CSE 373: Data Structures & Algorithms

Project 1 Group Writeup

Emily Ding & Pengfei He

Question 1

First, check the null first in order not to have null pointer exception while using the method of the potential null values. In other words, don't use any method of the entry that is potentially null because it will crash the program by causing NullPointerException.

Second, during the check, use the "XX == null" to check if it is null value and then use equals method later to check if the values are same if the entries are non-null values. And two checks can be combined by "&&" or "||" depending on different cases, but "==" check should be at the front of equals method.

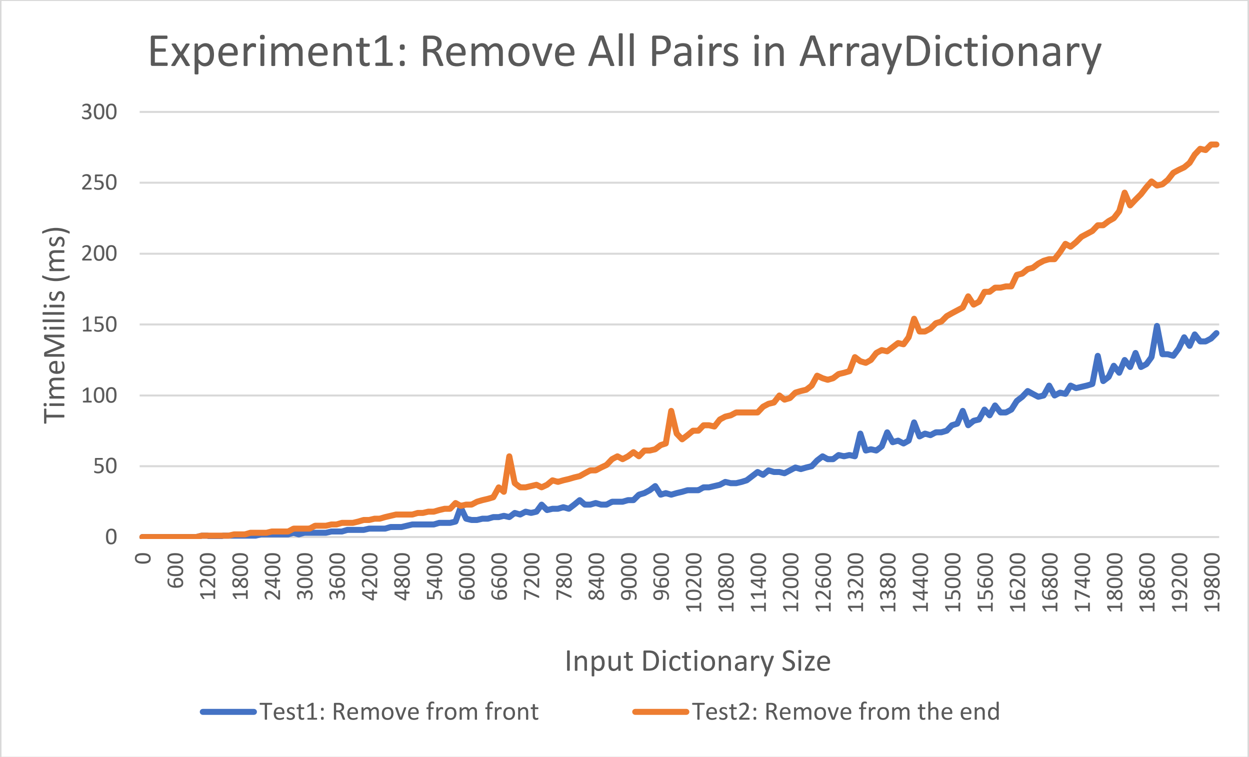
Question 2

We analyze them experiment by experiment. For example, for experiment 1, we have answered the four small questions 2.1, 2.2, 2.3, 2.4. Later on, we will analyze the next one and so on.

- Exp1:
  - 2.1 Test and compare the runtime of removing all pairs at the front and the end in ArrayDictionary.
  - 2.2 Prediction
    - For test1, the method deletes the pair whose key is increasingly built in the for-loop. According to the testing code, it will keep deleting the first node since the method makeArrayDictionary build each pair incrementally. Since it keeps deleting the first node, they only check once each time of the remove method in the for-loop. Therefore, the runtime is the single remove operation multiply the total times of for-loop. In other words, O(1) * O(n) = O(n).

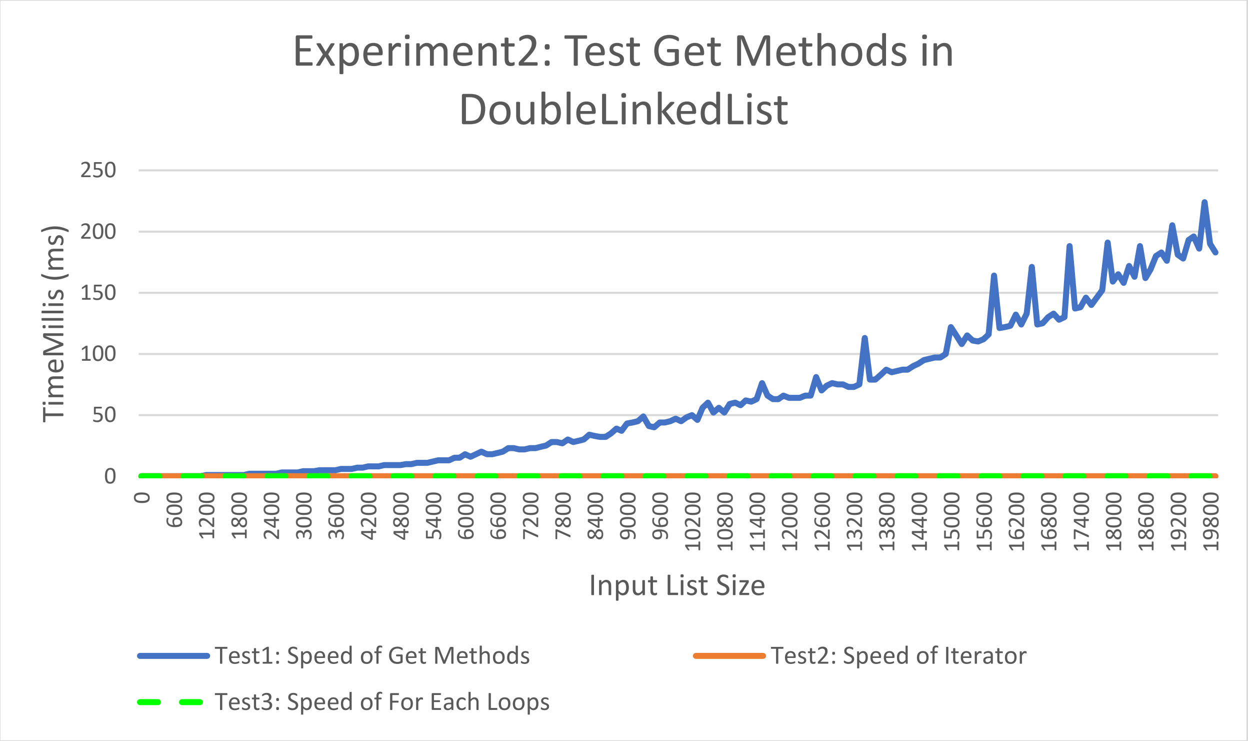
  - For test2, the method deletes the pair whose key is decreasingly built in the for-loop. According to the code, it will keep deleting the last node since the method makeArrayDictionary build each pair incrementally. Since it keeps deleting last node, they need check through the array from the beginning each time of the remove method in the for-loop. Therefore, the runtime is the single remove operation multiply the total times of for-loop, in other words, O(n) * O(n) = O(n^2). But if we consider the decreasing array size, the runtime would be O(n(n-1)/2), and after simplification, it would still be O(n^2). Based on the analysis, test2 will be significantly larger over time.

- 2.3 Figure



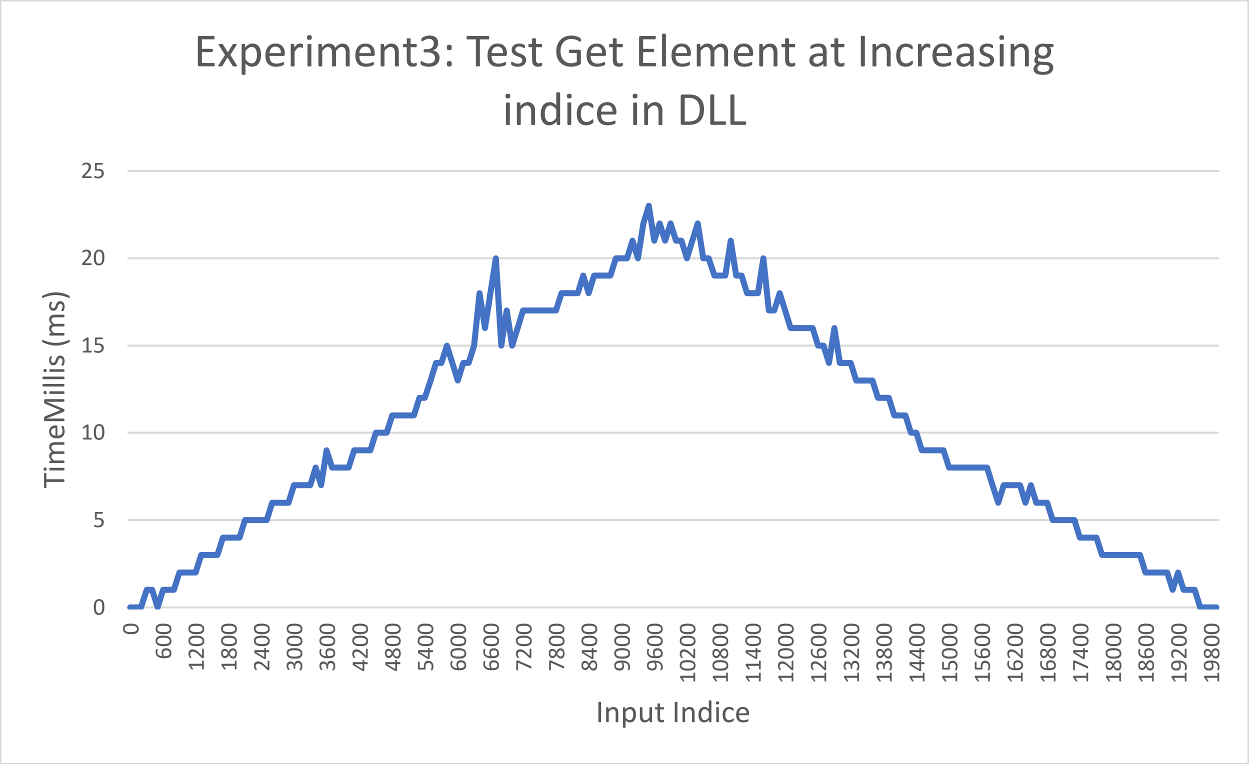
- 2.4. After the test, both tests result in accord with my hypothesis as shown in the picture. The orange line is higher than the blue line. And the blue line grows the purple line linearly grows quadratically. Because removing from the front just needs to check the first element and remove but removing from the end needs to traverse the whole array to check and finally delete the end.

- Exp2:
  - 2.1 Test get method and iterator in DoubleLinkedList.
  - 2.2 The runtime of the worst case of the get method on average is O(n/2) since the movePointerHelper in the DoubleLinkedList can smartly choose which end of linkedlist to start to traverse. For test1, the runtime is the time of each get method multiply times of the whole for-loop. O(n/2) * O(n) = O(n^2/2) -> O(n^2). For test2, the runtime is the time of the interaction time. O(listSize) -> O(1) it traverses the whole DoubleLinkedList. For test3, the runtime is the time of the interaction time. O(listSize)-> O(1). It traverses the whole doubleLinkedList in constant time. Therefore, test1 result would be significantly larger than that of test2 and test3. Plus, test2 and test3 should have the same results because they both use iterators.
  - 2.3 Figure



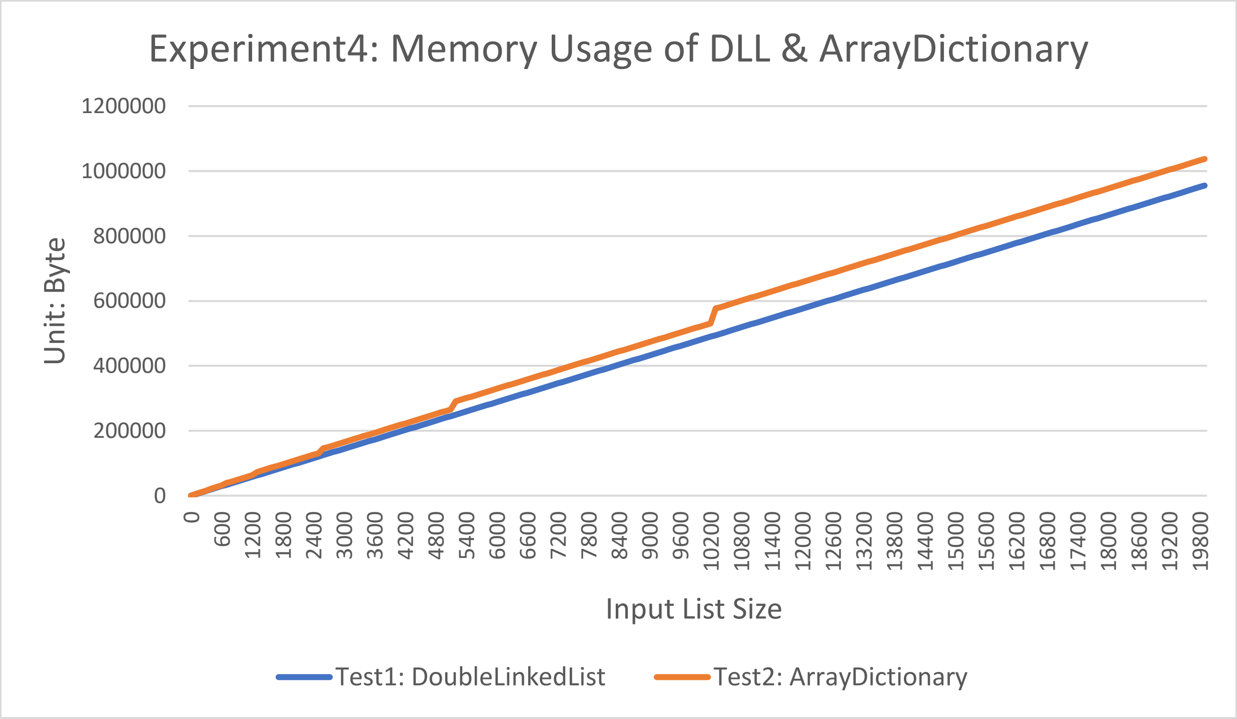
  - 2.4. After the comparison, we found it accords with our expectation. The two methods of test2 and test3 use the same iterator, and it traverses all element at a constant time. And the test1, as I calculated before, the runtime would be each runtime of single get method multiple times of for-loop which would cause quadratic runtime. However, we didn't expect that the whole test2 and test3 would be so fast as they keep at zero, and two lines overlap at the x-axis. This is because of the efficiency of the constant runtime.

- Exp3:
  - 2.1 Test get element at increasing indices in DoubleLinkedList
  - 2.2 For each test method running, the get method runs 1000 times, and each time it will try to get every data in the 2000-length DoubleLinkedList. The runtime of the get method in the worst case is (n/2) because of the movePointerHelper method as we stated before. O(n/2)*O(2000)*O(1000) -> O(n)
  - 2.3 Figure



  - 2.4 After comparison, we found that we forgot to consider that we implemented the efficient movePointerHelper method in my algorithm, and I only concluded a general result. And it is not just a straight line which we assumed it is O(n) as it varies as the input indices increases. In reality, the closer to mid of the list, the larger runtime it will be. Because movePointerHelper method will only start from two ends instead of mid-index because there is no node to start with at midpoint, and the midpoint is most far away from the two ends.

- Exp4:
    - 2.1 Test the memory space used by ArrayDictionary and DoubleLinkedList.
    - 2.2 In general, each element on DDL or ArrayDictionary needs to store the data based on the generic data type. For test1, DoubleLinkedList memory usage, we think for each node in DDL, it uses slightly more memory space to store the relative positions which are the prev and the next. And it grows linearly as the larger list it is initialized. For test2, ArrayDictionary its own is a list of pairs which store key and value. It depends on what object it stores. Since both functions stores Long value instead of “heavy” objects with many data, so that makes little difference in memory spaces. But it uses extra memory spaces if it is not filled up. As the array grows larger, every time right after it doubles its array size, it will use a lot of extra space(nearly the half of the array).
    - 2.3 Figure



- 2.4 By comparison, the memory spaces of the two are very similar and unexpectedly the memory space of each element makes little difference. What follows my expectation is that there are some bumps in the memory space usage of ArrayDictionary because every time it doubles its size, there are many wasted memory spaces. As it grows before the next doubleSize behavior, the memory space increases linearly only because more pairs are added in. On the other hand, the memory space of DDL is much more stable because it uses minimal memory space without the extra usage of memory space overall.