Andrew Aberer

Shrideep Pallickara

Computer Science 250

06 March 2024

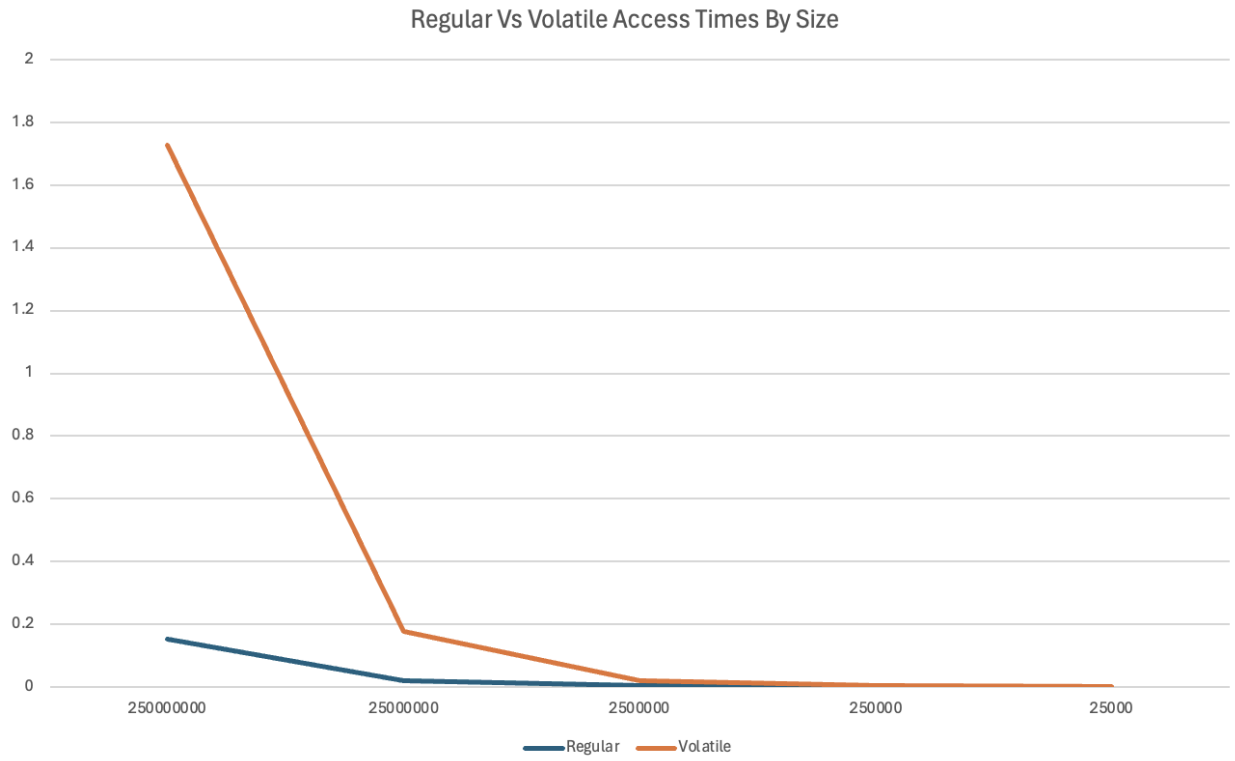
Caching, Memory accesses, and Choice of Data Structures

Task 1:

Starting with task number one, we compared the regular vs volatile memory average access times. Taking in only two input arguments regulating how many times a sum should be added or subtracted from depending on if the input is even or odd. This is then looped over the second input argument dictating how many iterations of this arithmetic loop should be completed. Computing the average time a normal cached read and access time takes as well as the average read and access time of volatile storage. The graph below shows the comparison of cached vs volatile times by memory size. As what can be observed in the chart, as memory size gets arbitrarily large, volatile memory times increase exponentially compared to the linear access time of cached information. The collected data makes sense because information stored closer to the Central Processing Unit (CPU), where the operations are calculated, is in result faster than information stored in main memory. As volatile information stored in main memory has to travel upstream further performing checks all the way until it reaches main memory. Increasing size by a factor of 10 slows down main memory access times drastically. However, if size is relatively small the access times are mostly indistinguishable with volatile information still being the slower of the two. The impact of the slower access time of the volatile is more clearly noticed when size is large, meaning more information is being stored farther away in main memory. If the same size of large information is cached closer then the computer has to not only check less

places for the information, but because the information is physically stored closer it adds to the speed of access times. During runtime since the loop variable is stored in main memory or volatile when performing the base statement check to see if the loop counter has reached the maximum size or not, the access takes place in main memory adding to the delay. Compared to the regular loop variable which is cached close to the CPU, another reason why regular access times are much faster when comparing access times. This relation is directly correlated to the graph where I recorded access times compared side by side with varying size allocations. First looking at the blue line, which is regular cached information, we see that even when the size is quite large like 250,000,000 the times recorded are still within 1 tenth of a second dropping by a factor of 10. Then we can see past the 25,000,000 mark, times only differ around 1 hundredth or even 1 thousandth of a second. Next, looking at the orange line, which is volatile information stored in main memory, with the same scale of 250,000,000 the access starts around 1 and a half seconds. Yet again, scaling down by a factor of ten the time drops to 1 tenth of a second. This is a drastic scaling decrease and if we iterate the scale down by another factor of 10, access time drops to a hundredth of a second. We can look at the graph and observe the large difference but we can also compare times in seconds with the largest scale tested resulting in an overall time difference of 1.42069 seconds respectively. This is only a small scale example but if numbers were to keep increasing at an exponential rate, the overall time difference would increase at an

exponential rate as well.

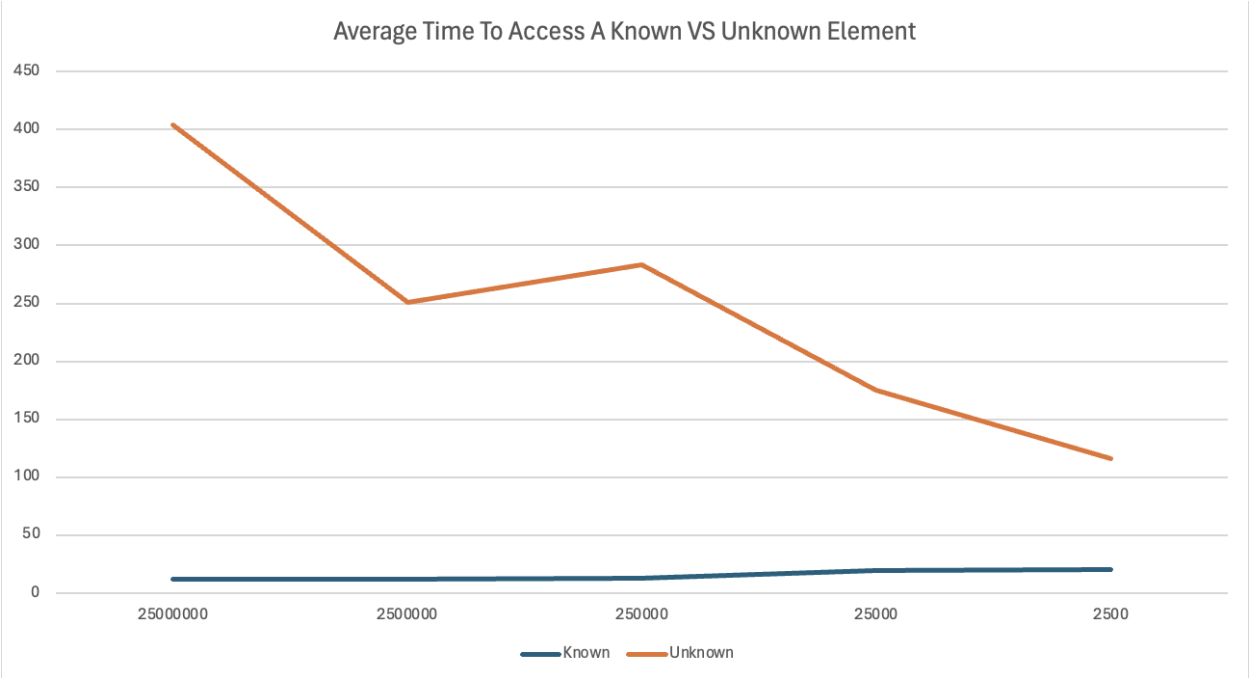


Task 2:

Now looking at task 2, we continue to have three input args first being number of size, number of experiments, and set seed for randomization. The third argument could vary to change the randomization based on the input, or no input for “complete randomness”. Creating an Integer array of the input size and filling the array with random values. Next we create a double nested for loop with the first iterating for the number of experiments we want to perform and the inner loop iterating of the array size. We then time the access times of getting a known element specified by the loop counter and add the access time to the total sum. Outside of this loop we try to access a random value, yet again specified by the input seed, for n number of experiments. These values are both added to a total sum for both the regular and random access times. Then, taking the total time to access a single element in the first 10 percent of the array and dividing it

by the number of experiments performed to compute the average. After this, the access time to check a single random element in the last 10 percent of the array is computed. The keyword here is random, as seen in the graph below, the access time is much slower and inconsistent when comparing the two access times side by side. Since the array type being accessed is not a primitive type but the object is an instance of Integer the difference in the two accesses is more pronounced. Taking a closer look at the graph and starting with the blue line, represented as a known element of an array access time we can see that the time remains at a consistent linear scale with very little variation no matter the array size. Next, looking at the orange line or the retrieval time of a random element of an array the same pattern follows as task one. In task one as size got very large volatile access times slowed down exponentially. Comparing these results with the orange line in task two, we yet again see this line increasing at a similar rate. When the array size is 250,000,000 I recorded the unknown access time being 391.50 nanoseconds compared to the known array element time of 12.15 nanoseconds, resulting in a time difference of 379.35 nanoseconds. This difference is so large in scale compared to the known element access time. This is with the largest array size I was able to test, looking at the smallest possible array size I was able to test the differences to get closer to each other with the known element still being the faster of the two. With a small size of only 2500 the unknown retrieval time was 95.9 nanoseconds and the known time was 20.05 nanoseconds, resulting in a 75.85 nanosecond average access time difference. As the size started to get smaller by the same factor of 10 the known element access time started to increase in my recorded data. The reason why I think this is because since the size is getting smaller, the number of experiments ran remains the same. So when the average run time is being computed it is still dividing by the same factor, in my case by 20. Therefore when computing the average, since the unknown access time is larger when it is

divided by the factor of 20 the time is reduced more noticeably.



Task 3:

Finally for task 3 we were assigned to compare the speed differences between two different data structures. The first being a TreeSet, a tree set is a self balancing tree which as items are added or removed, they are balanced when these operations are performed. The second data structure we are comparing is a Linked List, a linked list is assigned a type and runtime but the size is not specified similar to a tree set. However, since a size was specified through an argument at runtime, these data structures are filled with values starting from 0 ranging to the size. Since all elements are added before the time comparison is calculated, the addition and subtraction of elements should not be a factor in the average time. So with this out of the way, the only time differences in element accesses should be directly related to the data structures. Taking a look at the Big O complexity of common data structures and algorithms we see that a tree set average access time has a Big O complexity of O(log(n)). What this means is for n number of elements, on average it should only take log of n time to access any element in this

structure. Compared to a linked list which on an average access time takes O(n) which means this access time is for n elements it will take n amount of time. This data structure is much slower when comparing the two as this is relatively near a constant access time which is the best possible retrieval time a structure can have. So with this being said let's take a look at the data recorded in the graph. Starting again with the blue line of the tree set, we can see the Big O complexity with a visual representation. The blue line remains at a linear, or flat, rate showing similar access times with no regard to how many elements are added to the set. Compared to the orange line which is a linked list, this rate follows a similar trend as the two previous parts. As size starts to get very large, the access time increases at that same exponential scale. Let's look specifically at the largest size values compared to the smallest. When the size of the TreeSet and the LinkedList is 250,000,000 the average access time to find a random element is 6593.90 nanoseconds for the tree set and 118037691.45 nanoseconds. Yet again, the dramatic variation between average times is 118,031,097.55 nanoseconds or 118.03769145 milliseconds for a better perspective. Next with the smallest recorded size being 1950.10 nanoseconds from the TreeSet and 37805.60 nanoseconds from a LinkedList, the average time differs by 35,855.50 nanoseconds. Both data structures are using the .contains() method to try and access the random element so the only variation possible has to be coming from the actual implementations of the data structures. The data I recorded and the graph visualization matches with the expected access

times of the different data structures according to the average Big O complexity access time.