Timing Tests

Unordered Array

Insertion

For the Unordered Array, the insertion complexity is **O(1)**. This is because all it has to do for an insertion is add the object onto the end, and increase the according counters. The timing for this shows that it is very efficient in terms of inserting

Add Remove cycles

The cycle for adding and removing is highly ineffecient, having a complexity of **O(n)**. This is because for every remove, the array must go through the whole array comparing every element to find it's removal element.

Remove

The removal process, because of it's unsorted array, ranks at a complexity of **O(n)**, which is inefficient. The remove must go through thw whole array looking for the element.

Pros and cons

Pros

- O(1) insertion

Cons

- Fixed size without extra overhead to add in size adjustment
- O(n) removal and search
- Iteration will not go in order

Ordered Array

Insertion

The ordered array makes use of binary search to find the insertion point. However, it still retains a **O(n)** complexity due to the need to shift all the elements after finding it's insertion point

Add Remove cycles

Although it's removing is quick, it's counteracted by it's insertion, which bumps it back to **O(n)** complexity.

Remove

Removing is where the ordered array data structure shines. Because of the way that the array is structured, it only needs to pop the top of the array off, giving it a **O(1)** complexity.

Pros and cons

Pros

- O(1) removal and peeking due to the next element to be removed always at the top of the array.
- Ordered array allows you to easily search, retaining a O(log(n)) searching
- Iteration will go in order of priority

Cons

- To insert, you have to search AND shift the array.
- Fixed size without extra overhead to accomodate for this

Unordered Linked List

Insertion

Insertion is set number of operations that will never change based on size, giving it an efficient **O(1)** complexity.

Add Remove cycles

This data structure suffers from the fact that it must iterate to find the object to remove, giving it a **O(n)** complexity.

Remove

This data structure must iterate to find the object to remove, giving it a **O(n)** complexity.

Pros and Cons

Pros

- Quick insertion
- Variable size. No overhead to grow the list

Cons

- Slow iteration based removal
- Memory contents are linked rather than consecutive, resulting in a slight downside to iteration based operation (which is used a lot in this case)

Ordered Linked List

Insertion

Insertion is done by iterating through the list. Compared to Ordered Array's insertion, it is slower to find the insertion point because it cannot use a binary search, but it also does not need to shift the contents to make room for the new object. However, it is still **O(n)** complex.

Add Remove cycles

It's quick removal is counteracted by it's slow insertion, giving it **O(n)** complexity.

Remove

Removing is quick because the array is ordered and the next element to remove is always on the top. It has a set number of operations independent on the size, giving it **O(1)** complexity.

Pros and Cons

Pros

- It's removal is quick, which also makes it's peeking quick
- Variable size. No overhead to grow the list

Cons

- It has to iterate through the list to find the insertion point
- Memory contents are linked rather than consecutive

Reccomendations

Choosing which data structure to implement depends on how much each function will see used, as well as how many entries are planned. For uses that are expecting a variable amount of entries into the queue, a linked list implementation is perferred for it's variable size property, allowing the program to scale to the user's need. As for which linked list, it would depend on the needed functions. For example, a car dealership may want to throw entries into the queue as fast as possible to avoid taking up time of customers, and just have the computer process the next job to be finished while the worker works on the current job. The unordered linked list will be the the best option of the four. A hospital that needs to pull up the next job quickly may perfer the faster peeking and removing of the Ordered Linked List.

As for queues where the max size is known, Arrays are better because of their consecutive memory space, allowing for binary search in the ordered implementation. A teacher that grades their students, in which he knows the class size, may perfer the ordered array implementation. Once he's graded all the papers, the binary search will bring up their profiles quickly. Should he have to remove students from the class because of "budget cuts," he can simply pop the student with the lowest grade (which we can dub as the "highest priority" in this case)off his student list. As for the unordered array, it is a the best of these four structures for cases which the max entry size is known, but insertion time is essential. I can't really come up for an example for this, so maybe it's pretty useless.