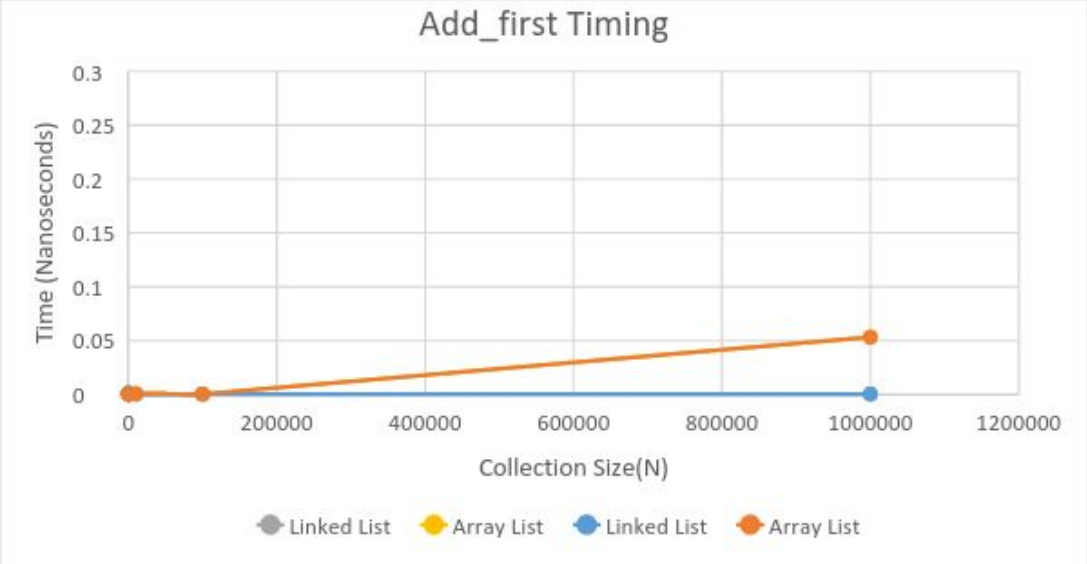
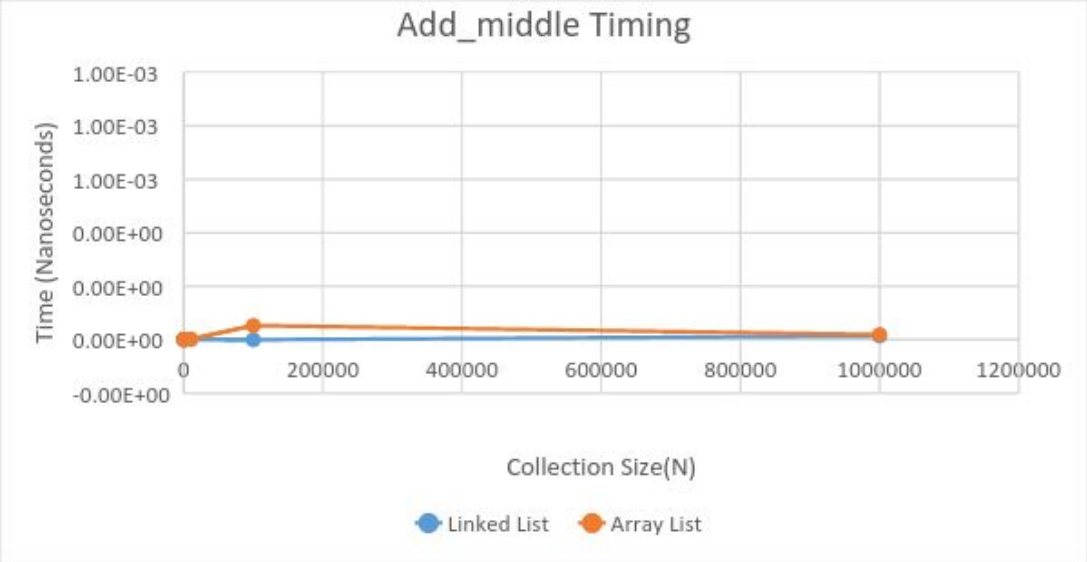
Tony Diep
2/23/17
CS 2420
Algorithms and Data Structures

Analysis: The Linked List

What the Graphs Tell Us

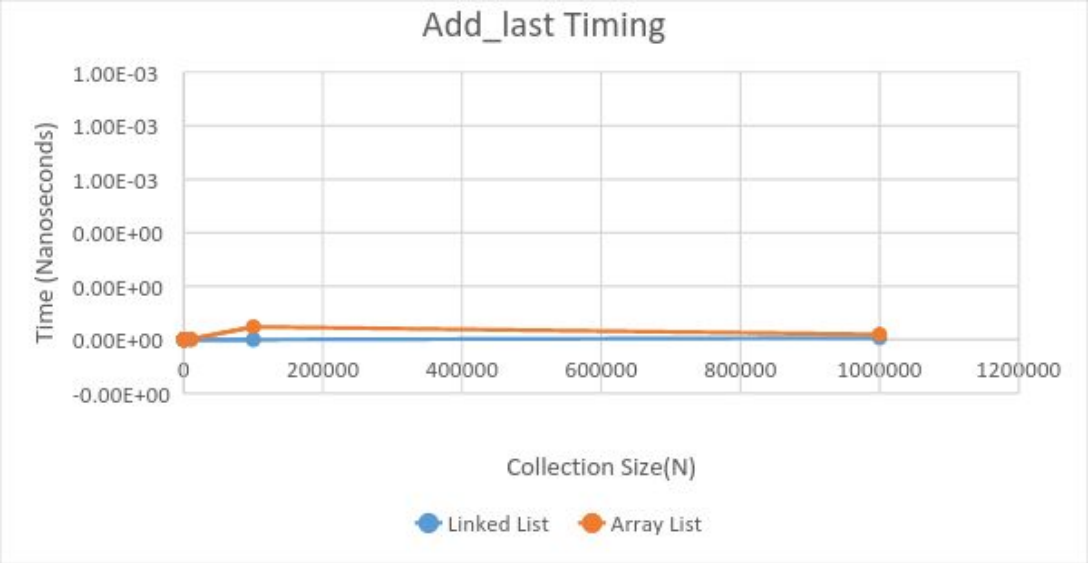


The graph above above shows a constant complexity (O(1)) for Linked List and a somewhat representation of O(N^2) for the Array List complexity. This is because Array List contains a backup array that needs to dynamically grow and shrink. Linked Lists do not require any qualifications before adding elements since each node has a data and already a reference to another node. This shows that adding elements to a Linked List is simple and efficient.

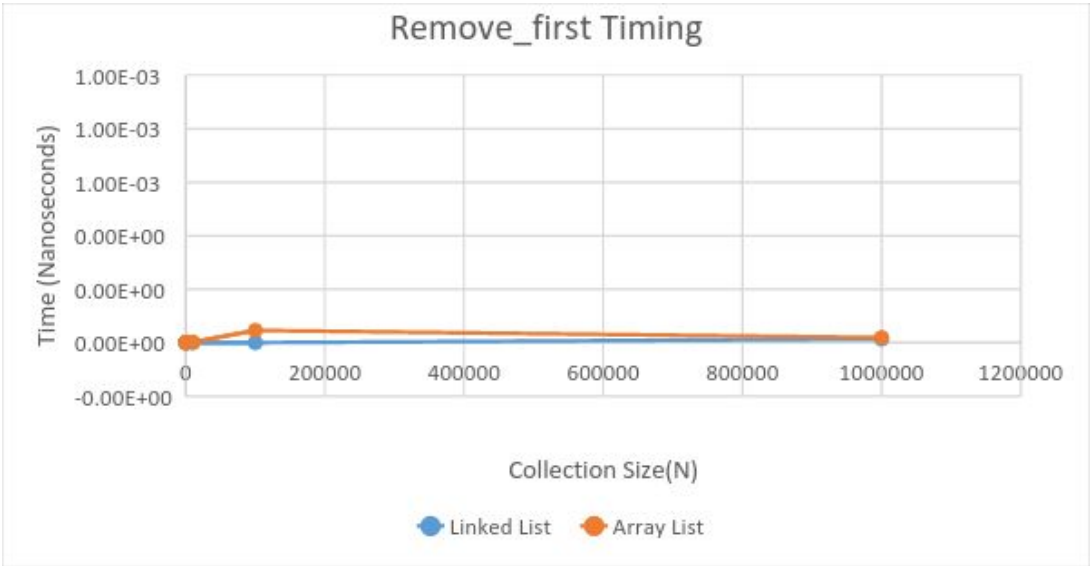


The add_middle() graph shows that the Linked List has relatively more constant

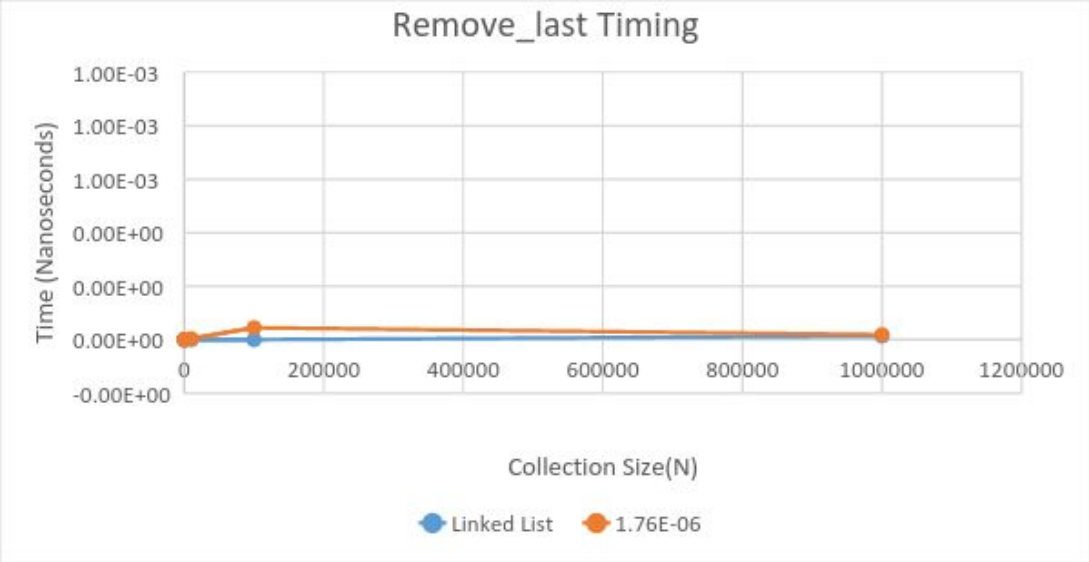
complexity compared to the Array List. This is because the ArrayList requires the backing array to shift all the current elements to the right in order to make room for newly added elements. However, the Linked List simply creates a temporary node in which the previous node must change its pointer so that the pointer references to the temporary node. Then, the temporary node that carries the newly added data will point to the next node. These Linked List operations all take constant time to perform as opposed to shifting elements in the backing array to make room for newly added elements (which takes N complexity).



This graph shows similar results from the “add_first()” method in that the Linked List showed more constant complexity compared to the Array List. For the purposes of this analysis, we ignore the case of when the Array List backing array must expand if the backing array is full prior to adding newly added elements.



Removing the first element both take constant complexity for Array List and Linked List. The Array List essentially moves the start pointer back to the very beginning of the virtual array. It contains three pointers: one being the start index of the virtual array, the second being the end index of the virtual array, and the final pointer being the end of the whole backing array. The Linked List has three nodes which similarly like pointers.



Removing the last element takes constant complexity for both lists. This is the expected behavior and matches what the graph displays. Since the Array List implementation has an end index, accessing the last element takes constant time as opposed to traversing through the whole backing array until reaching the end (which takes N complexity).

The Class Hierarchy

Each of the different classes required in this assignment all relate to each other. First, we have the List_2420 interface which represents all methods that are required for every type of list that exists. All lists are required to have functions that allow us to add and remove an element at the first and last positions as well as check if an element exists in the list and clearing all existing elements in the list. The List_2420 interface allows child classes to closely relate to other child classes. This is where the Array_List_2420 and Linked_List_2420 classes come in because they are both essentially types of Lists. They have similar implementations and visualizations but with moderate differences. For example, they are both required to add and remove elements at first and last positions and check if an element exists in them, but their behaviors are somewhat different from each other. In other words, the way the Array_List_2420 adds and removes elements at the first and last positions is by using a backing array and thus using indexing to store the elements in the backing array. On the other hand, the Linked_List_2420 class does not use any backing array but rather a Node static class to represent a box that has a data storage and a reference to the next Node. Of course, we will

need to verify the correctness and efficiency of the implementations between the three classes mentioned above. Thus, that is where we also create a List JUnit Test class as well as a Timing class to help test for timing between the List_2420_Test and the Linked_List_2420 classes. Going back to the Linked_List_2420 class regarding the “Node” static class, the Node class supports the Linked_List_2420 class. Conceptually, a Linked_List comprises of various Nodes that have links/references to each other which results a long chain of nodes. It would certainly be challenging to implement the Linked_List_2420 class without having the Node class because we would have to make the Linked_List_2420 class represent all of the nodes (despite how many potential nodes that can exist in any given Linked List) in the Linked List. We can mitigate the difficulty by creating the Node helper static class. The question is why we require to make the Node class a static helper class and not just a helper class. This is because we need to ensure that no single Node has overlapping access to the data of any other non-adjacent node. Every node should have access to the current data held as well as a link to the next Node but nothing more. Additionally, the Node class is an inner static class, meaning that the class is contained inside of the Linked_List_2420 class.

Iteration vs. Recursion

Let us define iteration and recursion. Iteration is how many times a loop (i.e. for loops, while loops) will execute depending on the longevity of a certain condition to be true. Recursion is the mechanism of a method calling itself a certain amount of times using various amounts of stacks. Recursion consists of a base case and a recursive case. The recursive case is what is iteratively doing the self-calling. The base case(s) will then be the stopping point for the recursive case because if the recursive case were to continue, this will cause a StackOverflowException, meaning the computer that ran the program ran out of stack frames to use when calling the methods iteratively.

When it comes to calculating the size of a list, there are two ways to solve the problem. First, by iteration, we can use looping such as a while loop and create a integer field called “size” in which we can update every time we set that a node holds a data in itself. The second way, which is by using recursion, is setting up the base case (that is, if the next Node does not have any reference to another node but is referenced to “this” current node), in which we return 1 because we are only counting for the current node. The recursive case would have to be counting the current node plus the next node. The recursive case is known as “tail recursion” because the recursive case consists of a constant as well as the recursive call itself added to the constant. Of course, these two solutions are situational. For example, the recursive method should likely be used if the linked list size is not immense because it relies on adequate amounts of stack frames which may not work for all computers. Thus, the recursive should be used for moderately or smaller sized linked lists. However, the iterative solution has more capability because it can safely work for varying linked list sizes, ranging from small to large sizes.

Besides calculating the size recursively, there are other recursive elements contained in this project. The Node class is recursive since it contains a data as well as reference to its

neighboring Node. If we look at the implementation of the Node class, we can see that one of the fields has the data type of itself. There are two base cases for the Node class: 1) There only is one node and therefore contains no references to other nodes, and 2) The tail node (the last node from the Linked List) does not contain other reference to another node.

I am confident that I understand recursion conceptually. However, the only issue with recursion is ensuring I implement the base case(s) and recursive cases correctly because in the past, I encounter a lot of StackOverflowExceptions. In essence, it is important that recursive methods cover sufficient base cases before these StackOverFlowExceptions occur.

Software Development Log

This week, we spent 18 hours working on this assignment. This was because we encountered small yet drastic errors in our code. Nonetheless, we implemented lots of JUnit Testing to ensure that our program is working excellently.

What took the majority of our time on this assignment were, again, the small bugs in our code (i.e. logical errors and exceptions such as NullPointer, ArrayIndexOutOfBounds, and StackOverflow). The Array_List class was somewhat challenging than expected even we have implemented a similar class back in CS 1410 but when it came to the Linked_List, it was not too difficult. Although we did start our JUnit Tests prior to implementing the classes, perhaps we should consider drawing pictures to pinpoint expected behaviors of the Linked_List methods and the Array_List methods.

Regression testing is a software testing mechanism that when a program was previously implemented to begin with yields the same working behavior after adding changes to the program. In our case, we are given the files to start, then before making any implementations, we are required to implement JUnit Tests. This mitigates countless hours on the assignment than necessary because by doing JUnits first, we as programmers are capable of getting general outlines as to what the program should be expected to behave.

Writing JUnits first certainly has helped us excel in the assignment because they do take a long time to implement. Luckily for us, JUnits do not really stress us that much and rather they are entertaining. Normally, lots of programmers dislike JUnits because of how much stress it can build when the program so called “does not work.”

Thought Problems

The singly linked list does not work well for a queue data structure. This is because the singly linked list can only traverse forward and not backward. Each node in the Linked List, starting at the head node, may or may not contain an element inside, and each node may or may not contain a reference to another Node. Conceptually, the queue allows us to push (add an element) and pop (remove an element) but only those operations apply to the topmost element on the stack. Consider we want to push three elements and then pop one element off the queue. We can think of pushing elements as traversing forward and popping elements as traversing backward in a Linked List.

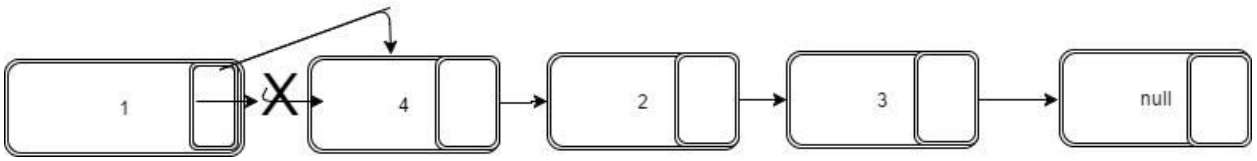
One way to fix this problem for the singly Linked List is to create an additional node that represents the previous node. Recall that the Linked List implementation has the “head” node

and the “tail” node. If we added a node representing the current node’s previous node, we can keep track of the previous node as we traverse forward.

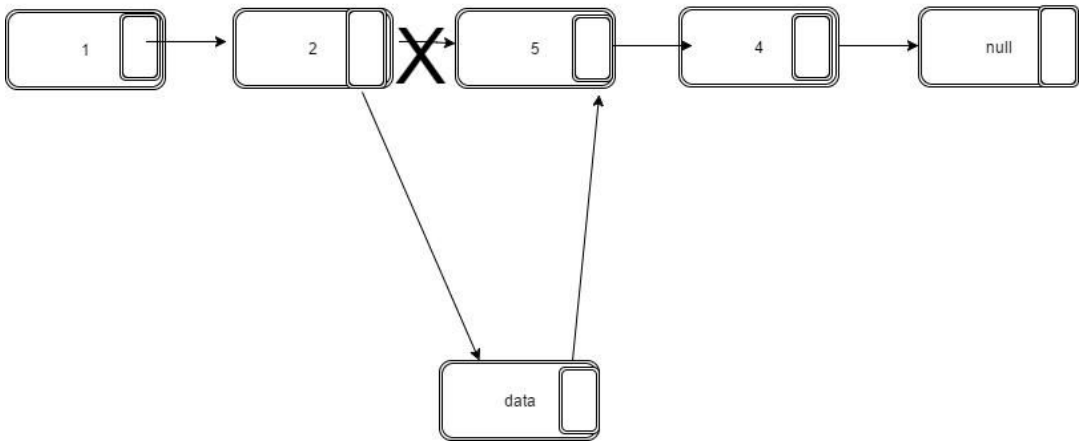
This assignment is not necessarily difficult but rather requires a high demand of pre-planning and visualizations. For example, we are required to provide visual documents (i.e. Draw.io) and provide JUnit Tests to at least get a good start on the assignment. After doing this assignment, I learned that it is strongly recommended to complete the preliminaries.

Linked List

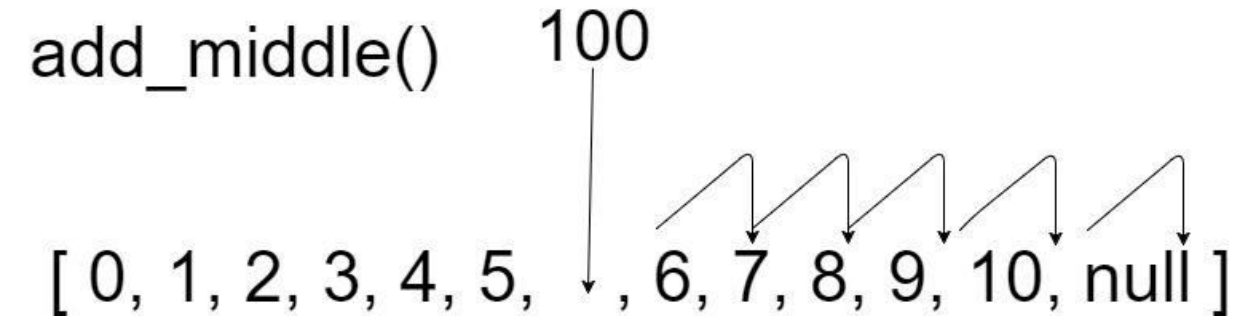
remove_first()

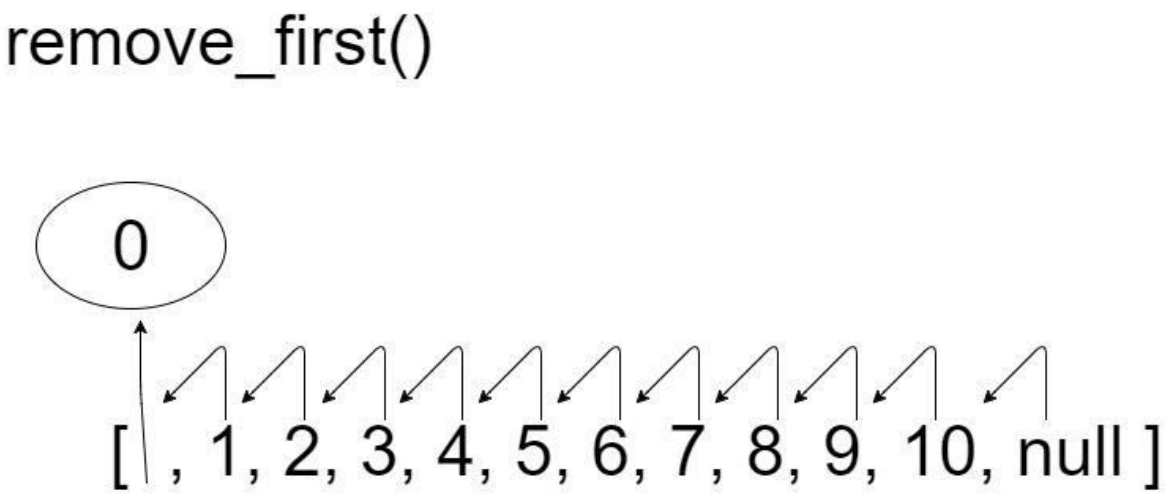


add_Middle()



Array List


add_middle()
100
[0, 1, 2, 3, 4, 5, , 6, 7, 8, 9, 10, null]


remove_first()
0
[, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, null]