Michael Ruby

CSE 373

Project 1 Group write-up

In the ArrayDictionary class, in order to handle the search for null keys I created a private method that returned the index of a key in the dictionary. By doing this I was able to create an if/else statement if they were searching for a null key and search for a pair that did not have a key associated with the mapped value. By doing this I was able to create a method that could be implemented multiple times to return index locations. In the DoubleLinkedList class I used the equals method which was able to handle both null and non-null values.

Experiment 1

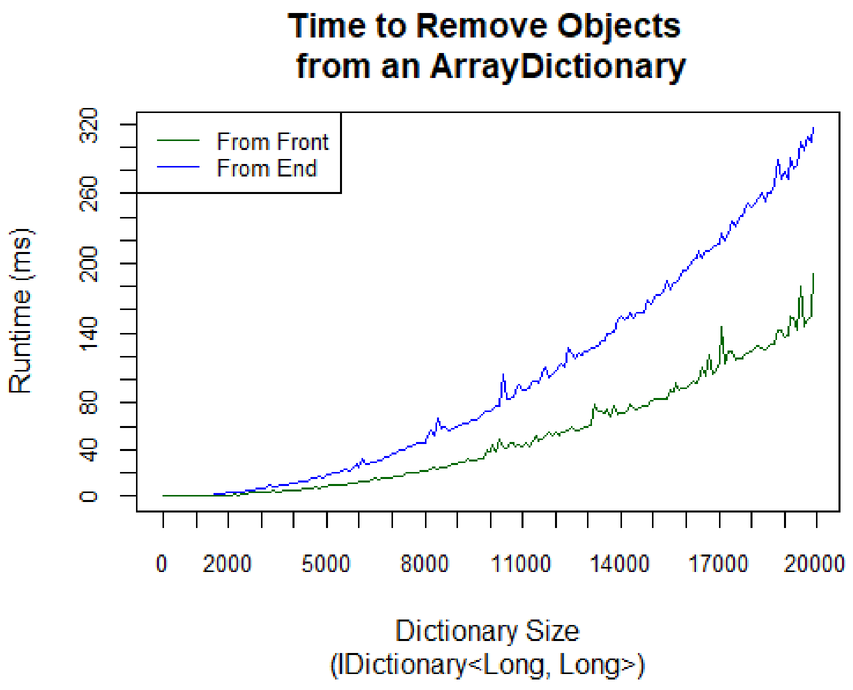
Here we are measuring the amount of time in milliseconds required to remove all values from an ArrayDictionary of longs. The sizes of the ArrayDictionary ranges from 0 to 20,000 in steps of 100, after running 5 trials the times are averaged for each size. In the Test 1 elements are removed in the order they were entered; while in Test 2 elements are removed from beginning with the last element added.

Predictions

The results of Test 1 are expected to be more efficient. Since the objects are being removed in the order they were added to the dictionary will need to be traversed in increasing amounts to find the element, however the element from the back is then assigned to that index so the half of the dictionary is then traversed twice. This means that for a dictionary of size two both elements are deleted from the first index, while a dictionary of size four will remove the first and second element and then remove the second and first element and a dictionary of size ten will delete elements one through five and then five through one.

The results of the Test 2 are expected to be far less efficient. Although there is no need to reassign an element, for the first n/2 elements the traversal of the dictionary is higher than that of any traversal that is made in Test 1. The increase in running time is due to the remove method implemented in the ArrayDictionary class; because for each object the entire array will need to be traversed to retrieve it from the end.

Results



Removing objects from the front as hypothesized was far more efficient than removing from the back, this is likely due to Test 1 only needing to traverse at most half of the list and then traversing less and less of the remaining remove() calls after the first half have been removed.

Experiment 2

In this experiment we are measuring the amount of time in milliseconds required to traverse a DoubleLinkedList with between 0 and 20,000 nodes, retrieving the value at each node, after conducting 5 trials the size of the list is increased by 100 and returns the average of the trials. In Test 1 elements are retrieved using the get() method, while in Test 2 elements are retrieved using the iterator and finally Test 3 retrieved elements using a foreach loop.

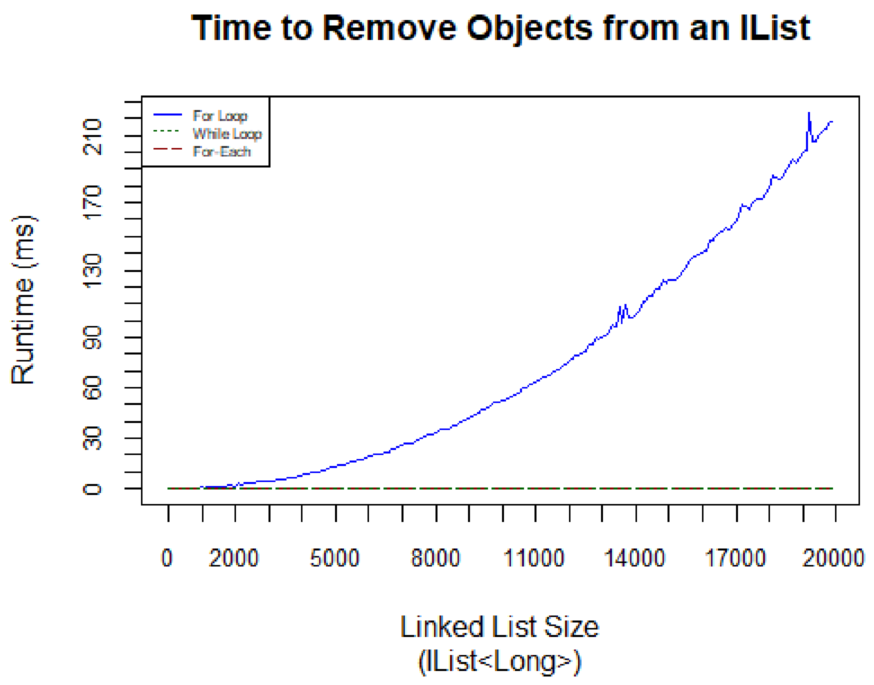
Predictions

Test 1 is expected to be the least efficient method, likely exponential time. Due to the use of a for loop coupled with a get() call, each element has to be searched for in the list in order to return the value and then start back at the beginning of the list for the next call.

Test 2 will be significantly more efficient and run in linear time. As we increase the size of the list by one element the while loop is only forced to execute an additional time. The use of the while loop however does introduce an additional step of checking the condition at each step.

Test 3 is expected to be the quickest method and run at nearly constant time as all that is going to be required is to retrieve the value of the element that the iterator is currently at.

Results



As expected the for-each loop ran in constant time and the for each loop appears to have run in approximately exponential time. However unexpectedly the while-loop also appears to have run in constant time; this is likely due to the while check not having as significant of an impact of runtime as expected.

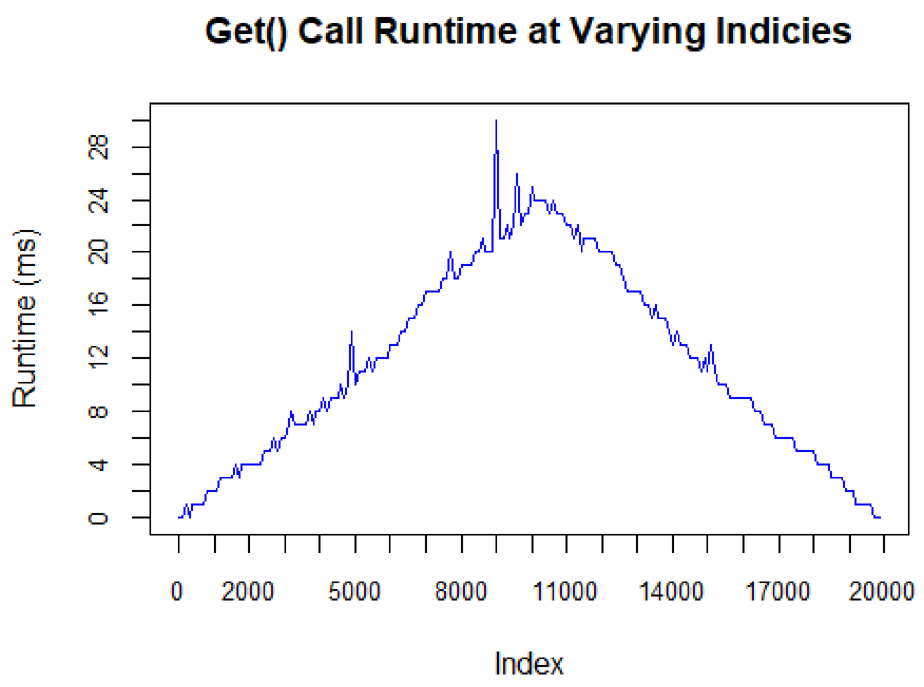
Experiment 3

In this experiment we calculate the the amount of time in milliseconds it takes to perform a 1000 gets at each index of a DoubleLinkedList with between 0 and 20,000 nodes, increasing the list size by 100 and conducting 5 trials for each step. At each step the average of the trials is returned.

Predictions

The runtime is expected to increase linearly for the first half of the List and then decrease linearly for the second half the the List. This change in runtime as the size of the List increases is a result of the implementation of the get() method. The get() method searhes from the back if the index is greater than half the size of the list and searches from the front otherwise.

Results



As expected we see the see the runtime increasing linearly through the first half of the indicies at which point it begins to decrease linearly over the second half until reaching the end of the list. Unsurprisingly the runtime is very nearly symmetric.

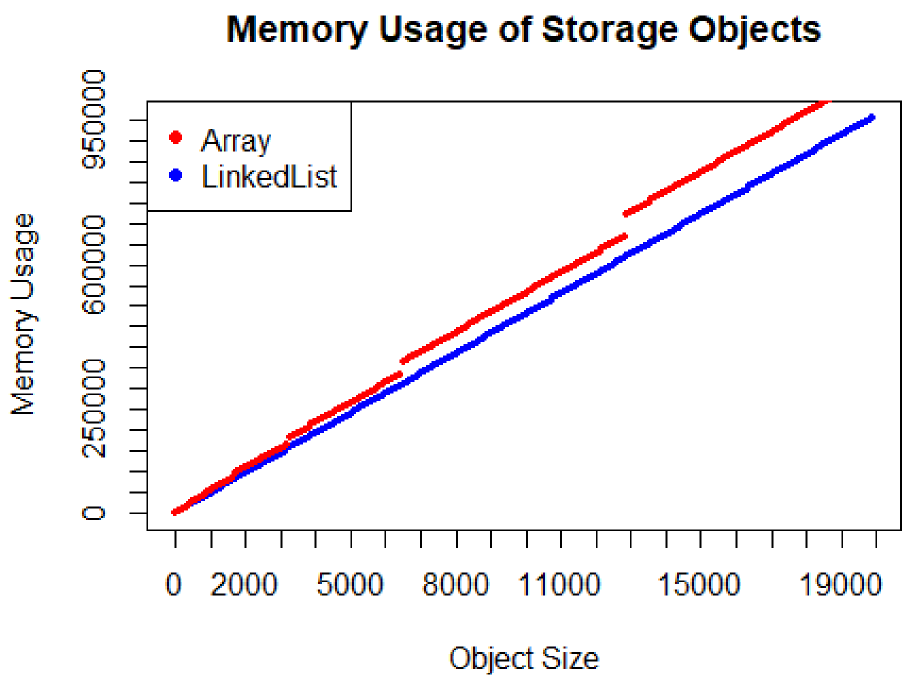
Experiment 4

The purpose of the final experiment is to determine the memory usage in bytes used by our two data structures of sizes between 0 and 20,000 with steps of 100. Test 1 examines the usage for a DoubleLinkedList, while Test 2 focuses on the ArrayDictionary.

Predictions

We would expect to see a lower memory usage in the DoubleLinkedList for larger data sizes than we would for the ArrayDictionary. For the ArrayDictionary there is expected to be nearly constant memory usage, where we will see increases only when the capacity is required to increase. Since everytime the capacity is increased it doubles, the memory usage is expected to begin as a low value and at every capacity increase we anticipate a jump in memory usage. Alternatively, when we look at DoubleLinkedList, the memory usage we expect to see is a linear rate of growth.

Results



As expected we see a linear growth rate of memory usage for LinkedList. Surprisingly however, we see nearly an identical growth rate for array. The prediction made about jumps in memory usage was correct however as we did see the anticipated jump in memory usage when the arrays capacity was increased. The unexpected linearity in the memory growth is likely atributable to the increased memory usage due to the change of a null value to a real value.