Priority Queue Final Project

By: Rami Mouro

03 MAY 2018

1 Linked List:

The linked list implementations can be done in several routes. We can either keep track of previous node so that way we can go back if the priorities are equal however in my implementation of the linked list priority queue I decided to first check if the node to be inserted is smaller or equal to the head of the list if it is smaller than the head then it becomes the new head and then if it is equal to check the secondary priority to determine where it needs to be inserted after the head or it needs to become the head.

Then we check we then move into a while loop that keeps moving down the linked list until we find the node before the node that we need to insert after we then check the priority to see if they are equal and then further narrow down our decision on where we need to insert the node.

Then to print the Priority Queue we can easily print the head of the linked list and then set a temporary node equal to head and then set the head equal to the next node in the list then delete the temporary node.

This implementation is very efficient if the number of patients is low because it runs through all the patients one by one and checks the priorities of each individual one until it finds the node that is needed.

The standard deviation for this implementation was high and was higher the more patients we had. This is important for us to understand that there was a few runs that were very quick and that the mean might be skewed a little.

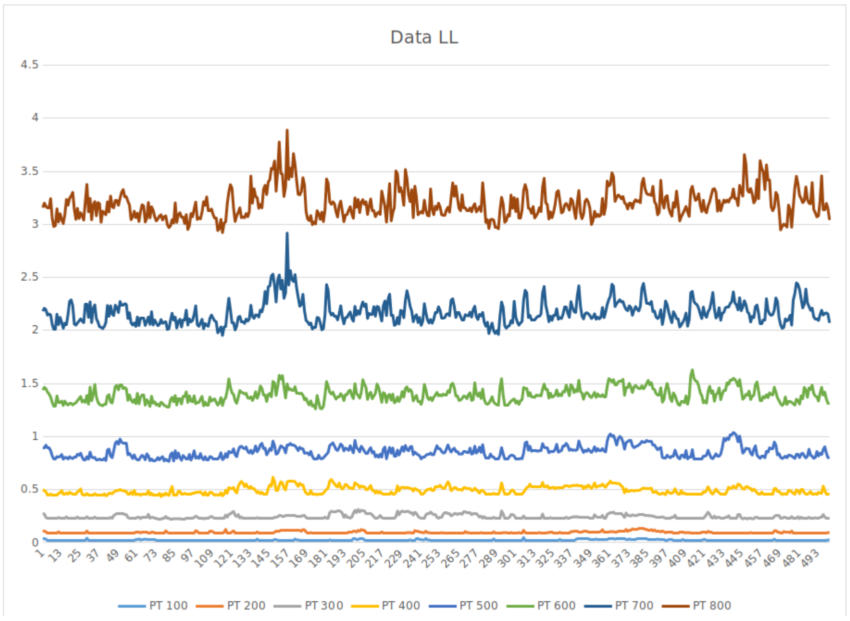


Figure 1: Linked List 500 Runs Graph

2 Standard Template Library:

For the Standard Template Library implementation I created a struct CompareNode that had a public standard binary function that compared two nodes and had a third argument which was the result stored in a boolean the boolean operator() which we have to define compares the two arguments given and returns a result.

Then to print the Priority Queue we set a temp node to the top most node and we print that node we then pop that node from the Priority Queue then delete the node that was the top node after we are done with it and now the next Node with the highest priority is up and we can print it or pop it without printing it.

The Standard Template Library still compares every node that are stored in vectors which means it still has to go through every single node to compare the one inserted which means it is still very efficient however the more patients the more comparisons there are and we are still comparing every patient. My results showed that the STL was less efficient than the linked list when the sample was lower than 800 patients however when it became over 800 patients it was faster.

The standard deviation for the STL implementation was always higher which means that there was a lot of variation in the run time between the different results. The standard deviation was also higher the more patients we had.

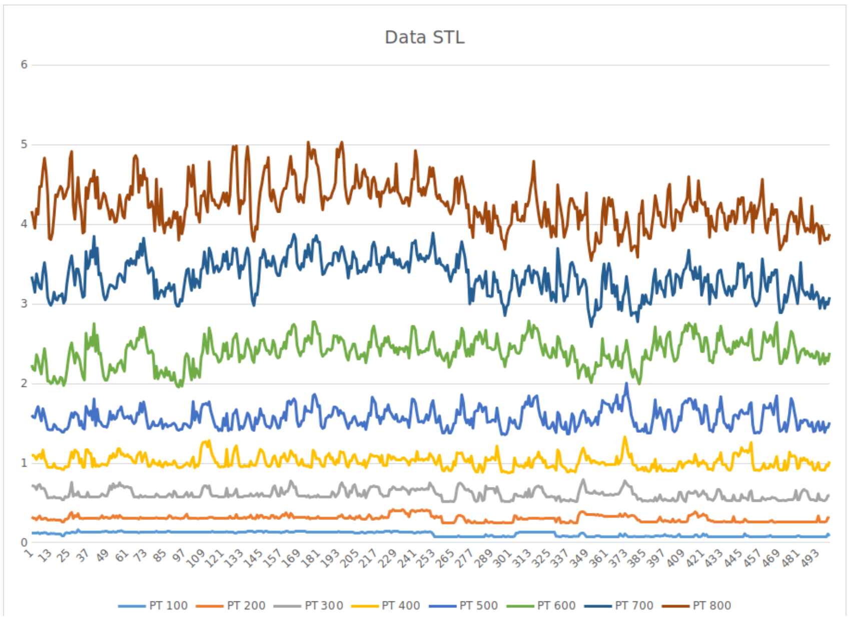


Figure 2: Linked List 500 Run time Graph

3 Binary Heap:

A min binary heap is a tree where the value of each node is greater than or equal to the value of it's parent with the minimum-value at the root. This makes it so that we have an order tree that is much easier to search through than having to search through every single node. Every time we cut the number of comparisons essentially in half.

When we compare every node until we find the correct node we run the risk of that node being at the end of the list and having to compare every single node however in a binary heap Priority Queue we cut the list in to a smaller list(branches of a tree) with every comparison the list becomes smaller eliminating multiple patients at a time. The Binary Heap worst case run time scenario is much faster than the worst case scenario of going through a list and checking each one and is related to the height of the tree.

However in the binary heap implementation when we de-queue it is not as easy as deleting the top most node because we have to replace that node with the new absolute minimum-value which means we have to check the primary priority and possibly check for the secondary priority as well.

The binary heap was very efficient in the runs and was almost 3x faster than the Linked List implementation and the Standard Template Library implementation and maintained it's standard deviation while it was also slower the more patients we added it was much faster than the other two implementations.

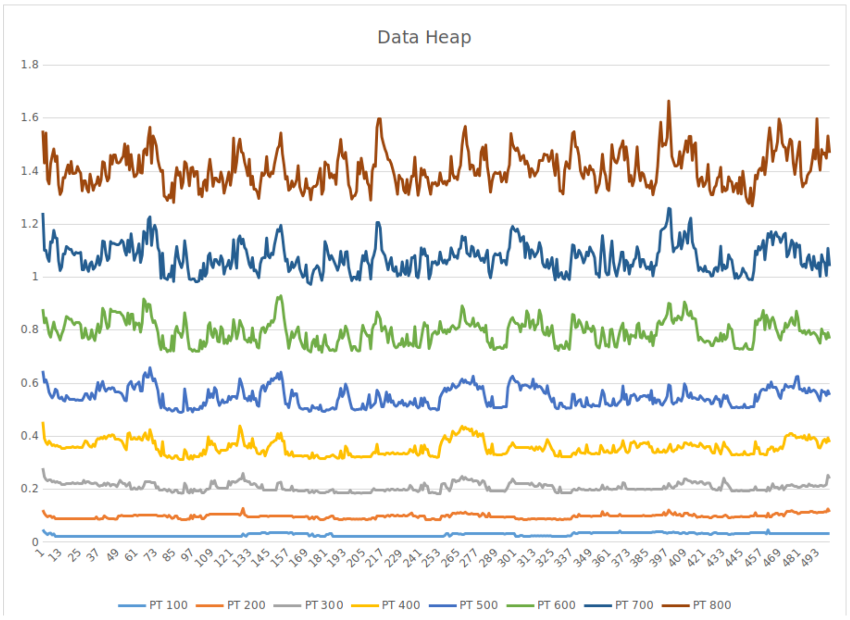
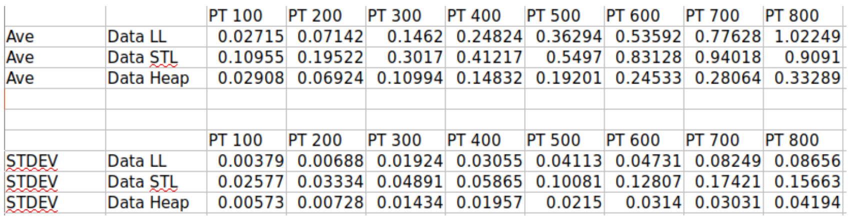


Figure 3: Linked List 500 Run time Graph

4 Results:

The Results were really intriguing the binary heap was the fastest implementation over all. When the patients ware under 500 the linked list and binary heap were really close however over 500 the linked list times started increasing very quickly however the binary heap was still very low and the Standard Template Library was consistently slower until the patients were above 800 it was slightly faster than the Linked List.

The STL also always had a very high Standard Deviation followed by the Linked list and then binary heap which had a significantly lower standard deviation.



		PT 100	PT 200	PT 300	PT 400	PT 500	PT 600	PT 700	PT 800
Ave	Data LL	0.02715	0.07142	0.1462	0.24824	0.36294	0.53592	0.77628	1.02249
Ave	Data STL	0.10955	0.19522	0.3017	0.41217	0.5497	0.83128	0.94018	0.9091
Ave	Data Heap	0.02908	0.06924	0.10994	0.14832	0.19201	0.24533	0.28064	0.33289
		PT 100	PT 200	PT 300	PT 400	PT 500	PT 600	PT 700	PT 800
STDEV	Data LL	0.00379	0.00688	0.01924	0.03055	0.04113	0.04731	0.08249	0.08656
STDEV	Data STL	0.02577	0.03334	0.04891	0.05865	0.10081	0.12807	0.17421	0.15663
STDEV	Data Heap	0.00573	0.00728	0.01434	0.01957	0.0215	0.0314	0.03031	0.04194

Figure 4: Table containing results

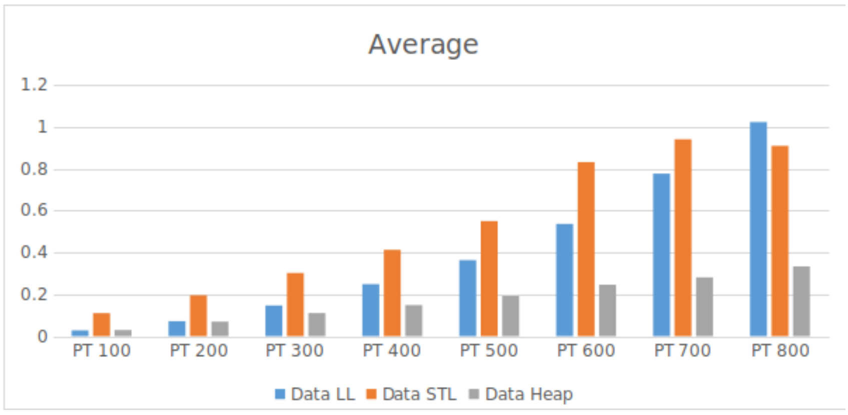


Figure 5: Averages

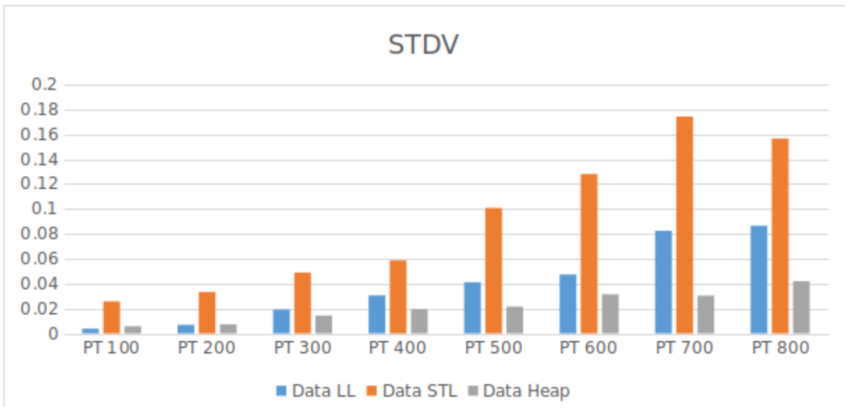


Figure 6: Standard Deviation