Profiling the impact of Caching, Memory accesses, and Choice of Data Structures for HW2

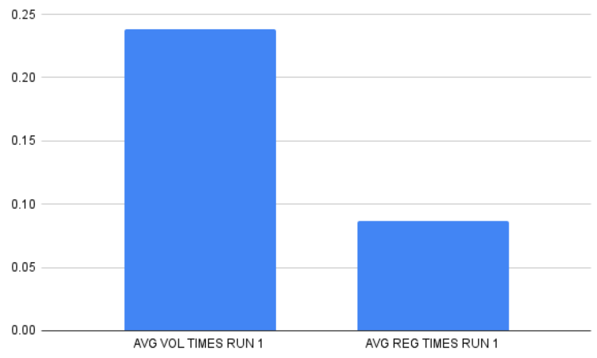
By: Chris Urbano

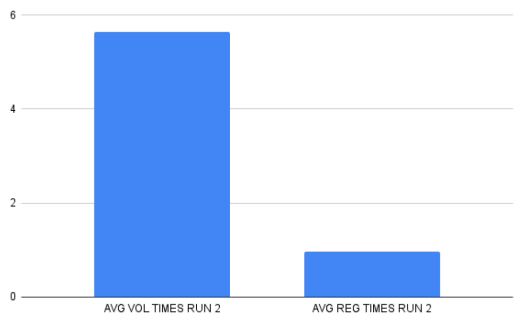
TASK 1 -
The main task of this function is to compare the performance/speed of programs both with and without caching. We can do this in java by using the *volatile* keyword. This keyword is used to instruct the compiler that the variable should not be cached and should go to main memory instead. I did this by using loops and the args given to me to help compare the execution time when the volatile keyword is used in the loop variable and when it is not used. My *taskOne* method header goes as following; `public static void taskOne(String[] args)`. My taskOne is a public static function with no return type and takes an array of strings as input, which contains the *size(Integer)*, the number of *experiments(Integer)*, and the *seed(Integer)* values.. My method starts by parsing the list for three arguments and assigning them to three newly declared *int* variables named size, experiments and seed. The second half of my *taskOne* starts by printing "Task 1" on a new line. This is followed by declaring a new variable named avgVolTotal and assigning the output of the method *vol(int size, int experiments, int seed)* to the new variable. This is followed by declaring a new variable named a*vgTotal* and assigning the output of the method *noVol(int size, int experiments, int seed)* to the new variable. Both *noVol* and *vol* are helper functions I created to profile the impact of caching when not using and using the *volatile* keyword. I will go more into the helper function after I finish describing the rest of the method *taskOne.* After the calls to the vol and noVol functions I have two printf lines that print out the running total of addition and subtraction operations for both *vol* and *noVol.* The lines are formatted so the average totals are converted to floats and are rounded to the second decimal place. Now that I discussed the code in my *taskOne* function I can discuss the workings of both *vol* and *noVol.*

noVol(helper function for task one)
The function *noVol* takes three arguments: name *size* (integer), *experiments* (integer), and *seed* (integer). The function's return type is a long value. At the beginning of the function, a local variable named *runningTotal* is initialized to zero and then the function records the start time using System.nanoTime(). The function then runs a nested for-loop. The outer loop iterates *experiment* times, and the inner loop iterates *size* times. Each iteration of the inner loop checks if *k* is even or odd using the modulo operator. If *k* is even, *runningTotal* is decremented by *k*, otherwise, it is incremented by *k*. After both loops have finished the function computes the average of *runningTotal* by dividing it by *experiments* and assigning the result to *avgTotal*. The function then records the end time using System.nanoTime() and calculates the difference between the start and end times. The time is then converted to seconds and printed to the console with five decimal places using a printf statement. At the very end, the function returns the *avgTotal* variable.

vol(helper function for task one)
The function *Vol* takes three arguments named *size* (integer), *experiments* (integer), and *seed* (integer). The function's return type is a long value. At the beginning of the function, a local variable named

runningTotal is initialized to zero and then the function records the start time using System.nanoTime(). The function then runs a nested for-loop. The outer loop iterates *experiment* times, and the inner loop iterates *size* times. The loop variable in this function is a volatile variable. Each iteration of the inner loop checks if *vol* is even or odd using the modulo operator. If *vol* is even, *runningTotal* is decremented by *vol*, otherwise it is incremented by *vol*. After both loops have finished the function computes the average of *runningTotal* by dividing it by *experiments* and assigning the result to *avgTotal*. The function then records the end time using System.nanoTime() and calculates the difference between the start and end times. The time is then converted to seconds and printed to the console with five decimal places using a printf statement. At the very end the function returns the *avgTotal* variable.





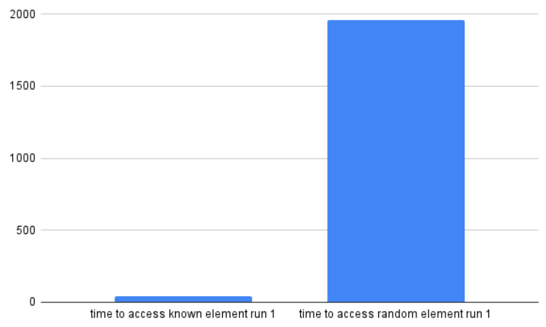
These graphs show the time in nanoseconds it takes to access a variable using the volatile keyword and when not using it.

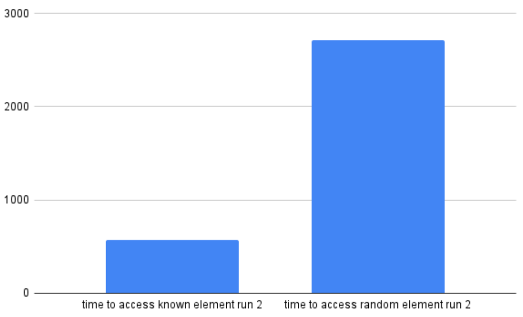
The general outcome of this test is that when using the *Volatile* keyword the average time it takes to access a variable is significantly slower than when without. Without the keyword, the time ranges between 40% faster with fewer experiments to 15% faster with more experiments. What this could possibly indicate is that the more you try to access an element the faster it comes when you try to access it whether you use caching or not.

Task 2-

The *taskTwo* method signature goes `public static void taskThree(String[] args)`.
The method takes in an array of strings as an argument. The array contains the *size*(Integer) which is used for the size of an array, the number of *experiments(*Integer), and the *seed(Integer)* for the random number generator. It creates an array of the specified *size* and fills it with random integers using the provided *seed*. The method then performs three tasks on the array, calculating the average time to access the first 10% of elements, the time to access a random element within the last 10% of the array, and the average of all the elements in the array. I do this by using multiple loops. In order to calculate the average time to access the first 10% of elements, the method initializes a variable *firstTenRunningTimeTotal* in order to keep track of the total time taken to access these elements in the first 10%, and a variable *firstTenSum* to keep track of the sum of the first 10%. The method then loops through the array *experiments a number* of times and each iteration loops through the first 10% of the array. The method then accesses each element and measures the time taken to do it using *System.nanoTime()*. The time taken is then assigned to *firstTenRunningTimeTotal* and the value of the element is added to *firstTenSum*. After all of the

experiments have been completed, the average time taken to access each element in the first 10% is calculated by dividing *firstTenRunningTimeTotal* by the total number of elements in the first 10% of the array. In order to calculate the time taken to access a random element within the last 10% of the array, the method generates a random index within the last 10% of the array using the *Random* object generated with the provided *seed*. It then accesses the element at this index and measures the time taken to do so using System.nanoTime(). The time taken is stored in the *lastTenTime* variable. I then calculated the average of all the elements in the array by initializing a variable *sum* to keep track of the sum of all the variables in the array. I then loop through the array and add each variable to the *sum.* I then find the average by dividing the *sum* by the length of the array. The method then outputs the results using 3 *printf* functions.



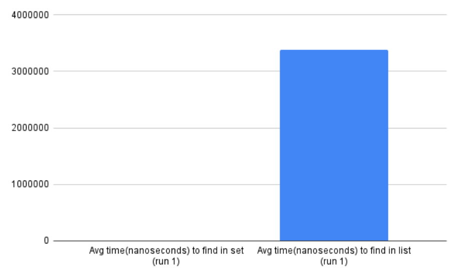


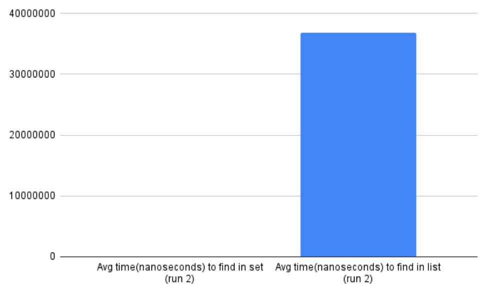
This graph shows the avg time in nanoseconds to access a single variable in the first 10% of the array vs a single random element is the last 10% of the array.

These graphs indicate that the time it takes to access alements closer to the beginning of the list are signifigantly faster than an element at the end of the list. This could be useful when deciding how to search through a particular array or data structure.

Task 3-

The *taskThree* method provides a useful comparison between two different data structures. The results of this method show the time it takes to access an element in an array. It can help determine the most appropriate data structure for whatever specific use case you could need it for. The taskThree method takes three arguments as input parameters: size*(Integer)*, *experiments(Integer)*, and *seed(Integer)*. These arguments define the size of the data structures, the number of experiments, and the seed for the random number generator. The method starts off by creating and initializing a LinkedList and a TreeSet with integers ranging from 0 to size. The LinkedList is a linear data structure while the TreeSet is a sorted data structure. The method then conducts a search operation on each data structure for *experiments* number of times using a random number generated by the *random variable*. The search operation for the LinkedList is done by calling the *contains()* method with the randomly generated number as its argument. Very similarly, the search operation for the TreeSet is performed by calling the *contains()* method with the same randomly generated number as its argument.In order to measure the time of the search operations on each data structure, the System.nanoTime() method is used to capture the time before and after each operation. The difference between the end time and start time is then used to calculate the total time taken. The time is taken for *experiments* number of times then I take the accumulated time and divide it by the size of the array .The average time taken to search for an element in the LinkedList and the TreeSet are stored in the variables avgLLTime and avgTSTime. The method then prints out the average search times for each data structure using printf lines in order to make sure it is formatted correctly.





This graph show the time it takes to search for an element in the two data types: TreeSets and Linked List

This graphs outcome indicated that a *LinkedList* is a useful data structure for maintaining the order of elements, but not efficient for searching. The TreeSet provides an efficient search operation but is not as good when it comes to maintaining the order of elements. This data can be very useful when trying to choose what data structure to use for a certain use case.