Produce a short report (450-500 words) with graphs and/or tables describing the observed behavior when using the volatile keyword versus without.

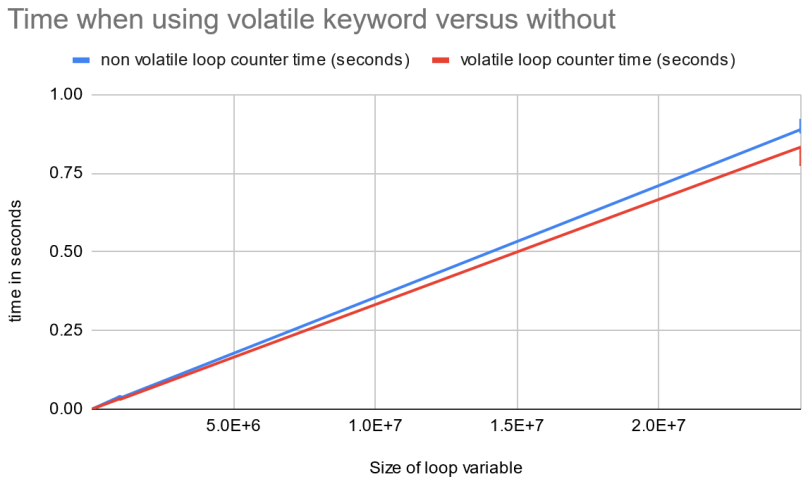

Figure 1

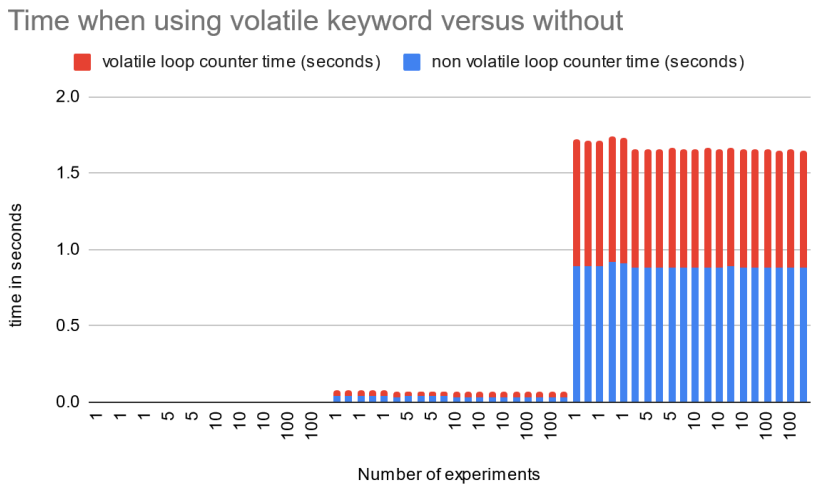

Figure 2

Within Java programming, the volatile keyword changes how a variable's data is stored in memory. When using the volatile keyword, the variable's data is stored directly in main memory everytime, preventing it from be chached in the local thread. This allows any changes made to the variable to be immediately seen and used by any other thread.

In my code, I was comparing the performance of a nonvolatile loop counter variable to a volatile loop counter variable. My program timed how long it took to update a separate variable using the loop counter across varying loop sizes and experiment numbers. The loop size is how many of these operations would be performed per experiment. The variables are used as loop counters, incrementing by one after a simple operation inside the loop. They are also both access in the loop to determine what operation is done to a different variable and accessed again to add or subtract from that variable.

When using the volatile keyword, the time it took update a sum using a volatile loop counter was always slightly less than the time it took using a nonvolatile loop counter. As you can see in *Figure 1*, the time it took for the nonvolatile and volatile operations were very similar, especially when the loop variable was smaller. When there were not many operations (the loop variable was small), both the volatile and nonvolatile keywords performed very similarly and both were extremely fast. However, as the number of operations increased (the size of loop variable), the volatile keyword started to perform slightly faster. As you can see in *Figure 2*, this was irrespective of the number of experiments performed. I would hypothesize that as more operations are needed, a volatile keyword would help the program run faster for these types of operations.

The volatile might have made my program run more efficiently due to the fact that it causes the information to always be stored in main memory. This ensured that the operations within the loop could immediately see any changes made to the variable, increasing operation speeds. However, the volatile keyword is not always the best. More complex programs with more dependent operations could run into synchronicity problems when using the volatile keyword, causing the program to slow down or produce the wrong output. This is because the data would always be stored in main memory and each operation on it cannot be interrupted by the next. This means for multiple threads that rely on the result of the other threads, they can end up waiting longer for the result, or began operations on the wrong value. In my code, the operation were simple, but the volatile keyword might've caused problems in a more complicated program or if two machines were running the program at the same time, but the volatile keyword only helped to speed up the program in this case.

Produce a short report (450-500 words) with graphs and/or tables describing the observed behavior when accessing elements at the prescribed portions of the array.

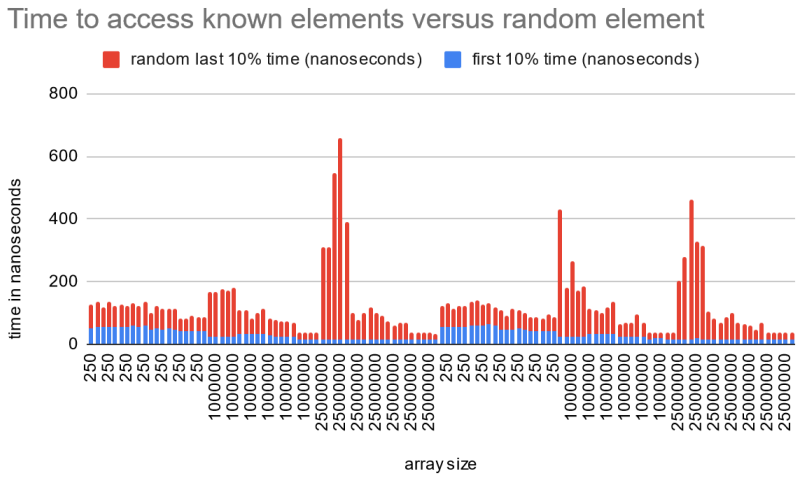

Figure 1

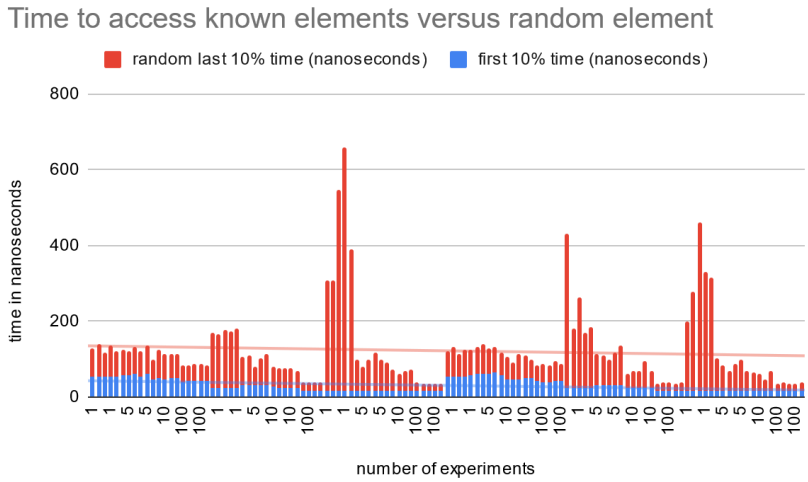

Figure 2

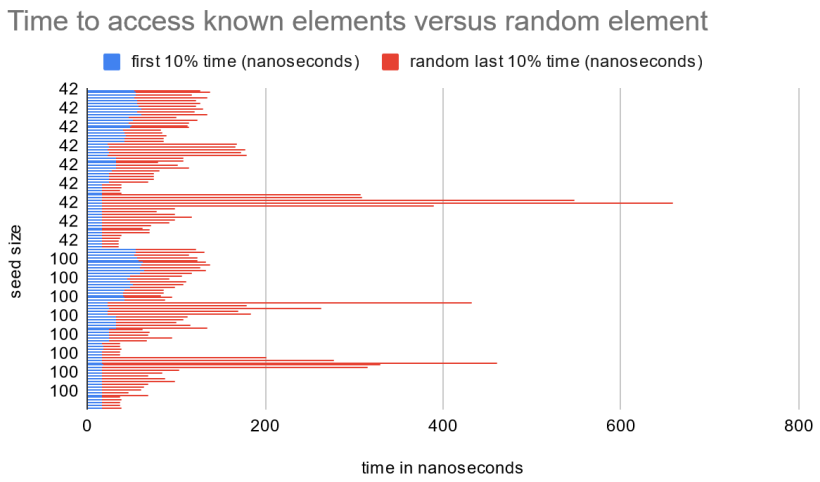

Figure 3
Consistenly, it was faster for the program to access known elements rather than a single random element.

My program first filled an array of a prescribed size with a range of random elements within the bounds of the gievn seed. Then, for a given experiment number of times, the program looped through the first 10% of the array and timed how long it took the program to access the elements, then got the average of those times. Then, it created a random index within the last 10% of the array and timed how long it took to access that random element. Regardless in variations of the seed size, number of experiments, or array size, the program was always faster at accessing the known elements in the first 10% of the array rather than the random element in the last 10%.

It took the program about the same time to access the known elements irrespective of the number of experiments, as seen in *Figure 2*. There is a curve showing it takes a certain amount of time which then decreases, not depending on the number of experiments. The data also shows a similar trend when looking at seed size. I tested two seed sizes: 42 and 100. As seen in *Figure 3*, they both took similar amount of time to access the first 10%. The trend in access time is a curved pattern showing the time starting high for the seed of 42, then decreasing. The curve is the same with the seed 100. These two figures suggest that there is something else that the time it took depends on. This is because, as seen in *Figure 1*, the time it took to access the elements mostly depended on the size of the array. The larger the array, the less time it took to access the known elements, which suggests a relationship between the size of the array and access efficiency.

This trend of a higher performance for accessing a known element over a random element shows how important locality of reference is in program optimization. Locality of reference is when a program access elements that are close to each other in the program's memory. Within my program, the accessing of known elements is more efficient than accessing a random element, most likely because of the elements' proximity to each other in the system's memory.

Accessing the first 10% of the elements is better usage of the cache and reduced memory access overhead compared to accessing a random element located elsewhere in memory. This also explains why access times for the first 10% of the array decreased as array size increased, showing that the repeated access of memory decreased the time needed to access the next element.

In conclusion, in order to develop the most efficient program, minimizing the number of random accesses and increasing the quality of data grouping so that most needed information is close together can greatly increase the efficiency of a program.

Produce a short report (450-500 words) with graphs and/or tables describing the observed behavior when using TreeSet versus a LinkedList

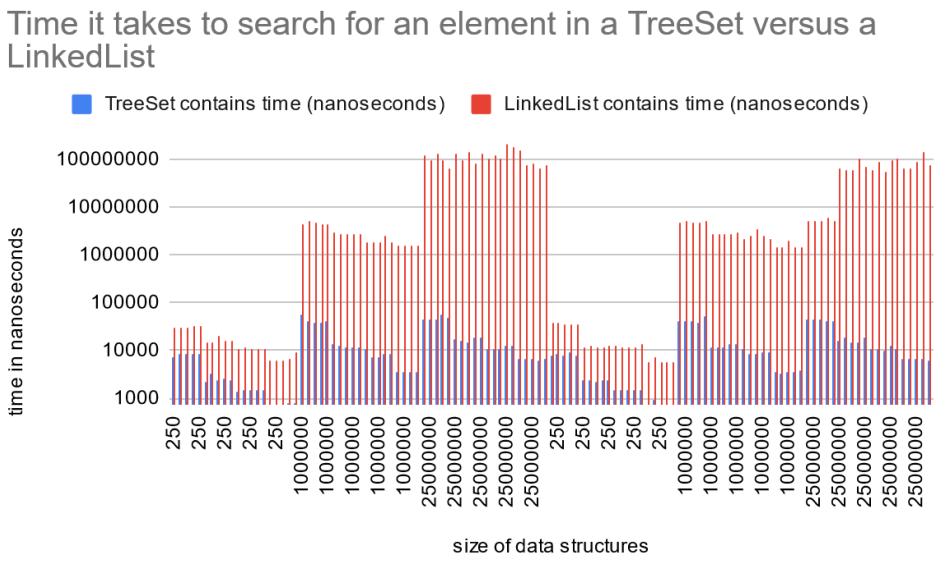

Figure 1

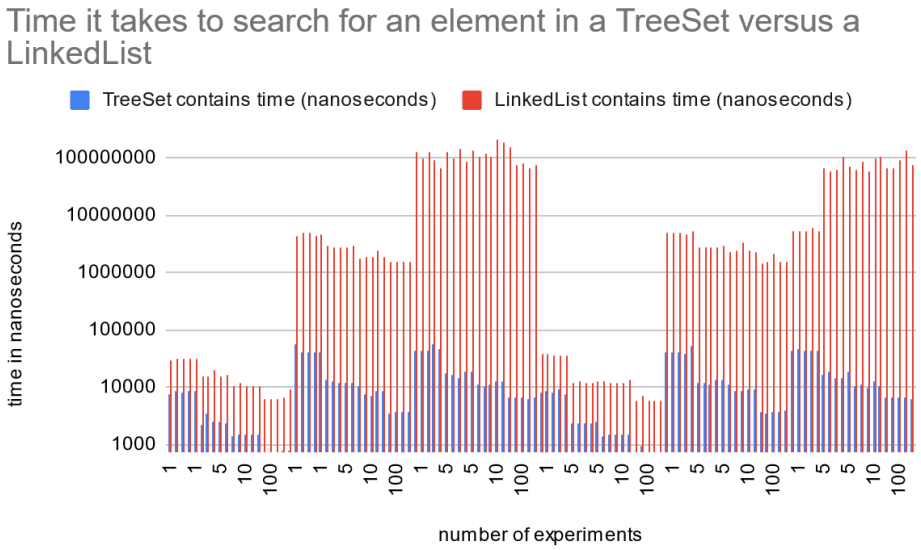

Figure 2

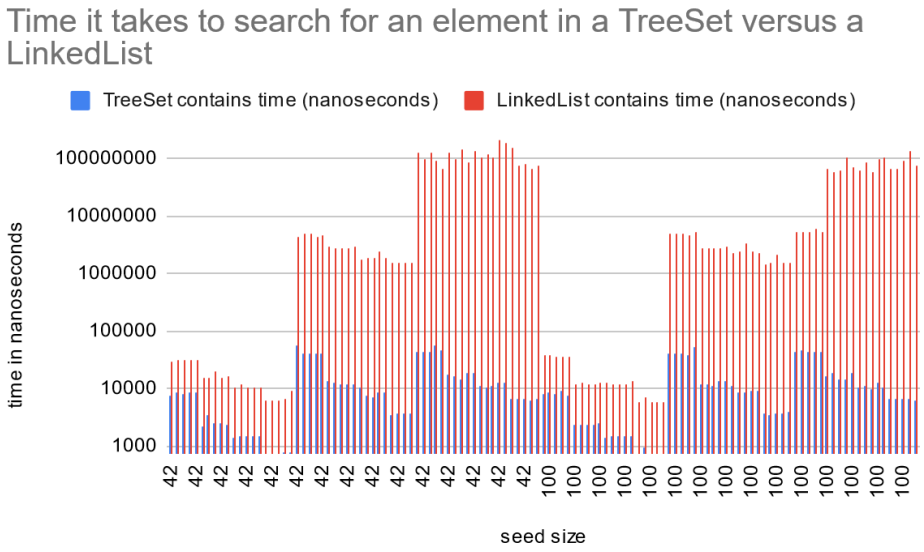

Figure 3

My program consistently demonstrates that it takes longer to find whether a LinkedList contains a random element than it takes for a TreeSet to find the same random element. Though a comparative analysis of the efficiency of the program and these two data structures, I have come to the conclusion that a TreeSet is more efficient at finding a specified element. Using the three above graphs, I will discuss the behavior of my program and how it demonstrates this.

My program first creates and fills both a TreeSet and LinkedList of a specified size with numbers 0 through size, not including size. Then a random number is generated with the bounds of size. Afterwards, my programt times how long it takes the .contains() method to find this random number in the TreeSet and then the LinkedList. This happens for experiments number of times. Experiments is another number, like size, that is specified when calling the program to run.

It takes far longer to find this random element in the LinkedList rather than in the TreeSet, regardless of experiment number, as seen in *Figure 2*. A similar trend is shown in *Figure 3* when looking at the impact of seed size on access times. I used two seed sizes: 42 and 100. As *Figure 3* shows, the time it takes to find the random element varies even using the same seed size, but it is always much less time for the TreeSet versus the LinkedList. This suggests that something else is affecting the time it takes for these data structures to find a random element. This something else, in other words, the primary factor affecting the performance is the size of the data structures. As *Figure 1* shows, the time it takes to find a random element in both data structures increases as the size of the data structures increases. As the data trends in the other graphs show, the time it takes for the TreeSet to find the random element is always less than the time it takes the LinkedList.

This data tells me TreeSets are more efficient data structures in terms of searching for a specified element than LinkedLists. This has many implications when building a program.

Depending on what behavior a program needs to exhibit, the software developer needs to chose the correct data structure. If the program needs to have efficient element retrieval and searching, a TreeSet would be a better choice. However, if the programs goals align more closely with efficient insertion and deletion, a LinkedList would be the more fitting choice.

In summary, this comparative analysis of the efficiency of these two data structures, TreeSet and LinkedList, lends an important insight into what data structures to choose depending on your program's target behavior. In order to create the most efficient program, a developer has to choose the correct data structure to optimize the performance across the specific computational tasks.