Part 2d: Writeup

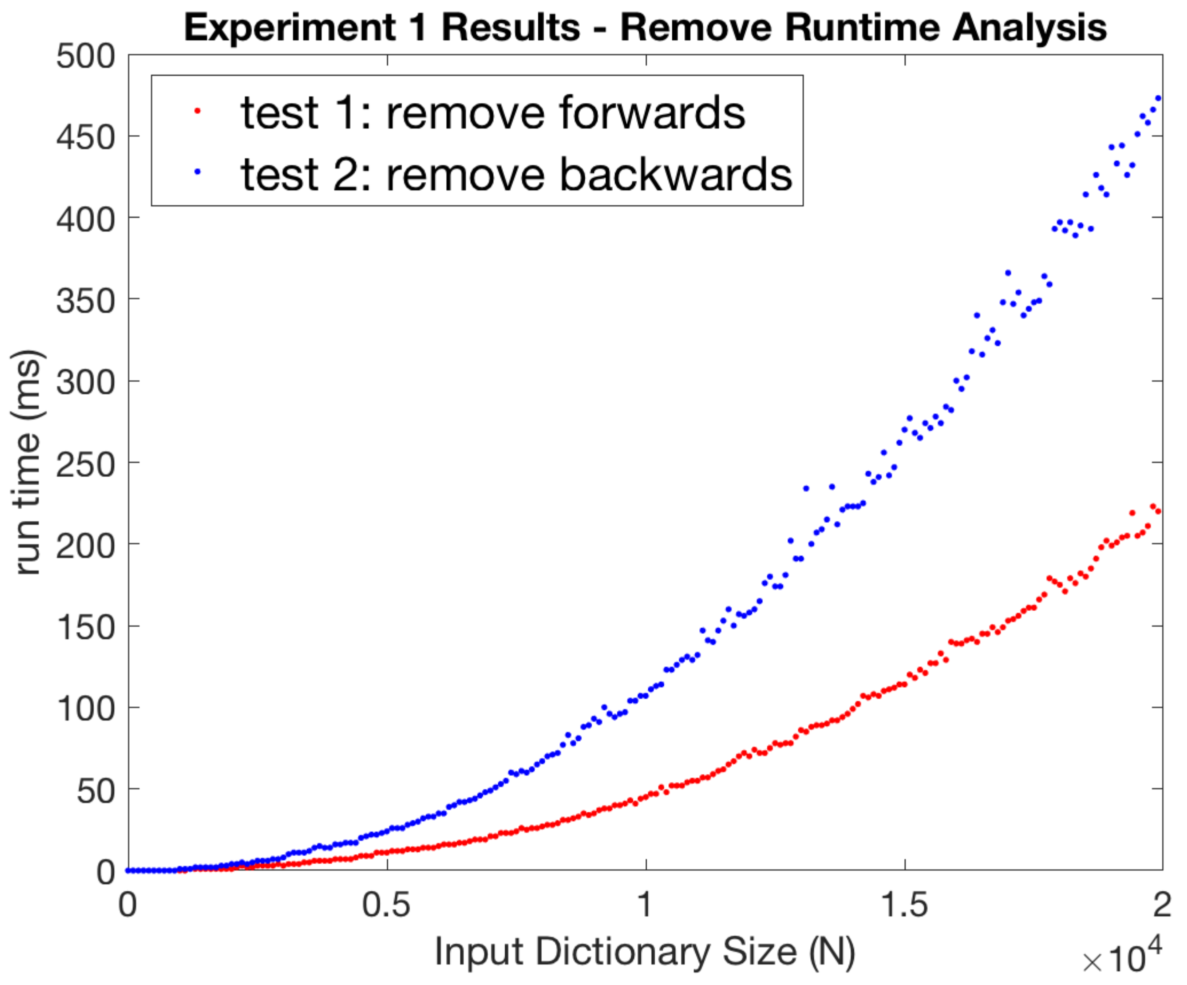
1. **While testing your DoubleLinkedList and ArrayDictionary classes, you should have discovered that your data structure needs to handle both null and non-null entries or keys. Explain how you modified your code so it could handle these two different types of inputs.**

   Justify the design decisions you made. (Your answer should be at most 1 to 2 paragraphs long).

   The only time null entries/keys became an issue was when you were trying to compare their value to something else during the indexOf or contains method. Otherwise, you are seldom interacting with these values directly. In these methods, the issue arises when you try to call .equals(). If the key/entry is null, it has no .equals() method and a NullPointerException occurs. For all non-null objects, comparison using .equals works just fine. If we want to match a null key/entry, we have to use == to compare if two values are null. Thus to handle both non-null and null keys, we just included a simple if statement checking if the key is null. If it is, check if it matches the item being compared to using ==. Else, compare using .equals() (even if the other value happens to be null, we can call .equals() as the key is non-null and .equals can accept a null parameter).

Experiment 1

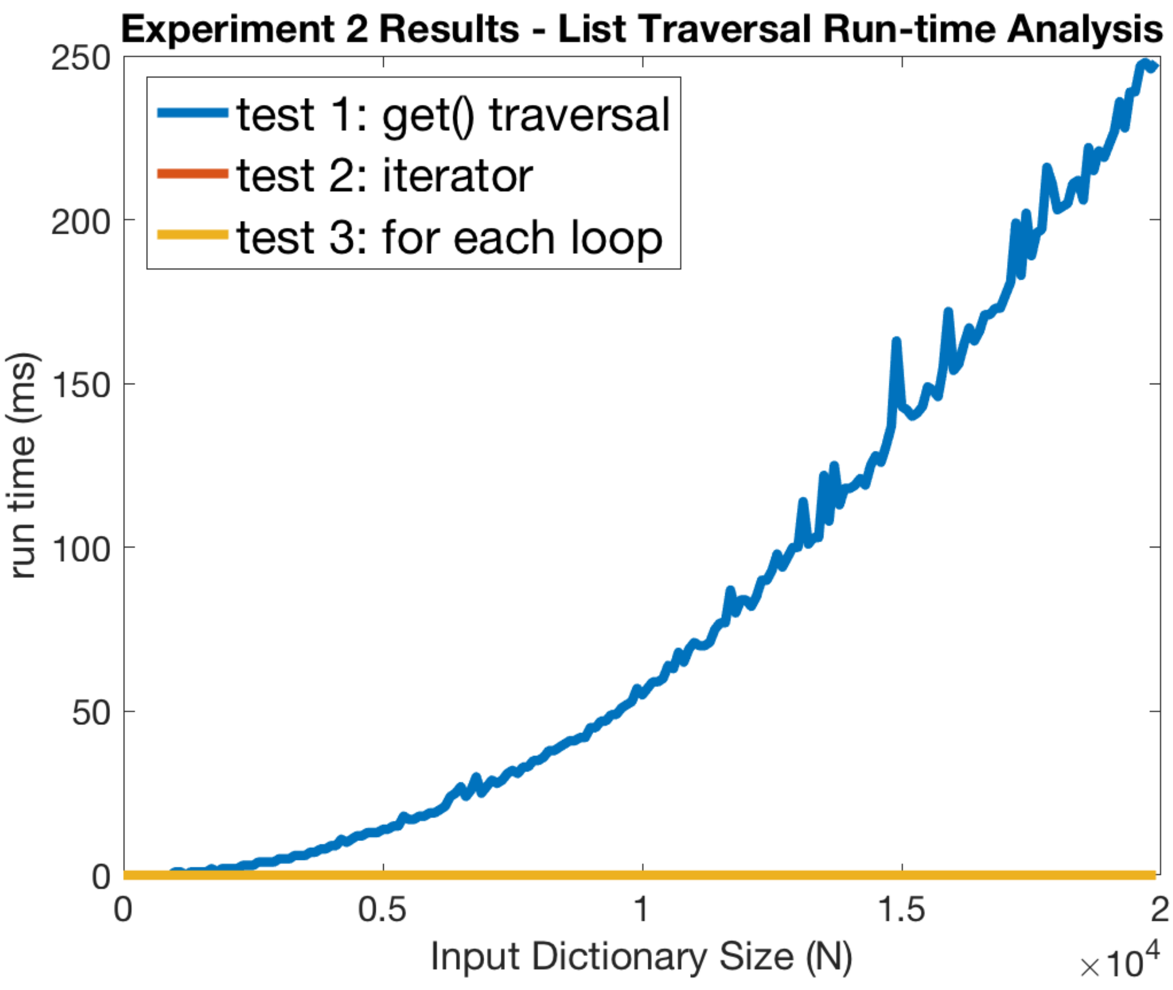
1. This experiment seems to be timing the ArrayDictionary remove method, keeping track of how long it takes to empty a dictionary of a given size. It tests the time to remove a given sorted sequence in both the forward direction and in the reverse.
2. I predict the outcome of these tests will that both tests will generate similar asymptotic behavior, as both involve fairly simple array traversals. However, when we are removing lower values (removing in the forward direction as in test 1), we do not have to traverse the list as far to find the key we are looking for as when we are removing values towards the back (test 2). Even though removing values from the front requires us to do an element swap within the array, this is a constant time operation and thus I don't think this will of balance the slowness of the traversal. Thus I predict test 1 will have a faster run time at all dictionary sizes. Overall, I expect the remove operation to be O(N) as the major operation involved is just an array traversal.
3.



4) I was correct in assuming removing from the front would be faster, but incorrectly predicted the long-term run time behavior: the runtime behavior does not appear linear. This is because I forgot to account for the fact that there are N array traversals during each trial, making the overall asymptotic runtime O(N^2) rather than O(N). The runtimes above are closer together than I would have thought. This might be because once an item is removed from the front, an item from the back is moved in to take its place. As a result, as the trial goes on, the remove from front test has to traverse more and more elements as these back elements start accumulating at the front.

Experiment 2

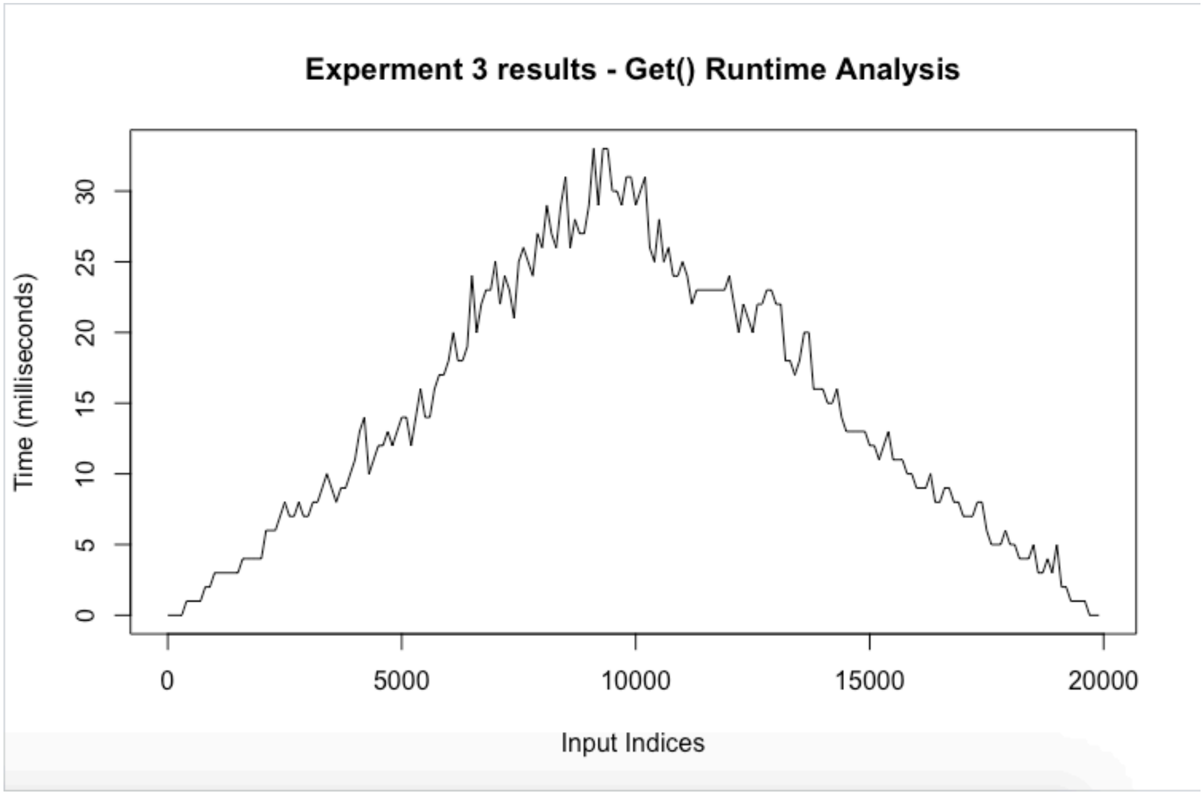
1. The experiment seems to be timing the rate at which lists of different sizes retrieve elements. The various tests time different retrieval methods: the .get method, the iterator, and a for each loop. Overall, by testing these retrieval methods at various list sizes, the experiment seems to be testing the asymptotic efficiency of these methods.
2. I would predict the get method to be the slowest retrieval method, because it traverses the whole list every time it looks for a new index. The for each loop implicitly creates an iterator, so it should behave similarly to the iterator test. As the iterator test only traverses the list forwards, and it stores its place in the list, I would expect it to be asymptotically faster than the get() method as it simply has to move one node forward every time the iterator is called. Thus traversing the list with the iterator is an O(N) linear operation, while traversing it with the .get() method requires N traversals, making it O(N^2). Thus, as dictionary size increases I expect the run time for .get() to explode above the iterator and for loop, both of which should be similar.
3.



4) I correctly predicted the relative behavior of the .get() and iterator()/for each loop run time. I was surprised to see how flat the runtimes for the iterator and for each loop were, but this might be because in comparison to O(N^2), at N as large as 2 x 10^4, O(N) is practically nothing. Even though the .get() method is optimized to search forwards or backwards

depending on what is closer to the index, it is not optimized for a traversal like an iterator is, so its performance clearly lags behind.

Experiment 3

1. This experiment seems to be testing the amount of time it takes for the ".get(index)" function for a DoubleLinkedList data structure to run in milliseconds.
2. I think that the result will be a bell shaped curve. It takes the short amounts of time to get indices close to the beginning and the end of the DoubleLinkedList as we do not have to traverse much from the back/front. But as the indices get closer to the middle, the DoubleLinkedList has to traverse a larger amount compared to when it was finding indices closer to the front and back. Because of this, I predict the time curve to be minimum at the beginning and the end, and the maximum at the middle.
3. 

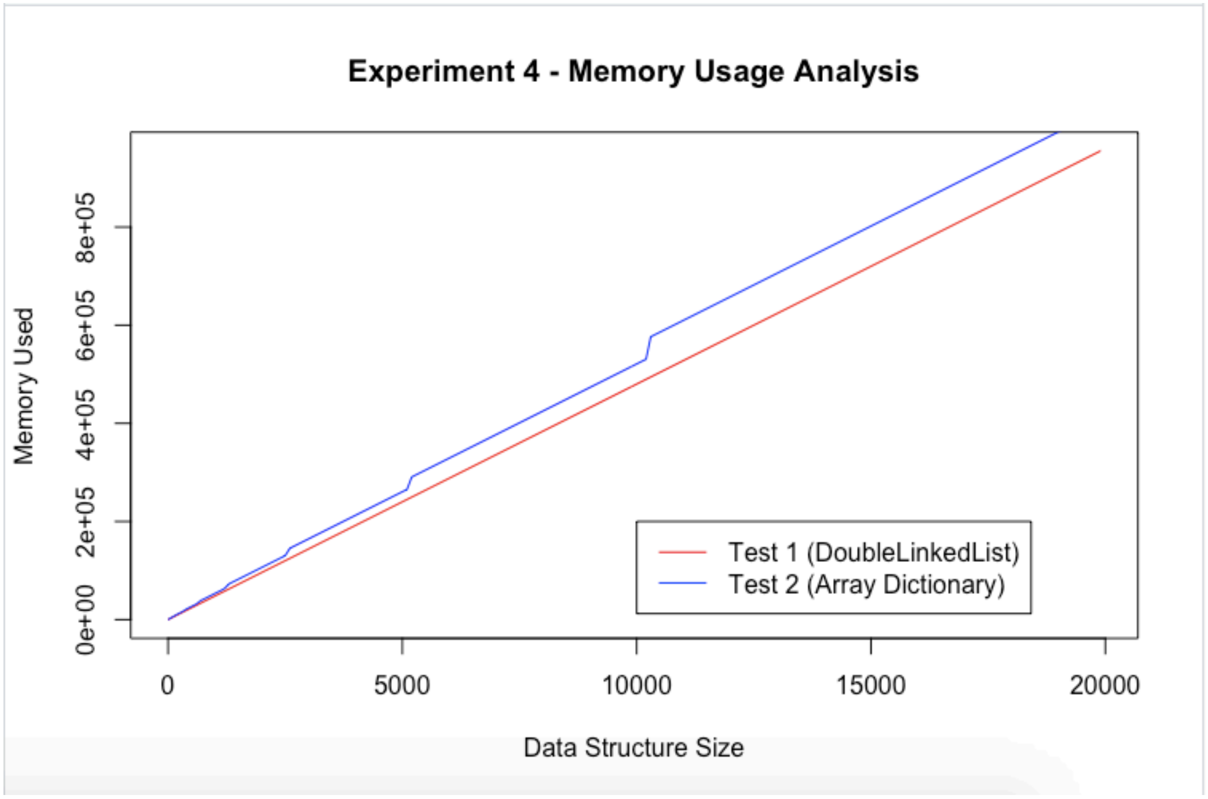
4. I correctly predicted the outcome as a bell shaped curve. This is so because in our DoubleLinkedList we have a pointer at the front and a pointer at the back. This makes it easier (faster) for the computer to "get" the value at indices that are closer to the front and the back because they do not have to traverse too much (hence minimal time taken to run). But as for the indices towards the middle, the computer has to traverse all the way to the middle from the front or the back of the list making it take a longer time (hence giving us the bell shaped curve with maximum at the middle).

Experiment 4

1. The experiment consists of 2 tests. The first one is testing the memory used for creating DoubleLinkedLists of different sizes and the second one is testing the same for ArrayDictionarys.
2. I predict that in the case of the DoubleLinkedList, as the size of the list increases, the memory used up increases too. This happens in a linear fashion. In the case of the

ArrayDictionary, as the size of the dictionary increases, the memory used up increases. This too happens in a linear fashion.

3.



4. I predicted the results of the experiment correctly. In the case of the DoubleLinkedList as the size of the list increases, the memory used up by the computer increases in a linear manner. This is partially true for the ArrayDictionary. The only slight misprediction that I made was that the memory used by increased in a linear manner. It seems that every time the size of the dictionary increases twofold, the is a slight non linear spike in the amount of memory used.

Extra Credit:

derive command

1. **Description**
   This method will find the derivative of a given expression. It can handle derivatives of any of the defined functions, including the natural log function that was implemented to help handle some cases within the derivative method. It handles chain rule, product rule, quotient rule, power rule, and the other common derivative forms.
2. **Use Instructions/Examples:**
   For this method to work, the given expression must contain an undefined variable.
   - derive(expression, variable_of_interest)
   - derive(cos(x), x) → -(1*sin(x))

i.If x is defined, an evaluation error will be thrown

Note, however, that if you try the derivative of x^2 you will get a very messy output

- derive(x^2, x) → handleSimplify(x ^ 2 * (2 / x * 1 + 0 * ln(x))) → x^2(2/x)

If you simplify this out by hand, you will see it gives the correct value. The reason why this is so messy is because the derivative techniques I use are generalized to work for any complex function. The extra terms are a result of chain and product rule. I added an extra step to handleSimplify (see the private method checkCommSimps) that removes some of these extraneous expressions involving multiplication by 1 or 0, addition of 0, and the likes. The final product is still a bit messy, but this messiness allows me to calculate some pretty neat derivatives. For example:

- derive(x^x, x) → x ^ x * (x / x * 1 + 1 * ln(x)) --> x^x(x/x+ln(x)) = x^x(1+lnx)

If the expression being differentiated contains any variables that are NOT the variable of interest (x in the above examples), those variables must simplify to a numeric value using defined variables, or it must simply to some function of x (the variable of interest). Else an evaluation error is thrown. Here is an example

- y := x^2
- derive(y^2, x) → x ^ 2 ^ 2 * 2 / x ^ 2 * x ^ 2 * 2 / x = 4x^3

3. **Miscellaneous**

In order to implement this method, I had to define a natural log function. For continuity, I also included this function in the toDouble method, so any derivatives can be evaluated by storing them in some variable. For example, if I wanted to evaluate the derivative of x^2 at 4, I would use the following commands:

- y := derive(x^2, x)
- x := 4
- toDouble(y)

In order to simplify the output of derive, I added an extra step in handleSimplify that reduces common expressions involving 1 and 0 (i.e. 1*x = x, 0*x = 0, 0 + x = x, etc)

solve command

1. **Description**

This command will iteratively solve for one possible solution to an unknown variable in
a

given equation. It can handle equations including any of the defined functions, although may run into issues if the solution lies close to an undefined point. If a solution cannot be found after a set number of iterations, an evaluation error is thrown: solution did not converge. A solution is found by converting the given equation into a new one where all terms are on one side, and the other side is 0 (i.e. subtract the right side from the left side and set it equal to zero). So if you gave the equation 3x + x^2 = 1, it would first convert this into 3x + x^2 - 1 = 0. Then, it applies the Newton-Raphson method for finding roots of the real-valued function off an initial guess provided by the user.

2. **Use Instructions/Examples:**

- General command: solve(equals(expression1, expression2), variable, guess) Variable is the variable you are trying to solve for in the equation, and guess is an initial guess for what the solution may be. If there are multiple solutions, guess will usually influence which one is found.The command will throw an evaluation error if either of the two expressions contains undefined variables, and will not produce a solution if the expressions do not contain the given variable. Here are some example calls:
- solve(equals(3*x, 1), x, 1) → 0.33333333333333337
- solve(equals(ln(y),1), y, 4) → 2.718281828269428

2. **Miscellaneous**

- I had to define the equals(expression1, expression2) operator to establish this command. For continuity, I added this to the toDouble method. It will return 1 if the two expressions evaluate to the same value, and 0 otherwise.

    - This command lies on the derive command as part of its iterative formula. Thus any errors the derive command can run into, this one may as well.