Olivia Tang and Laura Freeman

CSE 373 Project 1 - Part 2d

20 April 2018

Group Write-Up

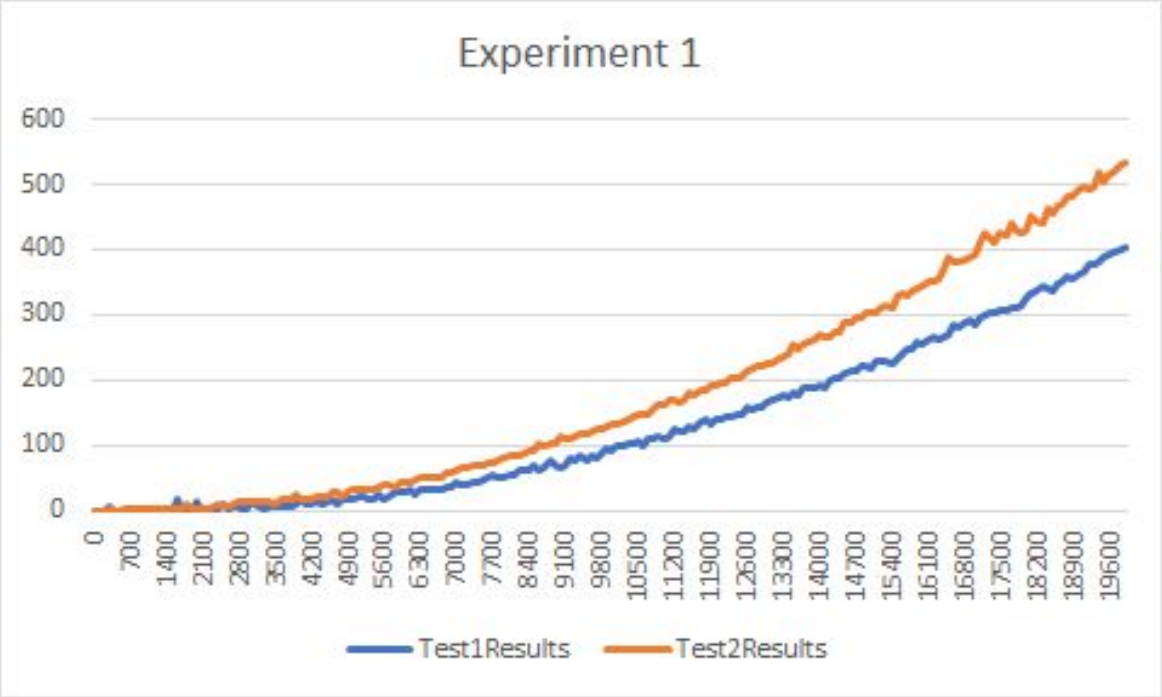
1.

In writing and testing our code, we realized that we needed to compare keys using the equals() method, rather than comparing with "==". This was because that comparison would only work if both objects pointed to the same reference. However, when we simply used "object1.equals(object2)", we received a null pointer exception when object 1 happened to be null. To mitigate this, we added a check to see whether object 1 was null or not, and we only compared keys when object 1 was not null. This way, we could compare keys without encountering errors. However, there was also the case where both objects had null values or they both pointed to the same reference. As a result, we also added another check for this and return true if this were the case. We encountered this issue both in testing our DoubleLinkedList class and our ArrayDictionary class, and we treated both cases in the same way.

2.

Experiment 1:

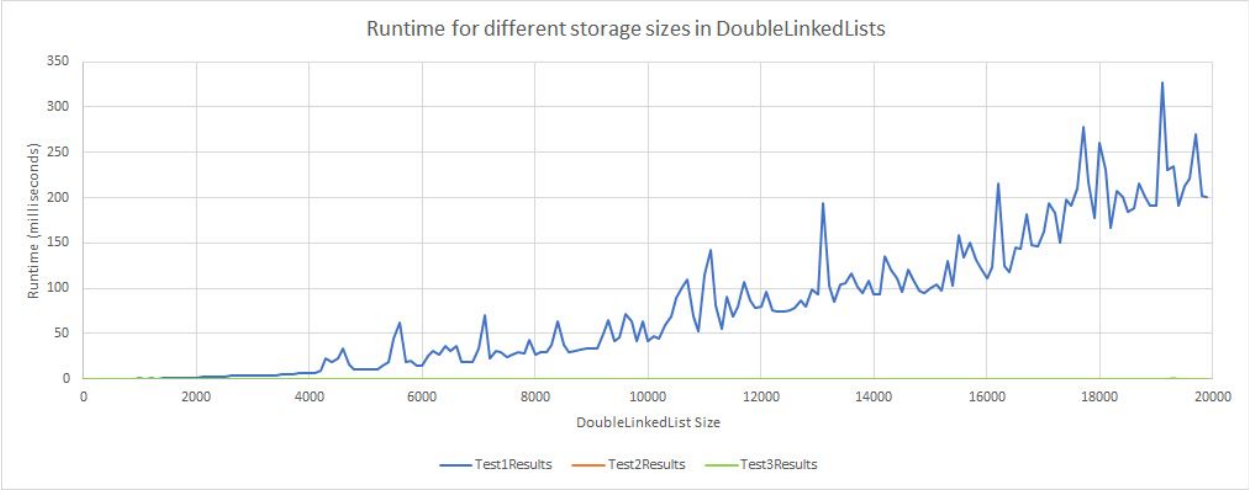
1. This experiment tests how long it takes, in milliseconds, for all values in an ArrayDictionary to be removed from the front (for test1) and from the back (for test2)
2. The larger ArrayDictionary is, the more time it will take for the tests to run. However, compared between removing from the front and removing the back, the runtime will be nearly the same. By removing from the front of the array, all pairs must be shifted down. By removing from the back of the array, the program must still traverse to the end of the array to remove the pair.
3. Plot



4. Test1Results depicts the time for removing from the front, and Test2Results depicts the time for removing from the end. We expected the runtimes between both tests to be the same, but it turned out that, as the size of the ArrayDictionary increased, the difference between runtimes for test1 versus test2 increased. As such, it seems that removing from the end takes longer than removing from the front. We think this is because removing pairs from the front only requires the removed pair to be compared with the next pair before all elements are shifted to the front. On the other hand, removing from the back requires all keys to be compared.

Experiment 2:

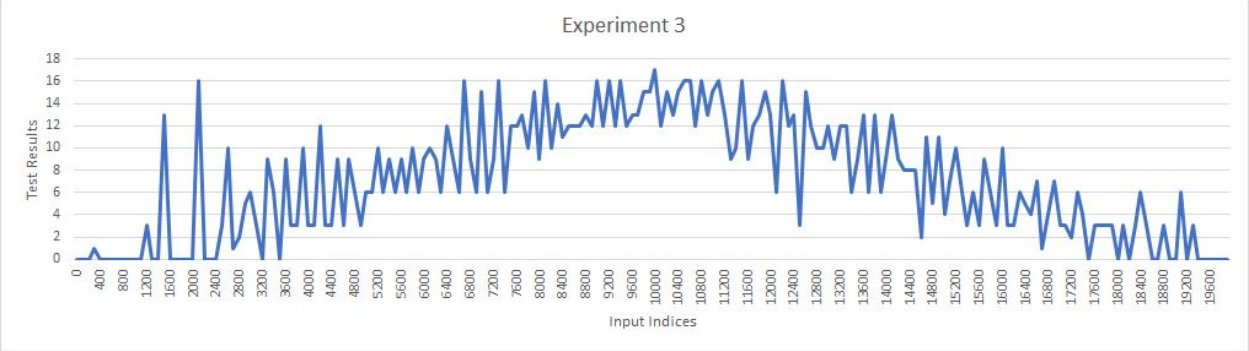
1. We believe this experiment tests the runtime of looping through a DoubleLinkedList at different sizes of storages with different methods: get and for loop, iteration, and for-each loop.
2. We predict that the for each loop and the iterator will run the experiment the fastest, because iterating and for each loops don't keep track of indices, and therefore use less storage. If we closely look at iterating through a for loop, we see that iterating requires us to keep track of a pointer indice to loop.
3. Plot



4. For Test 1 the runtime started to increase substantially when the list was around 3500 in size. We can see from the graph that our hypothesis was correct; The results from Test 1 yielded a steadily increasing runtime as the size of the list increased while the results from test2 and test 3 all yielded zero for runtime with an exception for when the data is size 19300, the runtime is 1 millisecond for test 3. We think this was because test1 is a nested loop, with the inner loop going through every node to find the element at a given index, and the outer loop going through every index in the list. The iterator and the for each loop did not need to keep track of indices, which is why they were so much faster.

Experiment 3:

1. This experiment tests the runtime of accessing different nodes in a DoubleLinkedList at different indices of a list size 20000.
2. We hypothesize that the runtime for accessing values will be the highest when the index of the value is in the middle of the list, and lower when the values approach the front or end of the list, due to the nature of the DoubleLinkedList of having a pointer node for the front and back of the list.
3. Plot

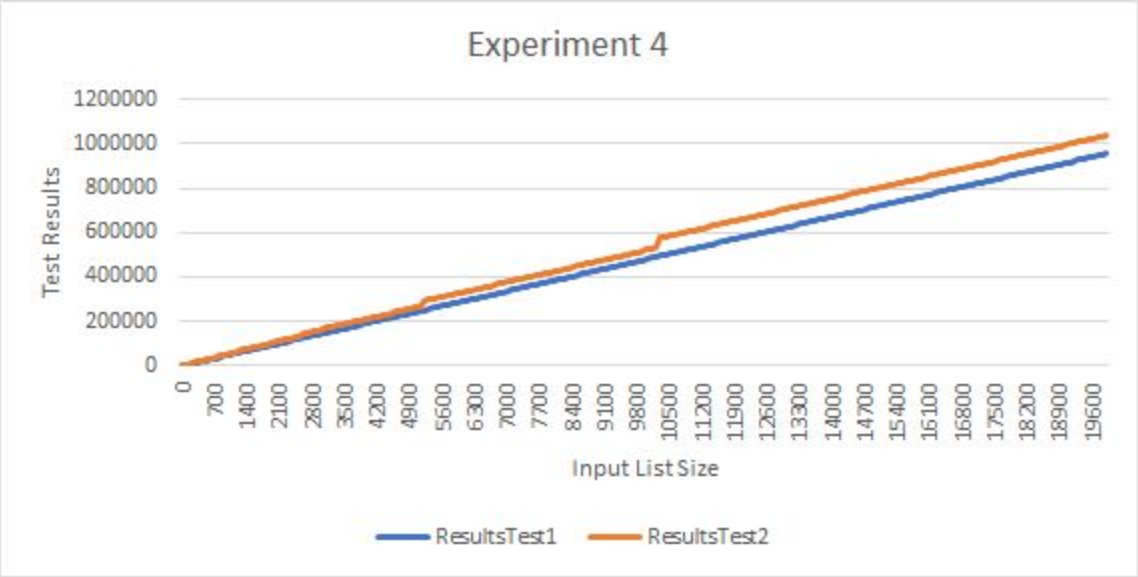


4. Results

   Our hypothesis was correct, according to the graph we can see that the runtime is substantially longer when the indices are in the middle of the list, and shorter when the indices are values close to the front or end of the list. However, what is interesting to note is that the runtime for accessing indices increases dramatically near the values 1600 and 2200, whereas the indices with a corresponding distance from the end (1600, 2200) in the front do not have a dramatic increase. We believe that this increase and subsequent increase occurred because methods accessing the first half (index < size / 2) of the list indices will start from the front, while methods accessing the second half (index > size / 2) of the list will start from the back. As a result, the close to the middle the index, the greater the time required.

Experiment 4:

1. This experiment tests for the memory used for ILists and IDictionaries of different storage sizes (0-20000).
2. We hypothesize that the memory used for the list will increase with the increasing size of the list, because a larger list would require more memory space to store more values. However we think that an IDictionary would use consistently less memory that an IList, due to the fact that an IList node will always store a reference to another node whereas an ArrayDictionary (which is an IDictionary) stores a Key-Value pair in an array.
3. Plot



4. We can conclude from the results that our hypothesis was incorrect. As we can see from the graph the overall memory used for an IDictionary and a IList were relatively the same until around input List Size 4900. From sizes 4900-19600, test2 (which tested for IDictionary) used up more memory than test1, which tested for IList. This is because of the jumps in Test Results, where the dictionary is doubled in capacity. However, Test1 and Test2 were both linear and generally parallel to each other. We think this is because every additional element in both the list and the dictionary results in a constant increase in memory required.