data.

Case #	Operation	Setup	Experiment
1	insert()	Initialize an empty set.	Sort `random_data` in ascending order and use it to populate the set.
2	insert()	Initialize an empty set.	Sort `random_data` in descending order and use it to populate the set.
3	insert()	Initialize an empty set.	Leave `random_data` unsorted and use it to populate the set.
4	find()	Populate the set with `random_data`.	Sort `random_data` in ascending order and search for each value in the set.
5	find()	Populate the set with `random_data`.	Sort `random_data` in descending order and search for each value in the set.
6	find()	Populate the set with `random_data`.	Search for each value in `random_data` which we will leave unsorted.

7	find()	Populate the set with random even values.	Search for random odd numbers, which are not present in the set.
8	delete()	Populate the set with `random_data`.	Use `random_data` sorted in ascending order and delete each value from the set.
9	delete()	Populate the set with `random_data`.	Use `random_data` sorted in descending order and delete each value from the set.
10	delete()	Populate the set with `random_data`.	Use `random_data` unsorted and delete each value from the set.

Implementation

Benchmarking an ordered set involves two key stages: generating the data and executing the benchmark for each data structure.

Generating Data

We generated data within a specified range using a fixed seed to ensure reproducibility for benchmarking. More specifically, we created 10,000 data points ranging from 0 to 1,000,000 inclusive. To create randomness in the data we utilized the Mersenne Twister 19937 engine to generate the data. However, due to this randomness, some duplicate data was created, resulting in 9,527 unique elements. This dataset was consistently used across all experiments except for Scenario 7 and was sorted or reverse-sorted for their respective scenarios after generation.

For Scenario 7, we maintained the same seed and data range but split the data equally by evens and odds instead of using the entire dataset to ensure we had non-existent values.

Running the Benchmark

Initially, we define `setupTasks` and tasks as described earlier. In addition to the data structure being evaluated, we include a reporter that logs individual benchmark results to a CSV file.

The `run` function executes the benchmark with a set number of iterations (in our case, 100) to collect our samples. Running the benchmark multiple times mitigates the risk of warm-up effects. When the program is started from scratch, it may initially take longer to execute the same task. This delay is due to various factors, such as the absence of an instruction cache. So to increase the stability of the benchmark, we collected data over several iterations.

The `setup` function manages the execution of setup tasks. For each iteration, we did three things: execute the setup tasks, execute the tasks we wanted to benchmark, and report the result. Our setup function takes the targeted data structure for that iteration and inserts the setup task into it. Additionally, it is important to note that we do not measure the time taken by the setup tasks, as this process is not relevant to our benchmarking objectives.

We surrounded the tasks we wanted to measure, such as insertion, removal, and lookup, by the `clock_gettime` function. This function combined with the `CLOCK_PROCESS_CPUTIME_ID` argument measures the amount of CPU time our enclosed piece of code took to execute. We chose to use `CLOCK_PROCESS_CPUTIME_ID` as it is preferable to other clock types since it accurately captures the CPU resources utilized by the process itself, excluding any time the process may be in a waiting queue or is being preempted by other processes.

Data Analysis

Our analysis will use the Kruskal-Wallis test to compare the different implementations of the C++ ordered sets and find statistically significant differences between them. Subsequently, we will perform the Dunn's test as our post hoc to determine which means are significant from each other. Using the results from Dunn's test, if the means are significantly different, which we define as the `p-value < 0.05`, we will compare the means of the run times for the two ordered set implementations and use that information to determine which implementation is faster.

Data Cleaning and Transformations

We were able to filter and visualize the data with a scatter plot, histogram, and boxplot to evaluate what would need to be done. Although there were outlier points in the data, they were kept as it would be untruthful to remove them without cause. Next, we had to normalize the data. Upon attempting to normalize with several transformations, our data was extremely right-skewed and unable to pass the normal test. Despite having equal variances with the Levene test, this meant we could not use the ANOVA test. After resolving that, with the loose requirements of our non-parametric tests, the data did not have to undergo much cleaning or noise filtering since the tests would simply take the ranked sum means.

Analysis Techniques

Our initial approach was to use the Analysis of Variance (ANOVA) test for our comparisons to find statistical differences combined with Tukey's Honest Significant Difference (HSD) test as our post hoc test for pairwise comparisons. Our data was unable to be normalized leading us to pivot to the Kruskal-Wallis test which was able to handle comparisons between more than two groups and did not have the limitation of requiring normally distributed data.

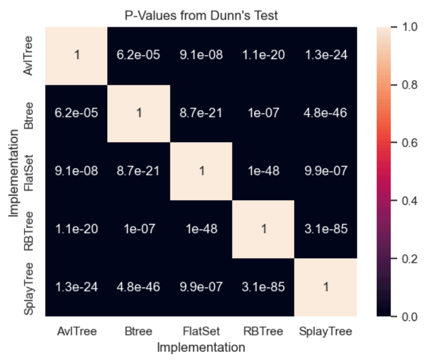
For our post hoc analysis, we looked at several methods such as the pairwise Mann-Whitney tests with Bonferroni correction and the Conover-Iman test, but ultimately we decided on the Dunn's test. Despite Dunn's test being known as one of the least powerful and a relatively conservative test, we went forward with it for its common pairing with the Kruskal-Wallis test and its simplicity.

Results

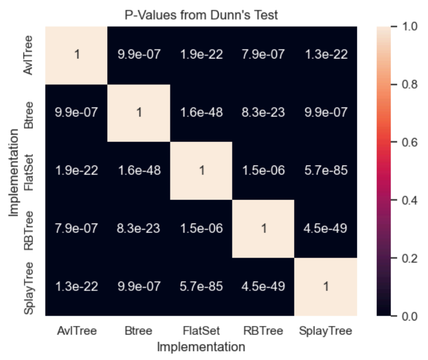
Findings

After performing the Kruskal-Wallis test followed by the Dunn's test we ranked each of the implementations for each of our ten cases based on their mean time. The combinations of implementations that failed to reject the null hypothesis due to a high p-value in our Dunn's test are listed as ranking in the same position.

Case 1: Insertion of Sorted Data

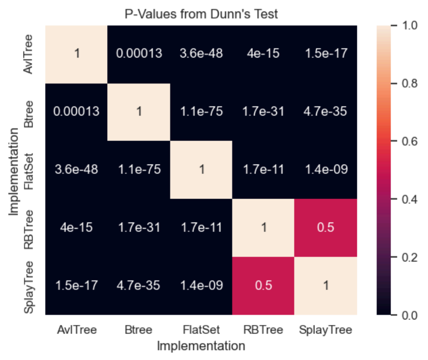
Ranking	Implementation	Mean Time (s)	
1	Splay Tree	0.0000522225	
2	Flat Set	0.0001578125	
3	AVL Tree	0.0003288275	
4	B Tree	0.0003821753	
5	Red Black Tree	0.0008416333	

Case 2: Insertion of Reverse Sorted Data

Ranking	Implementation	Mean Time (s)	
1	Splay Tree	0.0000520688	
2	B Tree	0.0002301987	
3	AVL Tree	0.0003160967	
4	Red Black Tree	0.0008006721	
5	Flat Set	0.0031257388	

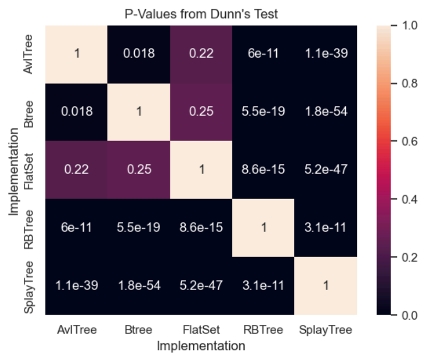
Case 3: Insertion of Unsorted Data

Ranking	Implementation	Mean Time (s)
1	B Tree	0.0006709288
2	AVL Tree	0.0006870455
3 & 4	Splay Tree	0.0011038880
3 & 4	Red Black tree	0.0011942617
5	Flat Set	0.00173228201



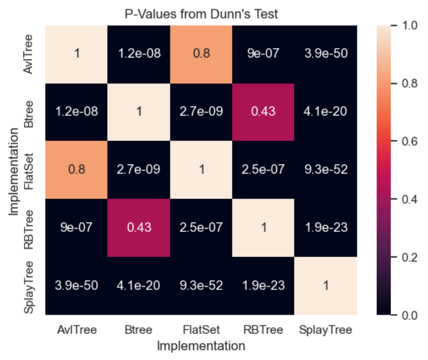
Case 4: Lookup of Sorted Data

Ranking	Implementation	Mean Time (s)
1 & 2	Flat Set	0.0000002238
1 & 2	B Tree	0.0000002467
3	AVL Tree	0.0000003059
4	Red Black Tree	0.0000004113
5	Splay Tree	0.0001349000

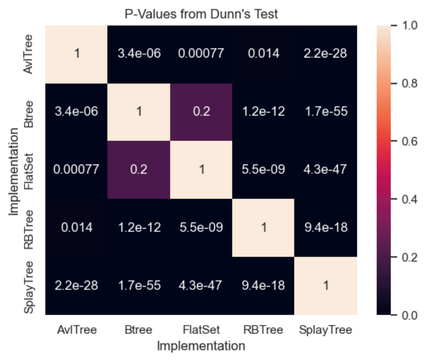


Case 5: Lookup of Reverse Sorted Data

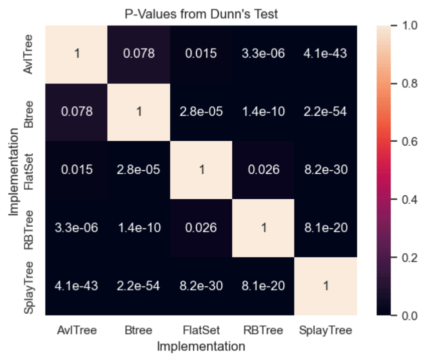
Ranking	Implementation	Mean Time (s)
1 & 2	AVL Tree	0.0000003013
1 & 2	Flat Set	0.0000003358
3 & 4	B Tree	0.0000004666
3 & 4	Red Black Tree	0.0000005029
5	Splay Tree	0.0001381571



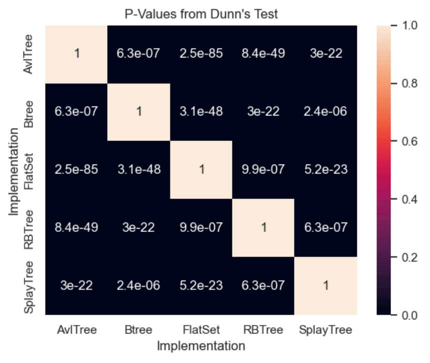
Case 6: Lookup of Unsorted Data

Ranking	Implementation	Mean Time (s)	
1 & 2	B Tree	0.0000002387	
1 & 2	Flat Set	0.0000003451	
3	Red Black Tree	0.0000004233	
4	AVL Tree	0.0000011108	
5	Splay Tree	0.0010638475	

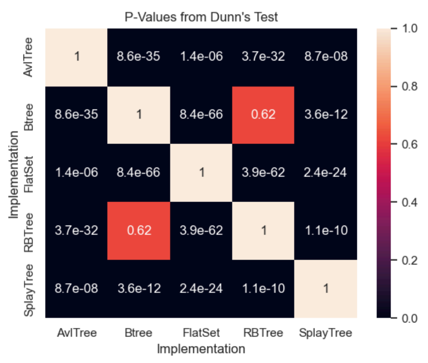
Case 7: Lookup of Non-Existent Data

Ranking	Implementation	Mean Time (s)	
1 & 2	B Tree	0.0000002338	
1 & 2	AVL Tree	0.0000002512	
3	Red Black Tree	0.0000003963	
4	Flat Set	0.0000004300	
5	Splay Tree	0.0010998713	

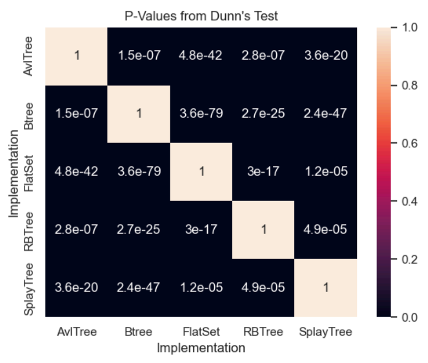
Case 8: Deletion of Sorted Data

Ranking	Implementation	Mean Time (s)	
1	AVL Tree	0.0001541501	
2	B Tree	0.0002510096	
3	Splay Tree	0.0002884075	
4	Red Black Tree	0.0005600041	
5	Flat Set	0.0028890900	

Case 9: Deletion of Reverse Sorted Data

Ranking	Implementation	Mean Time (s)	
1	Flat Set	0.0001672320	
2	AVL Tree	0.0002262413	
3	Splay Tree	0.0004078817	
4 & 5	B Tree	0.0004891483	
4 & 5	Red Black Tree	0.0005731628	

Case 10: Deletion of Unsorted Data

Ranking	Implementation	Mean Time (s)	
1	B Tree	0.0006243209	
2	AVL Tree	0.0008736325	
3	Red Black Tree	0.0011701071	
4	Splay Tree	0.0017852308	
5	Flat Set	0.0019858109	

Findings Summarized

	Insertion			Lookup				Deletion		
rank	Sorted Data	Reverse Sorted Data	Unsorted Data	Sorted Data	Reverse Sorted Data	Unsorted Data	Non-Existent Data	Sorted Data	Reverse Sorted Data	Unsorted Data
1	Splay Tree	Splay Tree	B Tree	Flat Set & B Tree	AVL Tree & Flat Set	B Tree & Flat Set	B Tree & AVL Tree	AVL Tree	Flat Set	B Tree
2	Flat Set	B Tree	AVL Tree					B Tree	AVL Tree	AVL Tree
3	AVL Tree	AVL Tree	Splay Tree & Red Black tree	AVL Tree	B Tree & Red Black Tree	Red Black Tree	Red Black Tree	Splay Tree	Splay Tree	Red Black Tree
4	B Tree	Red Black		Red Black		AVL Tree	Flat Set	Red Black	B Tree & Red	Splay Tree

		Tree		Tree				Tree	Black Tree	
5	Red Black Tree	Flat Set	Flat Set	Splay Tree	Splay Tree	Splay Tree	Splay Tree	Flat Set		Flat Set

Analysis of Findings

First, the nature of the data itself impacts the performance of different data structures. We saw that each use case had its own ranking of which data structure worked best, and no single data structure was universally optimal. For example, splay trees performed especially well on data that was already sorted or reverse sorted, because its design brings recently inserted elements to the root. Another example is the flat set, which can insert sorted data very quickly since no element shifting is needed, but slows down when inserting into an unsorted sequence. Conversely, for deletion, it is the other way around: deleting unsorted data in a FlatSet is fast, while deleting sorted data is slower because it requires shifting.

Second, we have observed that a cache-friendly data structure can have a significant impact on performance. The B-tree is a good illustration of this. Across all operations, insertions, lookups, and deletions, on unsorted data, B-trees generally outperform other data structures. We believe this is largely due to memory layout: B-trees store nodes in contiguous memory blocks, which makes them more cache-efficient. Other balanced trees store each node separately, which is less cache-friendly, even though the theoretical time complexity for all these operations remains O(log n).

Errors and Limitations

A stand-out inconsistency in the experiment that we recorded in our findings involves Case 4. When going to rank our implementations we observed that the Flat Set and the B Tree are unable to be compared due to their high p-value of 0.25 placing them both in the top spot. However, when we tried to compare them both to the AVL Tree which has the next largest mean time we found that when combined with the Flat Set we got a p-value of 0.22 which would not be rejected while the B Tree with AVL gave a p-value of 0.018 with would be rejected. It was particularly puzzling since the Flat Set's meantime was further away than the B Tree's from the AVL Tree. We suspect that since Dunn's test uses ranked sum means the resulting calculation may differ from our simple mean calculation throwing off our ranking.

We also observed that the Red Black Tree typically started up with high benchmark times and then settled down to a more stabilized time in almost all of the cases over the course of the 100 iterations. We did not anticipate that this would have a negative affect on our results. However, it is an interesting contrast to note considering the rest of the implementations had comparably steady benchmark times throughout the iterations.

Future Considerations

Due to our restriction on time and resources, we were not able to test every possible combination of ordered sets, their operations, and types of data. We had to refine our experiment and decide on what would be most important to test. First and foremost we wanted to create an environment that might replicate a real-world use case for an ordered set. This is why we used the insert(), find(), and delete() operations, as these are the most commonly used in this context and therefore the most relevant. We also chose to use the random uniformly distributed data while sorted, unsorted, and randomly ordered as these would test potential best and worst-case scenarios. In the future, if we were to have more time and resources, we would like to test a larger number of ordered set implementations, their operations, and types of data.

Conclusion

In this project, we conducted a comprehensive performance analysis of five different ordered set implementations under a variety of scenarios. Specifically, we examined the time it took to perform single operations, insertions, lookups, and deletions, on datasets with distinct characteristics, including sorted, reverse-sorted, and random inputs. Because our data could not be normalized, we employed the Kruskal-Wallis and Dunn’s tests to determine statistical significance in performance differences. From these tests, we not only established a clear ranking among the implementations for each case but also uncovered two key insights: the structure of the input data and how each implementation interacts with processor caches both play a crucial role in overall efficiency. In other words, when choosing which ordered set implementation to use, it is essential to consider both the nature of the data and the memory-access patterns involved in order to achieve the best possible performance.