Jong Tai Kim, Joy Phillips

Writeup:

1. We modified our original naive if statement to an if statement that considers all logical cases of given keys. We make sure to avoid null pointer exceptions when we try to compare the null key with the keys in the list. At first, we put null check in the if statement then we decided to perform a direct comparison between entries and items in the data structures by checking if the key passed in `==` the key at a given point in the structure or the structureKey.equals(non-null Key) (not key only equals null but key in general equal other keys).
2. If the given parameter is strictly equal (`==`) to null, it is only true when both are null, such that it will skip the object comparison (.equals(obj)) and throw NullPointerException. It is because of this bug that we decided to generalize the if test to be `structureKey == paramKey` along with the modification to the object comparison of checking that the paramKey was not null when comparing using `.equals()`. This way we took into account all cases of keys passed into the method. If neither keys are null, we test both parameter and element with object comparison. If only one is null, that means our `structureKey == paramKey` will find that both are not equal to each other and the if statement is false and next statement ((structureKey != null && structureKey.equals(paramKey)) will prevent the case where the null element is compared to param, which causes null pointer exception and null parameter cannot be equal to non-null element in object-wise if element is not null. This way, we can avoid NullPointerException.
3. No question to answer
4. Those experiments are similar to what we did in jUnit test on efficiency; all of experiments test on efficiency of our data structures, double linked list and array dictionary. To measure those, the test make the average value for the result of 5 trials.

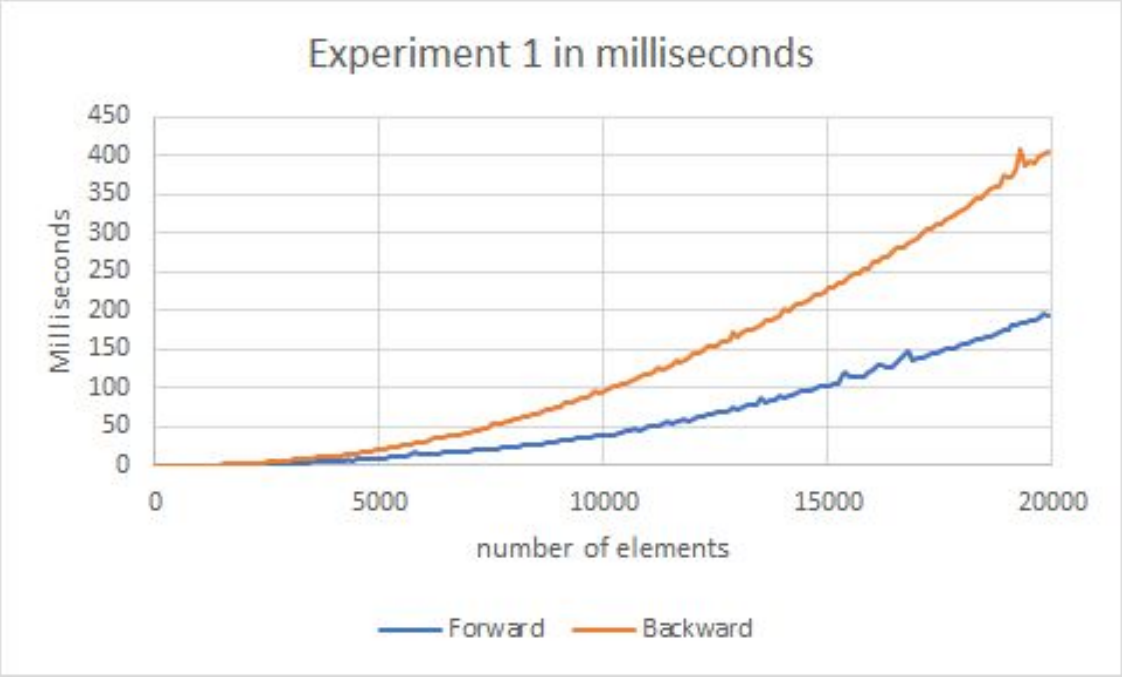
Description/Prediction/Result

Experiment 1

Description: This experiment is traversing data in forward order (from smallest key to largest) and in backward order (from largest key to smallest) while using the remove method for Array Dictionary. It will test on different size of array dictionary to check how runtime changes over the size of dictionary.

Prediction: The data is stored in order from smallest index to largest index.We predicted that our data structure will be slower when it is traversed from largest key to smallest and faster the data is traversed from smallest key to largest. We believe that because the way the experiment put pair of key and value in order --smallest key in smallest index and largest key in largest index-- and because we implement remove method to find key starting from smallest index to largest, it is inevitable that we are required to traverse from 0 to large elements to find large keys while we find the small keys relatively fast due to how the data is structured. .

Plot Graph:



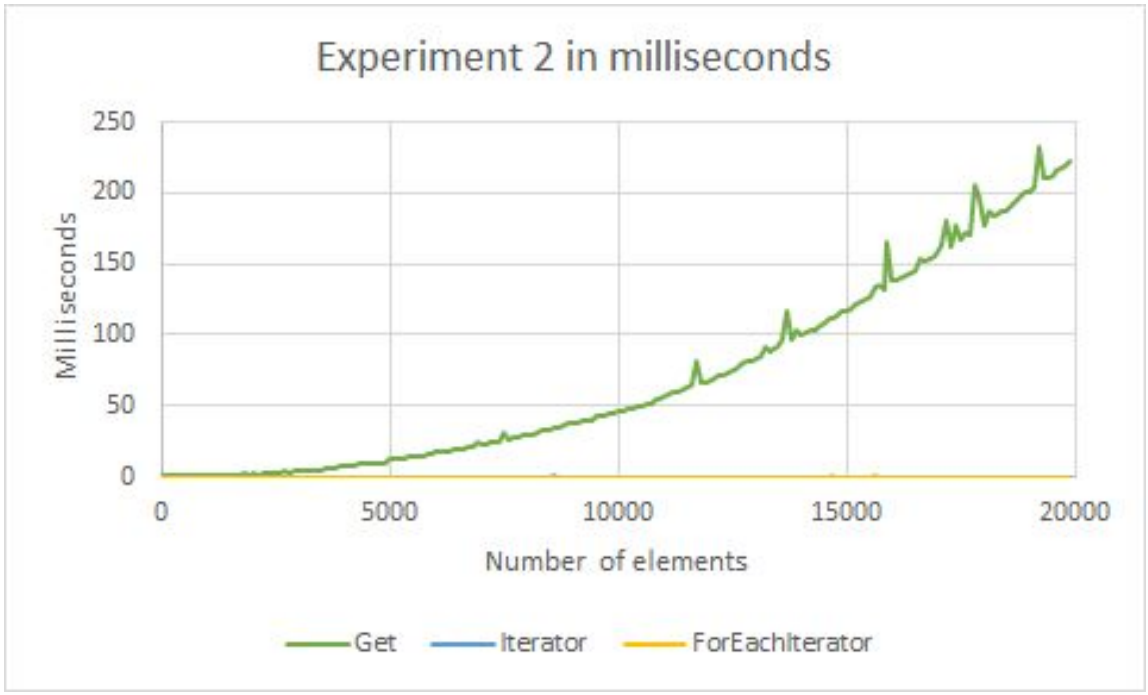
Results Reflection: As we have predicted the backward was slower than forward order. It is around runtime of backward order is double the runtime of forward order.

Experiment 2

Description: In this experiment we are testing on different methods on Double Linked List that traverse the data using a standard for loop using get method of list, and iterator with a while loop, and iterator with a for each loop.. Same as experiment 1, this experiment will test runtime of those 3 methods as size of Double Linked List grows.

Prediction: The iterator and for each loop will be "faster" in runtime than just calling the get method. We believe this because the iterator only traverses by checking current node and then its next node for traverse, while get method traverses from 0th element to the current element, which indicates summation from 0 to data size, yielding quadratic runtime.

Plot Graph:



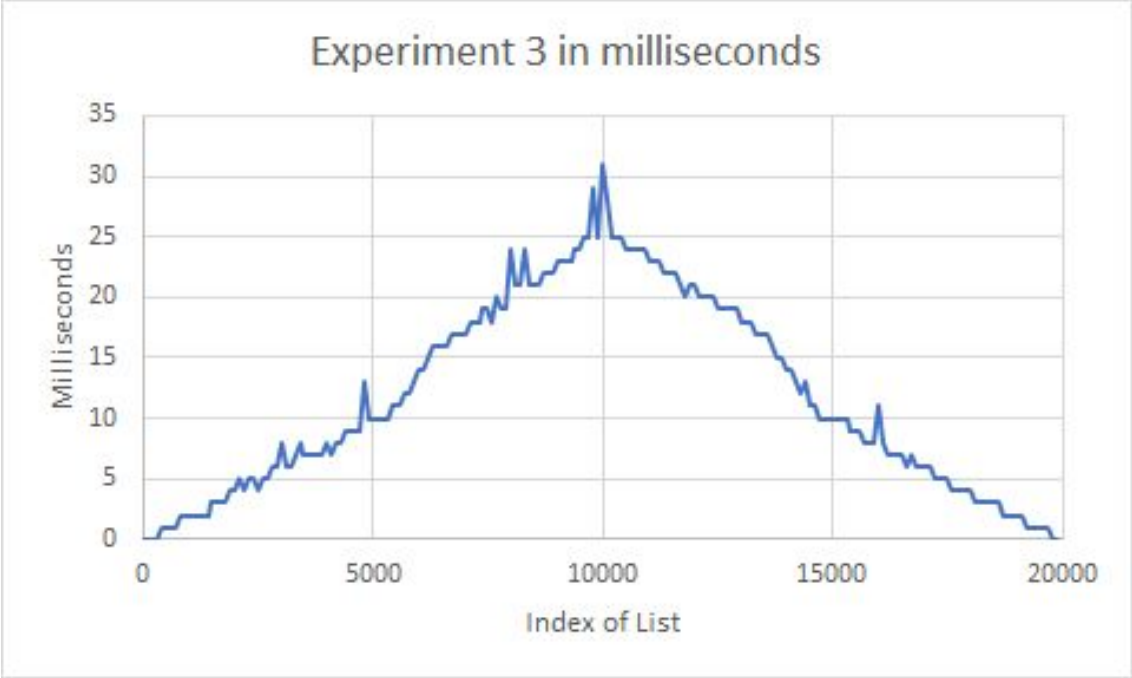
Result Reflection: As we predicted, the iterators are very efficient at finding stuff compared to the get method, and it was way faster than we have expected. It is because that traversing by current and next node is very efficient for linked list, while for loop is not efficient for linked list. It is intuitive that linked list is implemented to use the nodes, not looping by its indices.

Experiment 3

Description: The experiment is testing on how efficient our double linked list is to traverse to the elements at given indices for multiple times. It will track the runtime for traversing each index.

Prediction: We believe it is inevitable for us to have trouble to access the middle indices when it comes to double linked list because our code traverses the list from either the front or back node of the list, meaning that the code will run fast to find first and last elements faster while the worst runtime will be on the middle elements. The code's algorithm is to divide the data in half for the sake of faster traverse by half left of the data is managed by front and its next nodes and the other half is managed by back and its previous nodes and middle inevitably becomes last nodes for both cases.

Plot Graph:



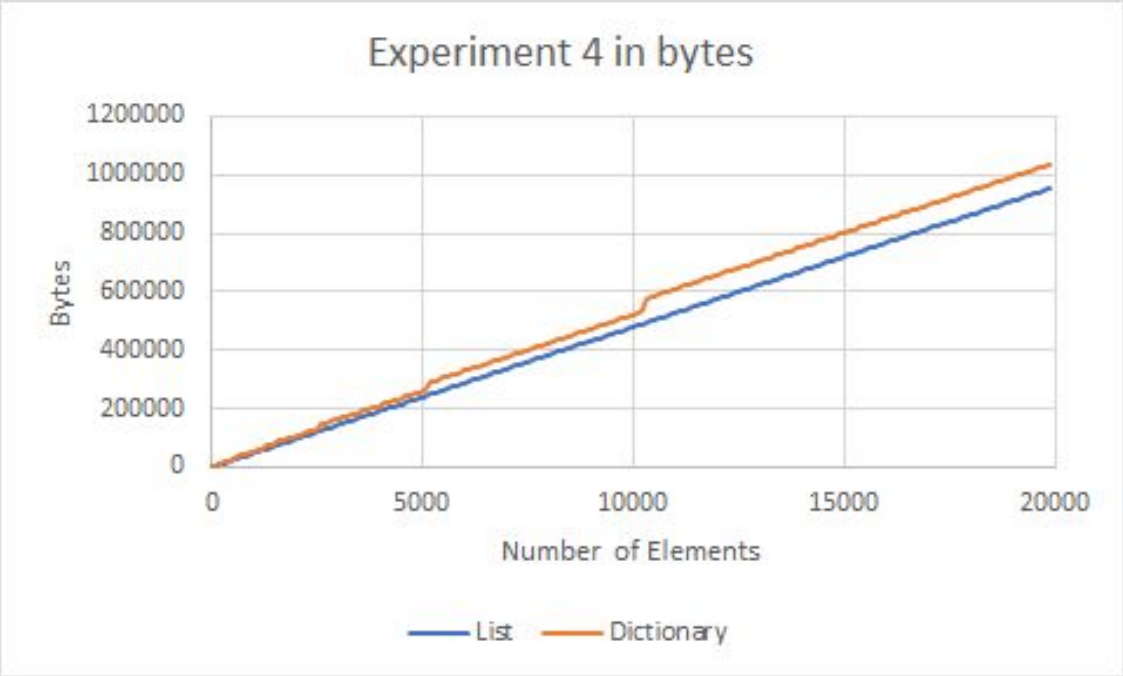
Results Reflection: as we expected, the data is really slow around the middle indices.

Experiment 4

Description: This experiment tests on how the double linked list and array dictionary are stored in memory as their size grows.

Prediction: We are testing on the memory usage of double linked list and array dictionary with growing on size. (they both have same initial size and same incremental on size). The prediction would be that array dictionary would have a bit more memory usage than the double linked list as array dictionary is a set memory allocation while double linked list only uses the amount of memory it needs based on number of nodes. However, we believe the difference will not be that large as both structures will be holding about the same amount of data.

Plot:



Result reflection: As we have predicted the usage of both double link list and array dictionary are mostly equal, since they have roughly equal amount of fields in terms of data and similar runtime methods.

Extra Credit

1. We are implementing control flow manipulator, and put methods to handle if, repeat and while.
2. All parameters below are evaluated via AST node logic.

- For if, it would be if(cond, then, else), where if **cond** is true, **then** is executed and return, and if not, **else** is returned. There are 4 syntax for cond, where gr represents greater than, sm represents smaller than, eq represents equal to, nq represents not equal to.
- Example input: first: declare x, then if (sm(x,0), -x , x); if x is less than 0, it will return -x and return x otherwise --this one is logic for absolute value.
- For repeat, repeat(times, body), where **body** is executed **times** times, and return the final value
- Example input: first, declare x, then repeat(3, x+3), it will return x+9, if x is not declared.
- For while, while(cond, body, lim), where **body** is executed repeatedly until **cond** becomes false, and until the iteration does not go over **lim** (this might indicate the loop is infinite loop).
- Example input: first, declare x, while(sm(x,10), x+1, 100), it will return 10.

3. A description of anything else you want the grader to look at or be aware of:

For cond in if, like, we only make >,<, ==, != as comparing left and right for condition for if and while. (did not include greater than and equal to (>=), and smaller than and equal to (<=) for nicety.

For repeat, we use the most left variable as main variable that is modified from repeat method. For instance, for repeat(3, x+y), x is the one that is modified throughout the repeat.

For while, we mixed the assumption from both cond and repeat, such that we modify the variable like in repeat and using modified variable to check if condition is still met, and if not met anymore, break the while loop and return the value node.