Project 1 Group Write-Up

Part I

The place where we need to handle this situation is the `indexOf` method of `DoubleLinkedList` and `getKeyIndex` method of `ArrayDictionary`. For `indexOf` method, our strategy is the following:
Let `refer` be the returned value of calling `iterator.next()`. Therefore, `refer` is a reference to the `data` field in each node. Let `item` be the parameter of `indexOf` method. The logic of `indexOf` method requires us to determine if a particular input matches a particular `data` stored in the nodes. Depending on whether a pointer points to an object or is `null`, there are four cases:

1. `item == null` and `refer == null`
2. `item == null` and `refer` is object
3. `item` is object and `refer == null`
4. `item` is object and `refer` is object

For case 1, since both `item` and `refer` has the same value (`null`), they are considered a match.
For case 2 and 3, since `null` can never equal to an object, they are not considered a match.
For case 4, whether `refer` match item depends on the value of `refer.equals(item)`. Note that since `refer` is guaranteed to be an object, calling `equals()` on it will not cause an error.
Therefore, when `refer == null`, match is determined by whether `item` is `null`. If `item` is `null`, then we have a match; otherwise, if `item` is an object, we don't have a match. When `refer` is an object, match is obtained only when `refer.equals(item)` is true. Based on these analysis, match is obtained when `(refer == null && item == null) || (refer != null && refer.equals(item))` and the whole method is:

```
public int indexOf(T item) {
  Iterator<T> itr = this.iterator();
  int i = 0;
  while (itr.hasNext()) {
    T refer = itr.next();
    if ((refer == null && item == null) || (refer != null && refer.equals(item))) {
      return i;
    }
    i++;
```

```
    }
    return -1;
  }
```

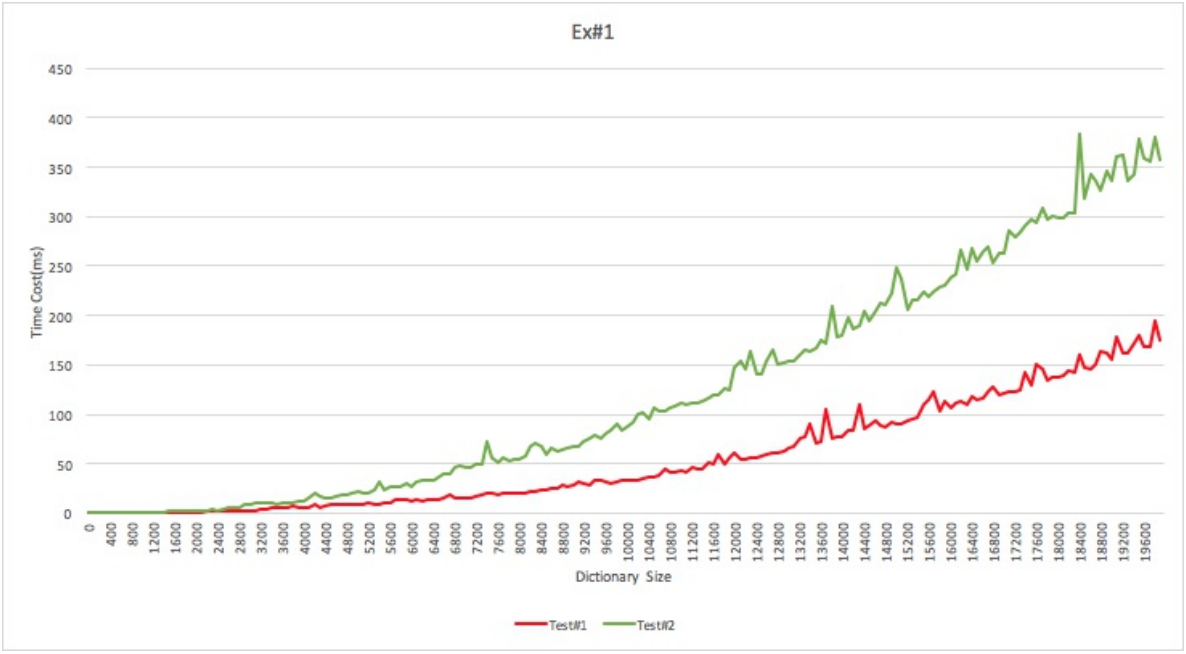
The strategy for `getKeyIndex` is the same: `pairs[i].key` is a reference to the data to match against the input key. By simply replacing the variables to A match is obtained when `(pairs[i].key == null && key == null) || (pairs[i].key != null && pairs[i].key.equals(key))` and the whole method is:

```
private int getKeyIndex(K key) {
  int repeat = this.size;
  for (int i = 0; i < repeat; i++) {
    if ((pairs[i].key == null && key == null) || (pairs[i].key != null
        && pairs[i].key.equals(key))) {
      return i;
    }
  }
  return -1;
}
```

Part II

Experiment 1:

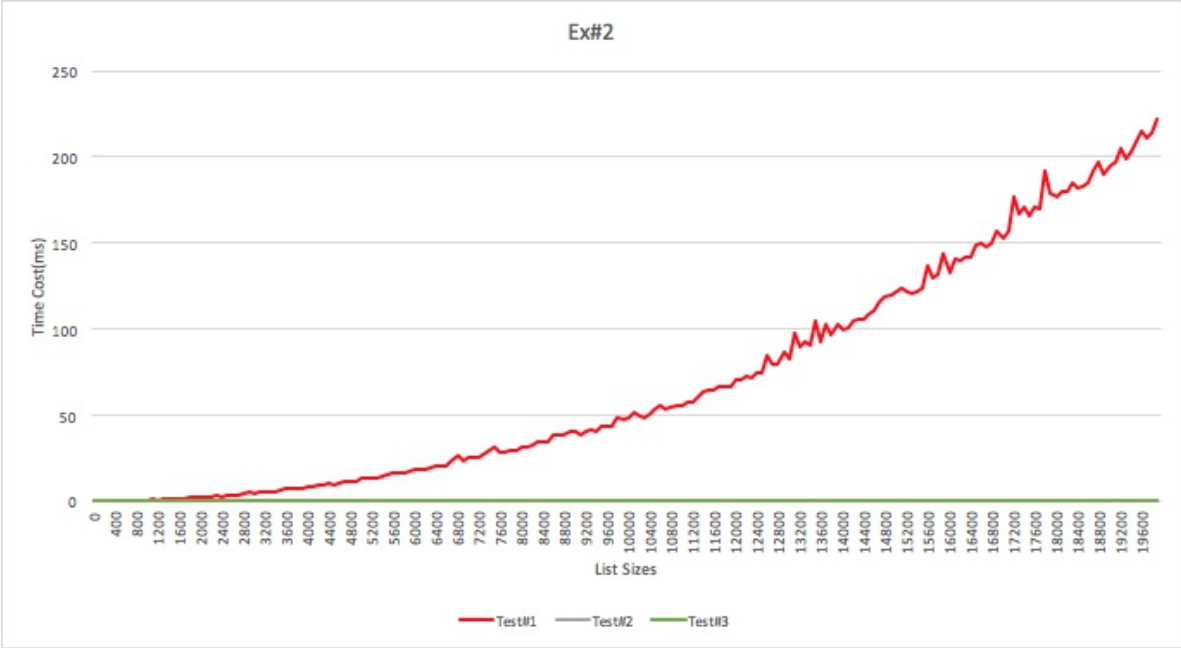
1. This experiment tests the time cost of `remove` method of `ArrayDictionary`. The tested dictionary will first be filled with elements with key values that are equal to their indices in the array. The first test traverse the array from the front of the dictionary to the end of it, and remove the each element repeatedly. Therefore, the elements of low indices will be removed first. The second test works backward: the elements with high index will be removed first.
2. We expect both test will give a quadratic curve, because both the outer for loop and `remove` method traverses the array in the dictionary.

3. 

4. The trend of the chart confirms our hypothesis. Both plot has a quadratic shape, which means they are of O(n^2). And the curve for Test#2 (removing in backward direction) has higher time cost than that for Test#1 (removing in forward direction). In order to remove a element in the dictionary, we first need to find the index of the element to be removed. To do that, `remove` has to traverse the whole array from index 0 to the end and compare each Key of each element against the input value until they match. Therefore, removing a key that is small (i.e., at the front of the array) takes less time than removing a key that is large (i.e., at the end of the array).Let n be the size of the array, each test traverses the whole array and at the same time call remove method, which also needs to traverse the whole array, therefore, we predict that the two tests are of O(n^2).

Experiment 2:

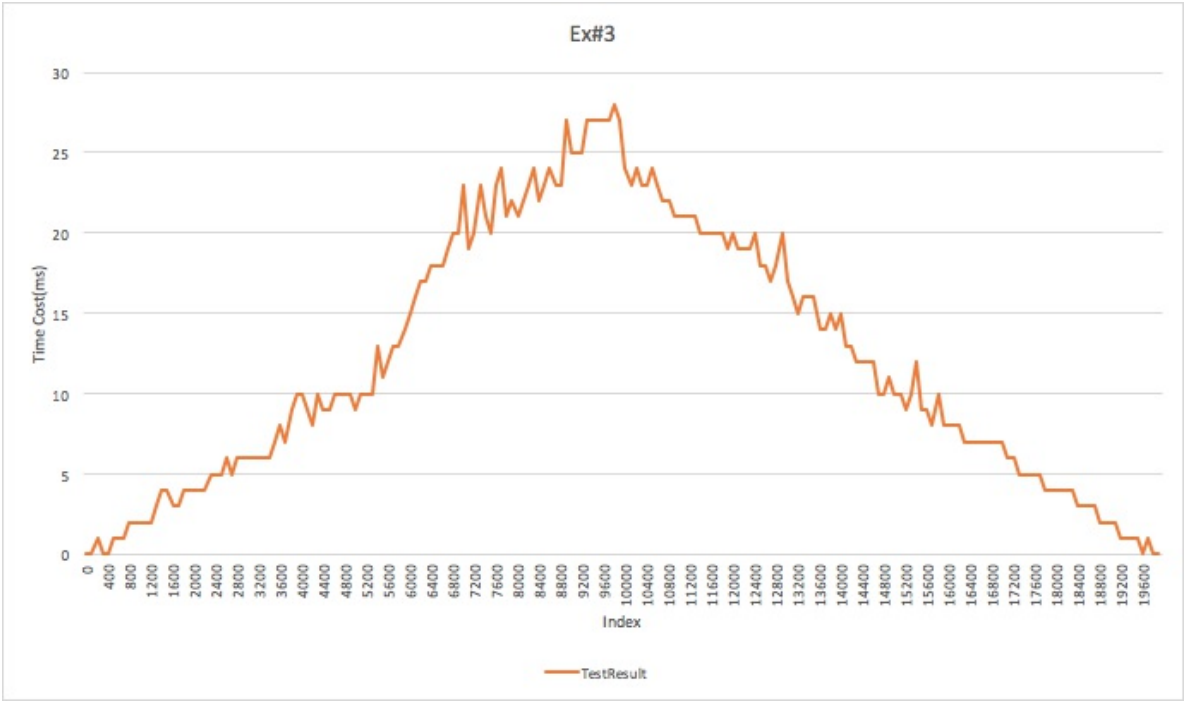
1. This experiment tests the time cost for traversing in a `DoubleLinkedList` and also get the data on each traversed element in three approaches. The first one is using for loop and `get` method, the second one is using a iterator, and the third one is using for each loop.
2. Let n be the size of the `DoubleLinkedList`. We expect the first way gives a quadratic curve (of O(n^2)), while the others give linear lines (of O(n)) that overlaps each other, because both for loop and `get` method traverses the list, so the total is O(n^2). But an iterator and for-each loop traverses the list only once, so they are in O(n).

3. 

4. The result we get confirms our guess for the first approach, but has some oddities for the second and third approach. Since both the outer for loop and `get` method needs to traverse the array, the curve for test1 is of O(n^2). However, according to the plot, the time cost of second and third approach is a constant very close to zero instead of O(n). We expect an O(n) cost because both the second and the third approach needs an iterator to traverse the list. One possible explanation is that the iterator works so fast that it almost seems to finish the traversing instantly.

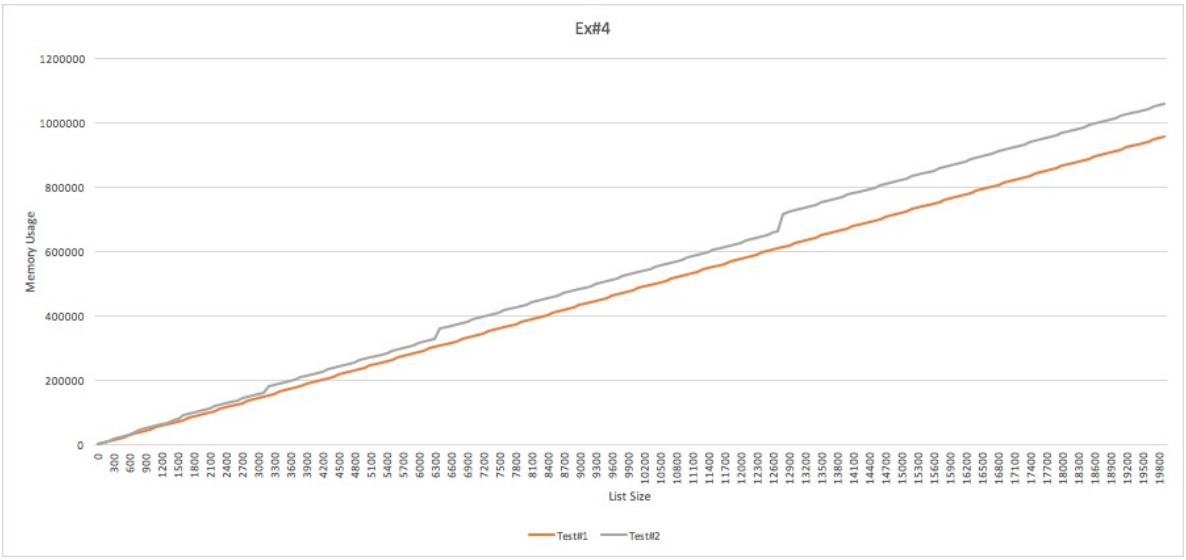
Experiment 3:

1. This experiment tests the relationship between the time cost of `get` method of `DoubleLinkedList` and the index parameter passed to it.
2. We expect the curve (time cost vs index) to increase first and then decrease linearly, where the maximum time cost happens at the middle of the index axis, because the way we implemented `get` is that if the index is greater than `size/2`, we search from the back of the linked list to the front and if index is smaller than `size/2`, we search from front to back of the list. In both case, the index at the middle of the list needs the largest amount of traversing. Since the time cost of get is proportional to the distance of the index parameter to the middle of the list, we expect the curve to increase and decrease linearly.

3. 

4. The result we get confirms our guess, but we also observed that both increasing part and decreasing part are linear. This is because we only traverse at most half of the list once.

Experiment 4:

1. This experiment tests memory cost of `DoubleLinkedList` and `ArrayDictionary` of different sizes.
2. We expect both of them will give a linear line (memory cost vs size), since their memory cost is linearly proportional to their size.
3. 

4. The trend of the chart basically confirms our hypothesis. However, we didn't expect that the plot for `ArrayDictionary` has "jumps" (short, high-slope regions) connecting regular long, low-slope regions. We believe the jumps are caused by resizing (double the capacity) when array in the `ArrayDictionary` is full. We confirmed

this by observing that each low-slope region is twice longer than the previous one, since we double the capacity when performing resizing.

Part III (Extra Credit)

We implemented a `derive()` function that takes a polynomial and returns its derivative. For example, executing `derive(3 * x ^ 2 + 2 * x + 1)` would return `3 * 2 * x ^ 1 + 2 + 0`. The result is not further simplified to `6 + 2`. Other examples are:

`derive(3)`

returns `0`

`derive(2 * x + 1)`

returns `2`

`derive(5 * x ^ 3 + x ^ 2 + 2 * x ^ 2 + 3 * x + 1)`

returns

`5 * 3 * x ^ 2 + 2 * x ^ 1 + 2 * 2 * x ^ 1 + 3 + 0`

We also implemented `if` function and `repeat` function. `if(cond, then, else)` statement evaluates `cond` and execute `then` or `else` based on `cond`'s value, as described in the spec. `repeat(amount, body)` evaluates `body` for `repeat` times and return result of the last execution.

Some examples are:

`x := 1`

`if (x, x-3, x-4)`

returns `-2`

`y := 0`

`if (y, x+1, x+2)`

returns `3`

`repeat(4, x-2)`

returns `-1`

`repeat(5, x-2)`

returns `-1`