Kristian Snyder  
Data Structures CS2021 Section 001  
Prof. Charles Zimmer, Prof. Karthik Vadambacheri  
Lab 05

Algorithm design was the main focus of this lab. We compared the performance and usage of three separate implementations of the same goal structure: a list that is sorted ascending across insertions and deletions. This is important because it displays both extremes of both design and improvement: small improvements that can make a large difference and a large improvement with diminishing returns.

The design of the reversed-search list was simple: the difference to the insertion and deletion functions is barely more than a few lines of code. However, it resulted in a dramatic improvement in performance during testing.

The use of open spaces in designing new insertion and deletion functions was different entirely; it resulted in about 1.25x the reduction in operations as the reversal did but required a full rewrite of the function and far more tooling and debugging to catch all the edge cases. In the end, the improvements were certainly worth the new design; compared to the original algorithm this implementation used *less than half the operations to perform the same task.*

This set of designs displays the concept of diminishing returns and over-optimization quite well, however. A change of an entire function only resulted in a doubling of the benefits compared to altering two lines of code. Further improvements of the third algorithm are certainly possible; however, the result would probably be only a reduction of another thousand operations instead of 4-5,000.

In practical work, a little optimization goes a long way. Past a small consideration, however, in-depth redesign should probably only take place on hot code paths that can result in real benefits from the work performed. Optimizing a function that is rarely used within an inch of its life is less beneficial than taking a few minutes to make a change to a function called thousands of times that can result in thousands of hours of saved CPU time.

## Task 3

Each version of the class seeks to perform the same task: maintain a list of pointers that are ordered ascending according to the values they point to. This needs to be consistent after both insertions and deletions from the list. Therefore, all three versions must have a way of ensuring new items are inserted in the correct place.

### Normal Ordered List - Task 1

This is the most basic implementation of an ordered list, with a back-to-front search on insertion and deletion; on each operation the list is iterated through from the first element to the last to either find the correct element to remove or the correct place to insert a new element. The contents of the array right of the insertion or deletion are then shifted right or left, respectively.

Searching from the first element has very predictable performance, as every element in the array is accessed on all operations. The only variability comes from the number of element moves, with the worst case being insertion or deletion at the beginning of the array. Since all elements are always accessed, inserting a list that is already mostly in ascending order has poor performance. However, inserting a reversed list has a very similar performance as and little variability.

If reliable and predictable performance is the goal, this list is a great choice.

### Search From End Ordered List - Task 2

Searching from the end on insertion and deletion operations has a much higher variability on number of accesses, although the average performance is the same. Since it starts at the end, finding the correct position on insertion *sooner* reduces the number of array accesses necessary to maintain state. However, it also means that having to insert an element at the very beginning of the list results in *double* the number of operations as a back-to-front search would need.

Since the performance of this algorithm is so dependent on the correct position being in the right half of the array, lists that are already mostly in order are a great choice to insert. However, for random insertion, there is a potential for extremely poor performance. On average the number of operations will be equal to Task 1, but since the variability is far higher this makes it a less safe approach. Finally, for reversed lists Task 2's algorithm should be avoided completely for insertion as it will result in double the operations as in Task 1.

### Empty Spaces - Task 3

Building on Task 1's predictable search algorithm, Task 3 seeks to reduce the number of operations to perform by not shifting elements on deletion. The location is set to null to avoid confusion, but no elements are shifted to fill the space occupied. On future insertions, if the correct location for the item is in the blank spot, once again no shifting must take place.

This is a clear improvement over Task 1 in terms of operations; in the worst case the number of accesses is the same as the initial implementation. For most deletions, the search has to only look through a portion of the array and clear the correct location. Insertions are not consistently better, as there must be an empty location to avoid shifting off the end. However, if one exists, a small portion of the array can be shifted to make room for the new element.

## Ranking of Algorithms

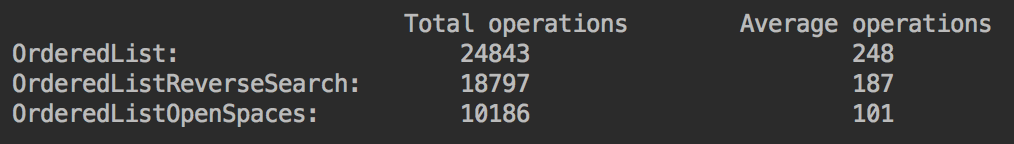
I predict the following ranking in performance for random insertion and deletion:

1. Back-to-front
2. Empty space front-to-back
3. Front-to-back

However, the number of operations separating 2 and 3 will most likely be far fewer than the number separating 1 and 2.

## Task 4

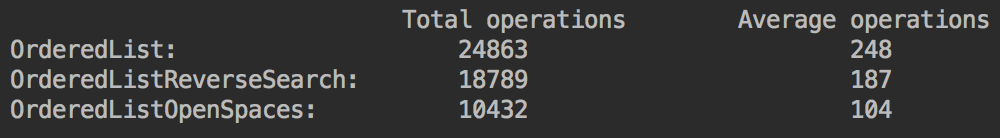
### 20-length arrays



Results of test length 20

The open-spaces algorithm is, by far, outperforming the other two algorithms enormously. While I was correct about the front-to-back algorithm performing the worst, the open-spaces algorithm is multiples faster.

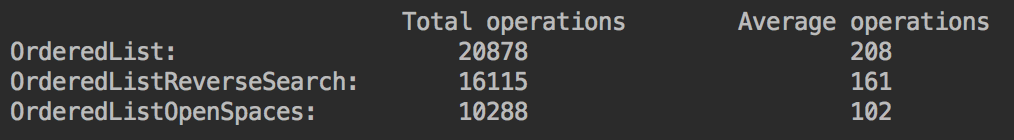
### 50-length arrays



Results of test length 50

Not much of a change here, although both the front-to-back and back-to-front arrays had increased numbers of operations. I imagine this is due to exceptions; if the array is full it doesn't perform any operations. This cannot happen with an array of length 50, so it results in a small increase.

### 10-length arrays



Results of test length 10

The exception theory from the previous test appears to be correct. For an array of length 10, there will be *far* more full arrays than before, resulting in a far smaller amount of operations for both of the first two algorithms.

#### Program description

For all of the tests, the open-spaces algorithm performed approximately the same number of operations, with the two other algorithms requiring anywhere from 21,000-25,000 or 16,000-18,500 operations in total. This places the open-spaces algorithm, without argument, as the most efficient in terms of checks and moves. By leaving null locations on removal, the algorithm can place new elements in those empty locations and also rarely has to shift many elements if it does, since a null location provides an easy spot to shift into. Since the number of operations was relatively consistent across many array lengths, it suggests that the implementation performs well at scale.

The program simply runs in two loops: an outer array of 100 runs and an inner array of 55 operations comprising all insertions and deletions. If either operation reaches its maximum number, the rest of that run is entirely the other operation.

To make deletions successful as much as possible, a separate ordered list is used to hold all additions. When a deletion is called and there are values that can be removed, one is randomly selected and popped off that list.

At the end of each run, the operations for each are added to operation counters of the respective lists. The lists are then emptied for the second run, including the list tracking the additions.

At the end of the program, the total operations performed for each implementation, as well as the average operations necessary for each run are displayed.

#### Results analysis

I believe that analyzing the averages is sufficient to compare the performance of each algorithm. They all perform within well-defined bounds that are only multiples of each other; this is a marked difference from many others that have a best case of O(N) and a worst case of O(N^2). Measuring the extremes of each would be helpful only if we wished to assume the worst case for each, which may result in a selection that performs worse in the real world than the others.

One approach that would be worth exploring is semi-sorted lists; in many cases array accesses and inserts are not random in the real world and could result in a surprising result depending on the type of list. Additionally, since all of these lists sort ascending, descending lists are horrible for use with them and are a good example of needing to check the use case before deciding.