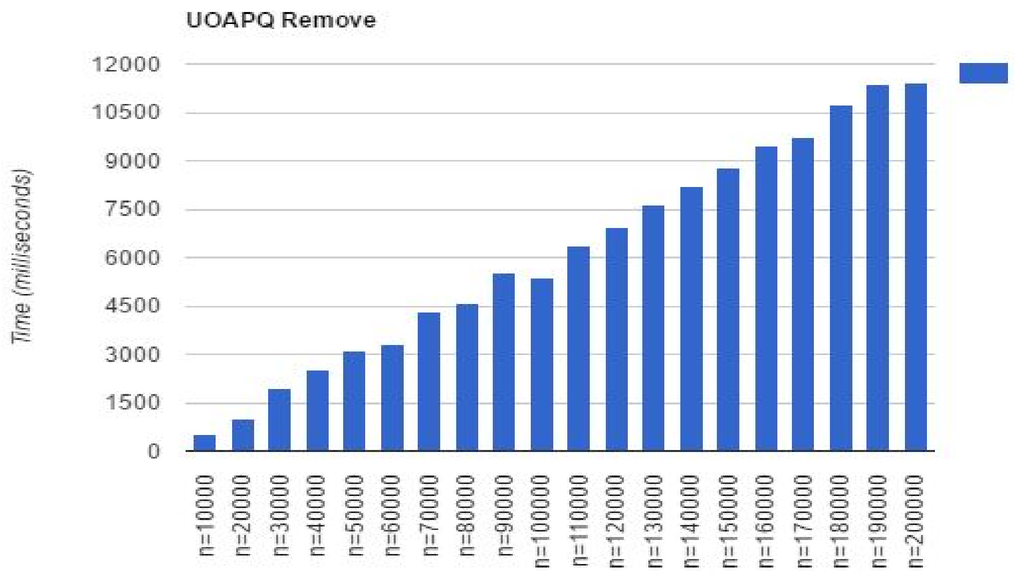
Luke Stodgel
Cssc0956
Mr. Riggins
3/15/17

Project 2 Analysis

UOAPQ



Time (y axis) - 531, 1024, 1936, 2534, 3130, 3330, 4335, 4565, 5548, 5367, 6356, 6960, 7641, 8206, 8799, 9460, 9758, 10740, 11389, 11400.

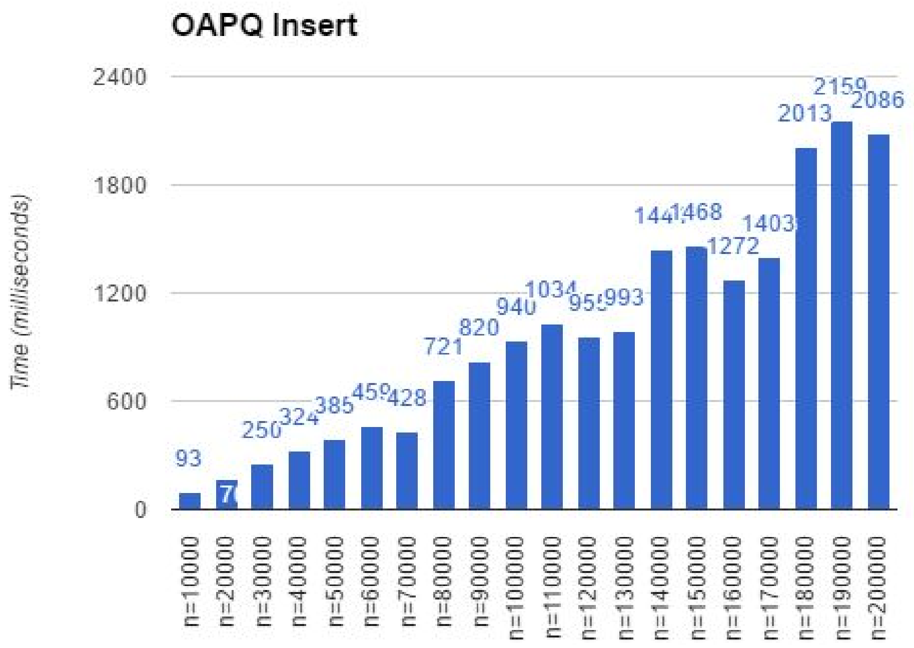
Insert

First I will talk about the complexity of the unordered array priority queue's insert method. For an unordered array priority queue it is known that the objects in the list are not kept in a specific order which means that we insert at the end of the list every time in O(1) complexity. In the graph above we tested the complexity of the remove method during runtime for the UOAPQ. It is evident that my insert runs in O(1) time because in order to test the remove, we first had to insert items and then remove them; my graph shows that happening by its linear shape. In other words, O(1) + O(n) = O(n).

Remove

Next I will talk about the complexity of the unordered array priority queue's remove method. This method is very crucial in terms of having a properly functioning UOAPQ. Although we insert at the end of the unordered list in O(1), removing is O(n) because we have to remove the object with the highest priority. To remove the object in the list with the highest priority we must find the object with the highest priority, compare that object to every other object in the list, once we've found the object of the highest priority we set temp to point there and have a pointer set to the object immediately before the object to remove, we then perform a shiftLeft on each index in the array starting from the object immediately before the one being removed and goes until there are no more objects to be shifted thus removing an object. Because we had two for loops in the method, one for searching for the item that was to be removed and also for the shiftLeft, it was O(n) + O(n) which = O(n). It is clear that my remove worked in O(n) because you can see from the graph that the time test gave me more or less linear results.

OAPQ



Time(y-axis) - 93, 170, 250, 324, 385, 459, 428, 721, 820, 940, 1034, 955, 993, 1444, 1468, 1272, 1403, 2013, 2159, 2086

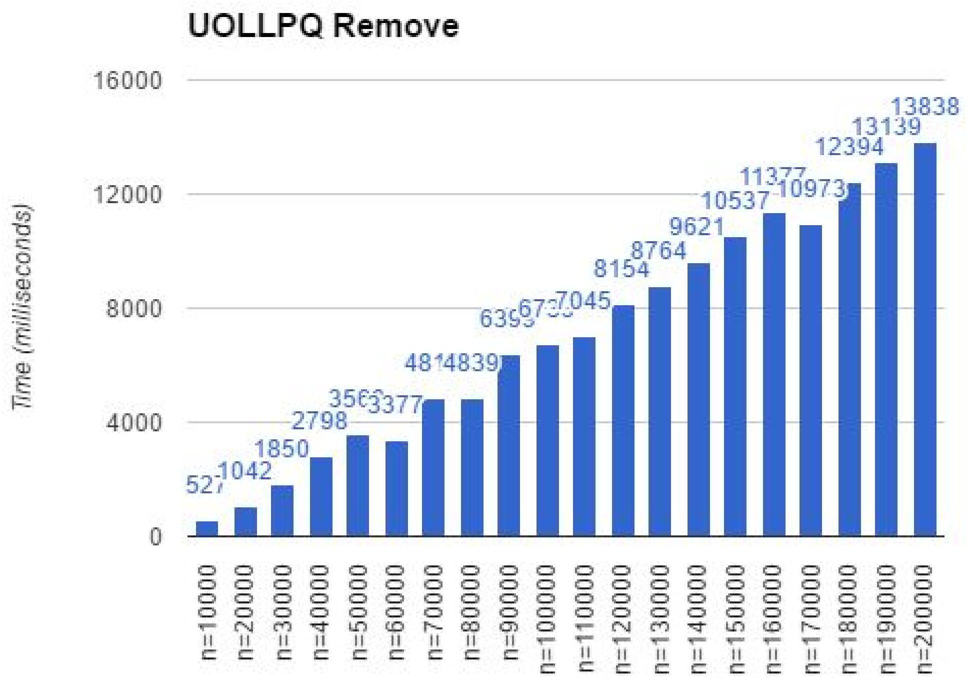
Insert

Next I will talk about the complexity of the ordered array priority queue's insert method. This method works in O(n) time because we have to insert an item into this list in correct order based on that object's priority. Instead of just inserting onto the end of the list in O(1), we have to search through the list using binSearch and find where we are going to insert an object (O(logn)). Next we have to shift every object in the list over to make room for something to be inserted (O(n)), and then we drop in the object (O(logn) + O(n) = O(n)). Arrays are linear so we have to shift the indices for an insert. The results on the graph for the insert for this OAPQ show that this worked in O(n), and when compared with the graphed results for the insert method in the ordered linked list priority queue, the hiccups where caching occurred on the graph show similar behavior.

<u>Remove</u>

Next I will talk about the complexity of the ordered array queue's remove method. For the OAPQ it is known that we simply remove from the end of the list in O(1) time. We do this because we assume that all of the objects in the list were already inserted in order based on their priority. If the items are already in order, we just follow the known, first in, first out, functionality of a queue and just remove the last item in the list because it was inserted first. So, removing from an ordered list is O(1) complexity and inserting is O(n).

<u>UOLLPQ</u>



<u>Time (y-axis)</u> - 527, 1042, 1850, 2798, 3560, 3377, 4815, 6399, 6733, 7045, 8154, 8764, 9621, 10537, 11377, 10973, 12394, 13139, 13838.

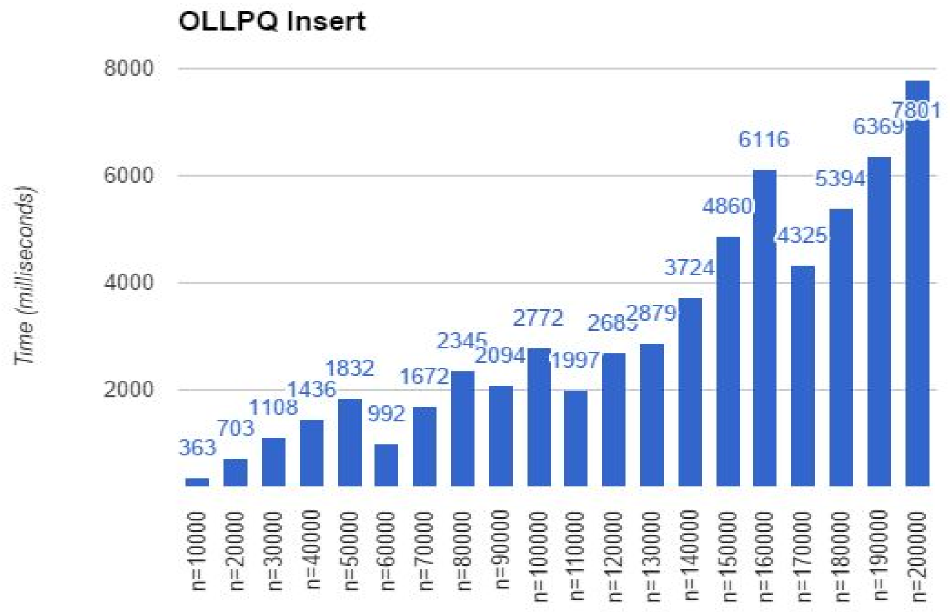
Insert

Next I will talk about the complexity of the unordered linked list priority queue's insert method. Like the insert for the unordered array priority queue, we are always inserting at the end of the UOLLPQ in O(1) time. Based on the results from the graph of the remove method for this class, it is evident that my insert runs in O(1) time because in order to test the remove, we first had to insert items and then remove them; my graph shows that happening by its linear shape. In other words, O(1) + O(n) = O(n).

Remove

Next I will talk about the complexity of the unordered linked list priority queue's remove method. Again, for this class to have the behavior of a priority queue we have to remove objects from this list in correct priority order. This takes O(n) complexity because you must search through the list to find the object with the highest priority, save a pointer to that object and also one to the object right before it, and then you change pointers to delete the highest priority object and return it. As you can see from the linear shape of my graph, the time results indicate that my remove indeed worked in O(n) time.

OLLPQ



Time (y-axis) - 363, 703, 1108, 1436, 1832, 992, 1672, 2345, 2094, 2772, 1997, 2689, 2879, 3724, 4860, 6116, 4325, 5394, 6369, 7801

Insert

Next I will talk about the complexity of the ordered linked list priority queue's insert method. This method is crucial for the correct functionality of an ordered list priority queue. Because this class is an ordered priority queue, we have to insert objects into this list while following the insertion order based on priority. We have this new object that we will insert, but we have to find where it goes, so we compare the data in that new object with the objects in the list until we find where it needs to go (O(n)), then we change pointers to add it into the list. As you can see from the graphed results, the insert is indeed O(n), linear, with some minor caching hiccups.

Remove

Next I will talk about the complexity of the ordered linked list priority queue's remove method. For this remove method, we simply remove the object at the end of the list in O(1). Because the objects in the list were assumed to have been inserted into the list in order, we just remove from the end to satisfy the FiFo behavior of a queue.

SUMMARY

It became more evident to me that arrays perform better during runtime than linked lists. I found that this is because arrays have better caching locality than linked lists. This means that arrays allocate the memory it needs during runtime in a more contiguous manner than linked lists do, thus having better performance during runtime.

Next I will discuss the advantages and disadvantages for arrays and linked lists when compared with the other. A disadvantage for arrays when compared with linked lists is that when inserting items, more space/memory needs to be allocated for the object being added, and also all of the objects after the insertion point need more space/memory to be shifted, but with a linked list we just change pointers and allocate memory onto a random location on the heap. Also while deleting items with arrays, every item in the list needs to be shifted over to cover up the hole in the linear list made by the removal, while with linked lists we just change the pointers and some memory is garbage collected. To summarize, linked lists insert and delete faster than arrays, but arrays have faster access to its contents.

An advantage for linked lists when compared to arrays is that linked lists have dynamic size and arrays don't. A disadvantage for linked lists is that while inserting, extra memory is needed for every object in the list.

So if I were to choose the best implementation for this priority queue project, even though the array implementation is a bit quicker performance wise, I would still use a linked list implementation because it isn’t restricted by size.