# CSCE 221 Cover Page

## Programming Assignment #3

## Due Date: Wednesday October 30, 11:59pm

## Submit this cover page along with your report

First Name: Bryant Last Name: Passage UIN: 326009948

**Any assignment turned in without a fully completed coverpage will receive ZERO POINTS.**

Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

| CSCE 221 Students | Other People | Printed Material | Web Material (URL) | Other |
|---|---|---|---|---|
| 1. Josh | 1. | 1. | 1. stackoverflow.com | 1. |
| 2. Elaine | 2. | 2. | 2. geekforgeek.com | 2. |
| 3. | 3. | 3. | 3. | 3. |
| 4. | 4. | 4. | 4. | 4. |
| 5. | 5. | 5. | 5. | 5. |

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today's Date: 10/30/19

Printed Name (in lieu of a signature): Bryant Passage

# Priority Queue Data Structure Implementation

Bryant Passage

**Introduction:**

In this project, three priority queue data structures were constructed with the goal of calculating the time complexities of each one. A queue is an abstract data type which has the concept of "first in first out" when dealing with elements entering and leaving the queue. A priority queue is a queue that usually contains a key and a value, where the key is the determining factor of which element in the queue gets removed first. To make the project simpler, the queues only contain the key of each element, and for each queue, the minimum key of the queue will have the highest priority. Therefore, the first element to be removed will be the element with the lowest value key. As the queue gets larger and larger, memory allocation will be a problem. However, for this project, dynamic memory allocation is used for all data types. The three queue data structures used for this lab include an unsorted queue and sorted queue implemented using a linked list as well as a heap queue implemented using C++ vectors of the C++17 standard. In this project, we will explore the three queue priority data structures as well as find the time complexity of each data structure.

**Theoretical Analysis:**

An insert operation for each of the three priority queue data structures take different Big-Oh complexities. The underlying time complexity of one insert operation depends on the way each priority queue is implemented.

For the unsorted priority queue, to insert an element into the queue, all the queue needs to do is add another node at the end of the linked list. Since the number of elements in the list do not affect the length of the insert operation because of the fact that the queue appends the new element at the end of the list, the time complexity of one insert is $O(1)$. That means that to insert $n$ elements into the unsorted priority queue, we multiply $n$ with $O(1)$ which gives us a time complexity of $O(n)$. This also means that the amortized complexity to insert $n$ elements is $O(n)/n$ which is $O(1)$. To insert and remove the elements in a priority queue is to actually sort the elements based on their priorities. This is a useful operation as this would allow a queue that would be sorted by the highest priority to the lowest priority. For an unsorted priority queue implemented by a linked list, we found that to insert an element into the queue takes $O(1)$ time. However, to remove the highest priority (which in this case is the smallest value key) takes longer when more elements are present in the list. This is because the algorithm to find the smallest key value in the unsorted list requires the queue to traverse through all elements in the list and find minimum value by comparing all the elements to each other linearly. That means that for an unsorted queue with $n$ elements, the queue would need to search through all $n$ elements to find the smallest key element and then remove it from the list (which with a linked list we know that to remove an element takes $O(1)$ time). This means the total time it takes to remove the minimum element from an unsorted queue takes $O(n + 1)$ which simplifies to $O(n)$. The total time it takes to remove all $n$ elements from the list would also take $n*O(n)$ which is $O(n^2)$. If we combine both operations together and we needed to sort $n$ elements (insert $n$ elements and then remove all $n$ elements), the operation's Big-Oh complexity would take $O(n + n^2)$ which simplifies to $O(n^2)$. The amortized cost of inserting and removing $n$ elements would be $O(n^2)/n$ which is $O(n)$. Therefore, the average time to insert and remove one element is $O(n)$.

For a sorted priority queue whose implementation is also a linked list, the insert operation is more complicated than the unsorted priority queue's insert. To insert an element into the sorted priority queue, the queue must find the position of the new element in the queue before inserting it. This means that if there are $n$ elements in the queue and we inserted one element into the queue, the queue would need to traverse through the list and find the position where the previous element is smaller than the current element, but less than the next element. This means that the number of

elements the linked list needs to traverse through is between 1 and $n$.[1] The best case for this scenario is if the element being inserted is the smallest of all elements in the queue and can be placed at the beginning/front of the queue. This operation would take $O(1)$ time. The worse case would be if the element is bigger than all elements/keys in the queue which would make the queue traverse through all $n$ elements in the list before inserting the element at the end of the queue. This would make the Big-Oh complexity of an insert $O(n)$. For this project, we will focus on the worst case. This means that to insert $n$ new elements into a list, it would take $O(n^2)$ time with an amortized/average time of $O(n)$ to insert an element into the sorted queue. Removing an element is simpler for this sorted queue. Since the queue is already sorted, the minimum key/element with the highest priority is already at the beginning of the queue which means that the time to dequeue is $O(n)$ for $n$ elements with an average of $O(1)$ per remove. Combining the two operations by inserting $n$ elements and removing those $n$ elements gives us a Big-Oh complexity of $O(n^2 + n)$ which is just $O(n^2)$ with an amortized cost of $O(n^2)/n$ or $O(n)$.

The heap priority queue combines methods of a binary-tree-like sort with an array/vector. For this data structure implementation, the elements are put inside a C++ vector, however, the elements are sorted as parent-child relations in a binary tree. This means that the $0^{th}$ element of the array represents the root of a binary tree and the left and right children are represented as $2i+1$ and $2i+2$ where $i$ represents the height of the parent node in the binary tree. The elements in the priority queue are ordered so that each child only needs to have a greater key than their parents. This eliminates the need for each element to be compared to every element in the list and rather only need to be ordered specifically by their parent. As the queue gets larger through the insertion of elements, the elements are inserted at the end of the vector which corresponds to the bottom right element in the binary-tree representation of the queue. However, this insertion does not guarantee that the heap is still sorted. Therefore, the queue must sort the newly inserted element through a process called an upheap. This method checks the newly inserted element with its parent. If the new element is greater than its parent, then the order of the binary-tree is correct, and no movement of the new element is needed. However, if the newly inserted element is smaller (or greater in priority) than its parent, then the two nodes would need to switch so that the child becomes the parent and the parent becomes the child. The process repeats with the new parent (previous child) and its parent until the heap becomes fully sorted. The best case for this scenario is if the new element is already a greater key value (lesser priority) than its parent. This would make the best case time complexity for an insert to be $O(1)$. However, the worst case for this scenario would be if the new element is of the highest priority. This would mean that the new element would need to upheap until it reaches the root position. The number of upheaps required before the bottom element reaches the root would be the height of the binary-tree. The height of the binary tree is $i$ and the height in relation to the number of elements/nodes in a binary tree is $i = log_2(n)$. This means that the worst-case time complexity for an insert into a heap priority queue is $O(logn)$. To insert $n$ elements into a heap priority queue would be $n * O(logn)$ or $O(nlogn)$. Removing an element from the heap priority queue is simple at first. Since the highest priority element is the root of the binary-tree, then the queue will remove the root and replace it with the last element in the binary-tree (which is the last element in the vector as well). However, if the new root is greater in key value (lesser in priority) than its children, then the queue will need to perform a series of downheaps (basically upheaps but traversing downwards). The worst-case for this scenario is if the new root needs to traverse all the way to the bottom of the binary-tree. Luckily, from previous calculations, it was found that the number of downheaps/swaps required to bring a root to the bottom of the tree is equivalent to the height of the tree which is $i$. The height of the tree in relation with the number of elements is, again, $i = log_2(n)$.

[1] Note that $n$ is the number of elements CURRENTLY in the linked list and will increase every time a new element is stored.

Therefore, the worst-case time complexity to remove the highest priority element is *O(logn)* and to remove all *n* elements would be *O(nlogn)*. Combining the insert and remove functions for *n* elements, if we were to insert *n* objects and then remove all *n* objects, it would take *O(nlogn + nlogn)* which will simplify to *O(nlogn)*. To calculate the amortized cost of inserting and removing one element, it would be *O(nlogn)/n* or *O(logn)*.

**Experimental Setup:**

A. Machine Specifications:
   - Custom built PC
     - Ryzen 7 3700X @ 3.6GHz Base → 4.4Ghz Boost[2]
     - ASUS X470-F Gaming Motherboard
     - 16 GB Trident Z RGB 3200MHz DDR4 RAM
     - Nvidia GeForce GTX 1070 Founders Edition

B. Test Setup:
   - Each priority queue was tested with 100,000 inserts for trial 1 and then 1,000,000 inserts for trial 2 to ensure the boost functions of the CPU did not fluctuate the overall push operations.
   - Each priority queue was tested with 100,000 inserts and removes.
   - Each queue was set to insert random numbers into the queue to avoid having a pre-sorted queue before removing the elements.

C. Unsorted Priority Queue using Linked List
   - Trial 1:
     - 100,000 inserts
   - Trial 2:
     - 1,000,000 inserts
   - Trial 3:
     - 100,000 inserts followed by 100,000 removes

D. Sorted Priority Queue
   - Trial 1:
     - 100,000 inserts
   - Trial 2:
     - 1,000,000 inserts
   - Trial 3:
     - 100,000 inserts followed by 100,000 removes

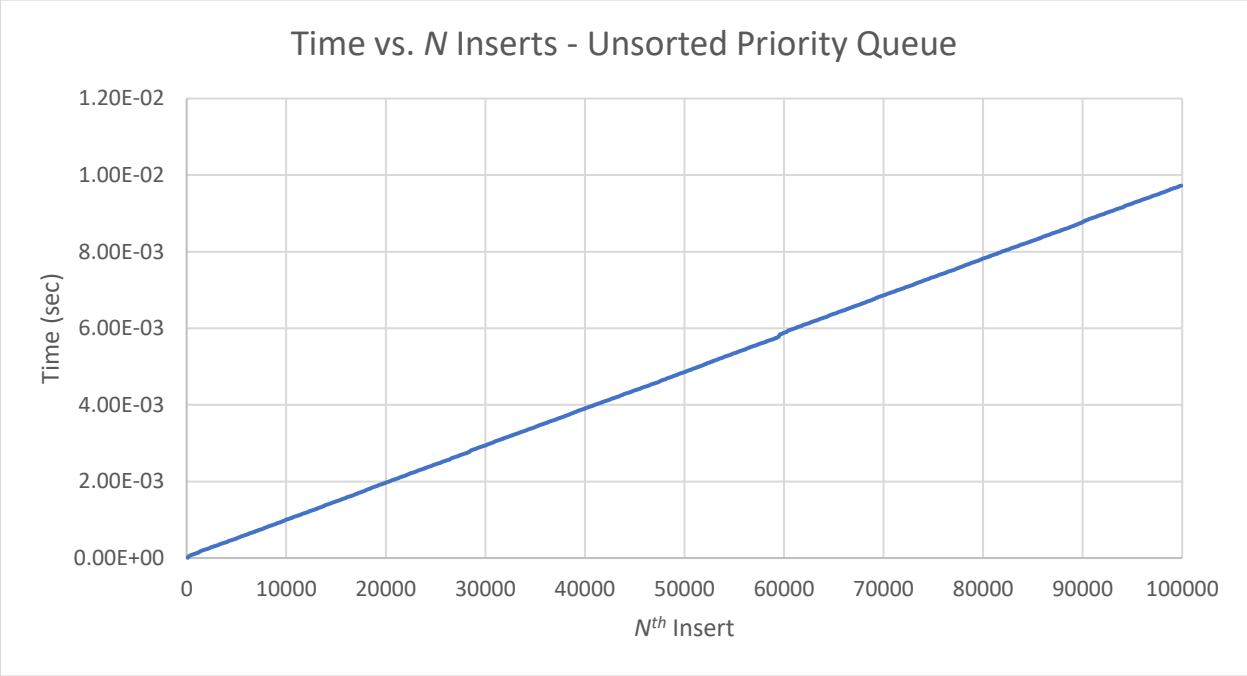
E. Heap Priority Queue
   - Trial 1:
     - 100,000 inserts
   - Trial 2:
     - 1,000,000 inserts
   - Trial 3:
     - 100,000 inserts followed by 100,000 removes

[2] Note: The boost specifications of this CPU vary throughout the time the program runs. Therefore, the comparison should only be made relative to the other tests conducted with the same test bench.

**Experimental Results:**

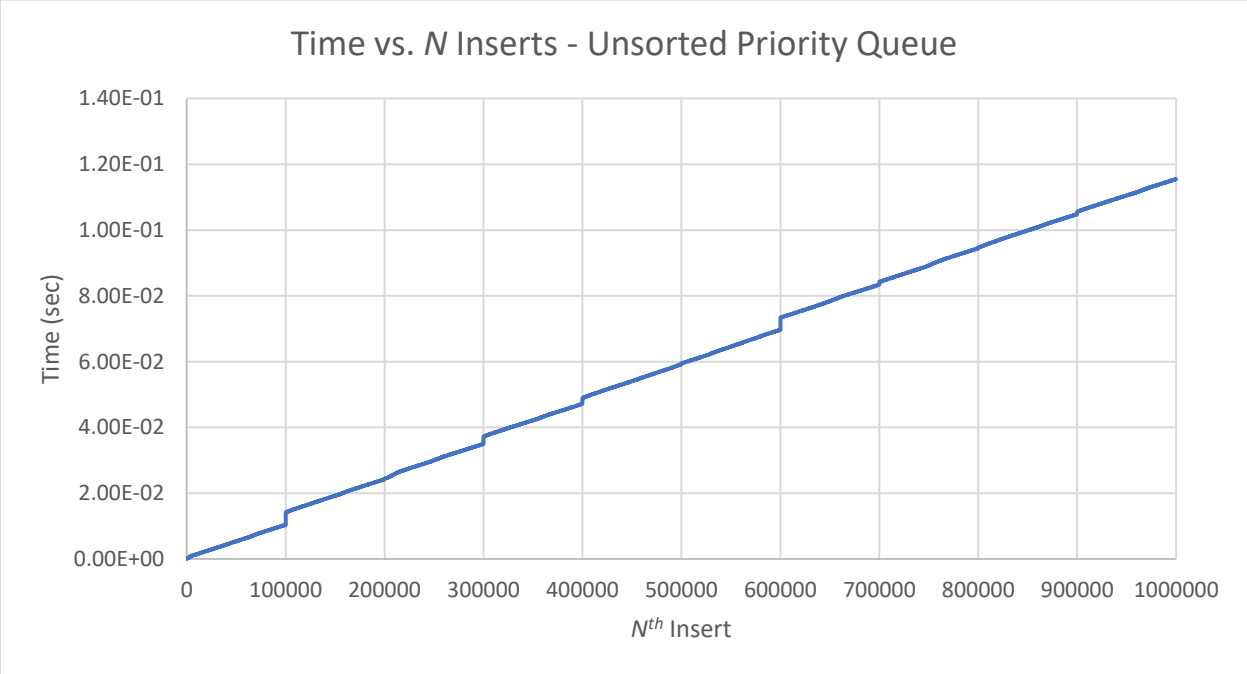
Unsorted Priority Queue Insert Only:

- 100,000 Inserts



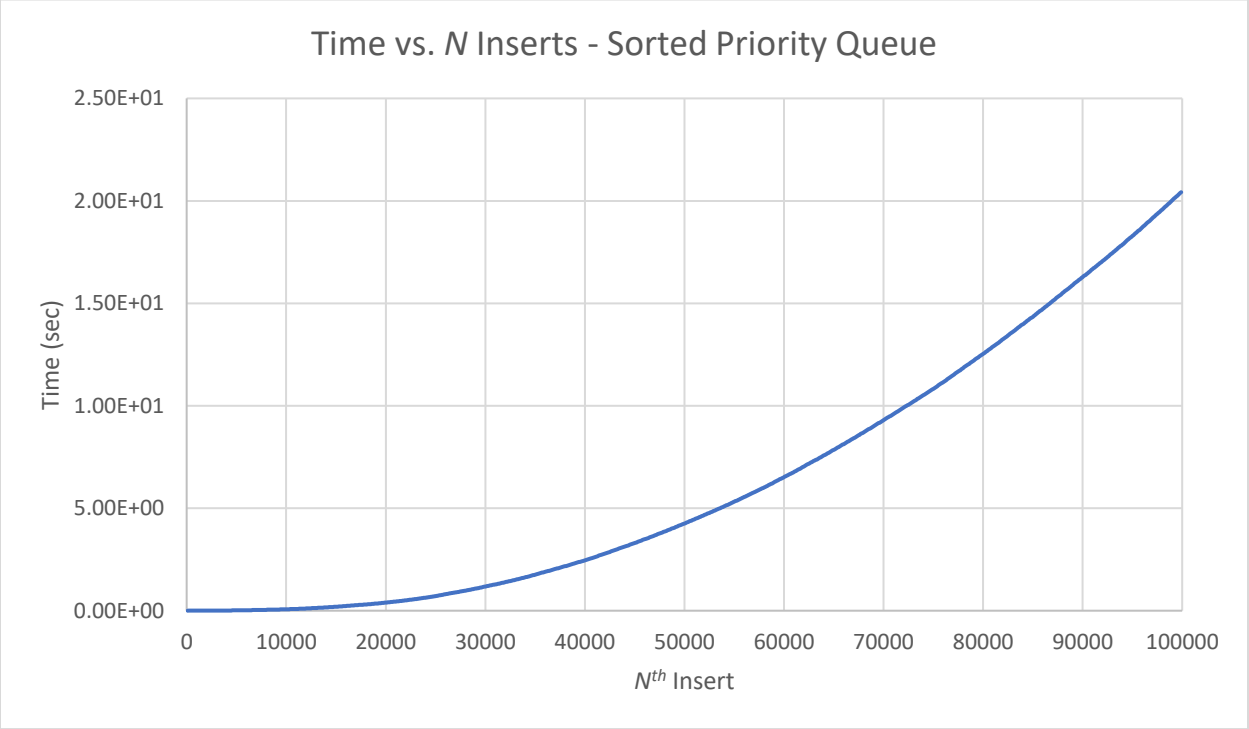

Unsorted Priority Queue Insert Only:

- 1,000,000 Inserts



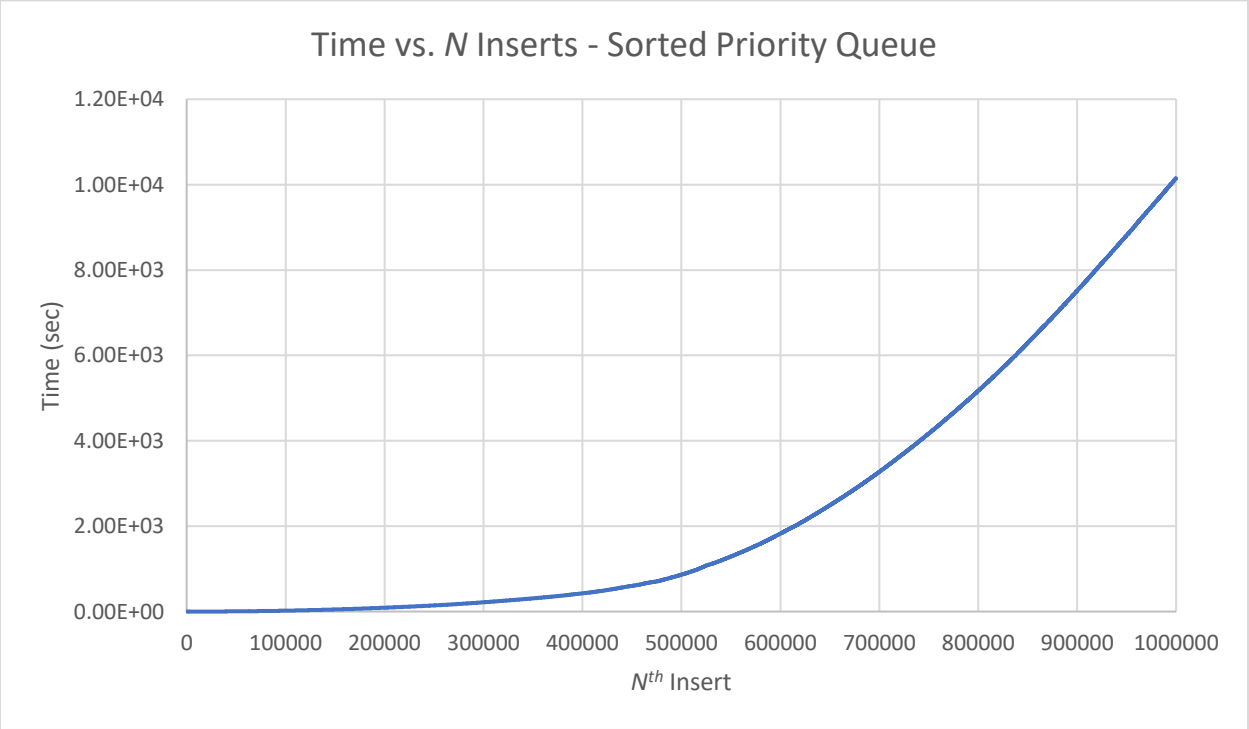
Sorted Queue Insert Only:

- 100,000 Inserts



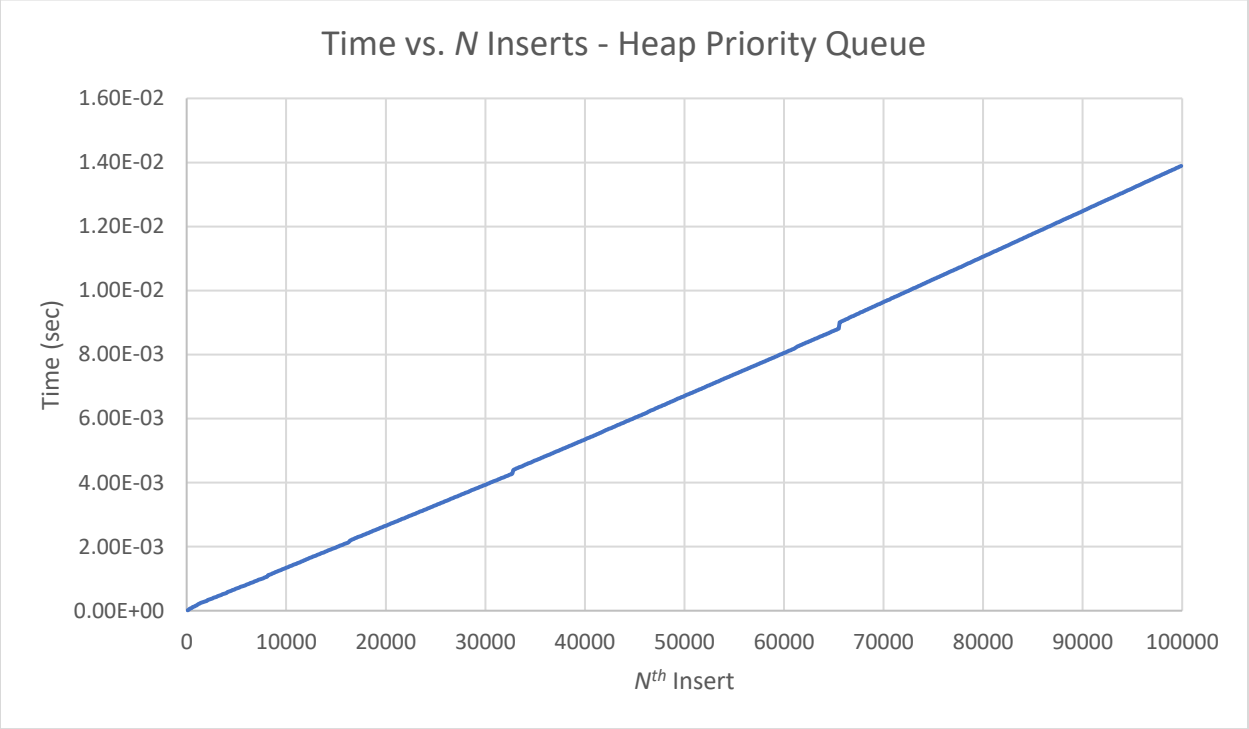

Sorted Priority Queue Insert Only:

- 1,000,000 Inserts



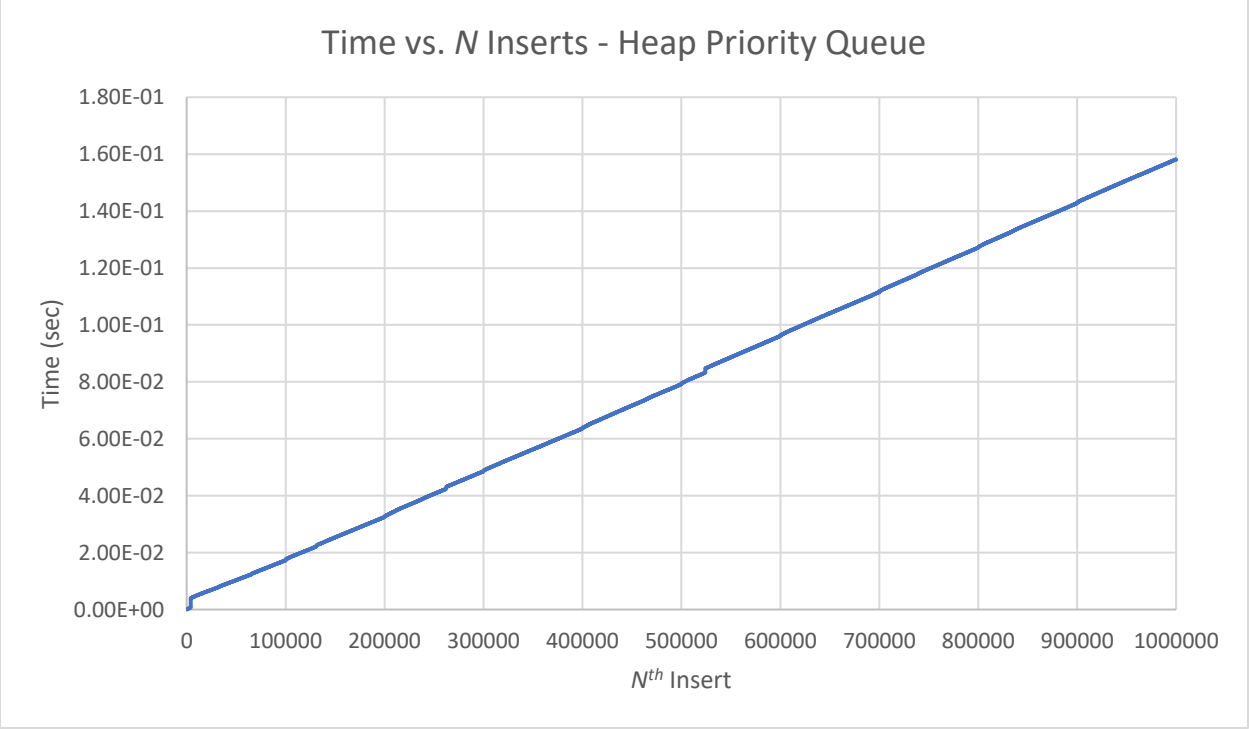
Heap Priority Queue Insert Only:

- 100,000 Inserts



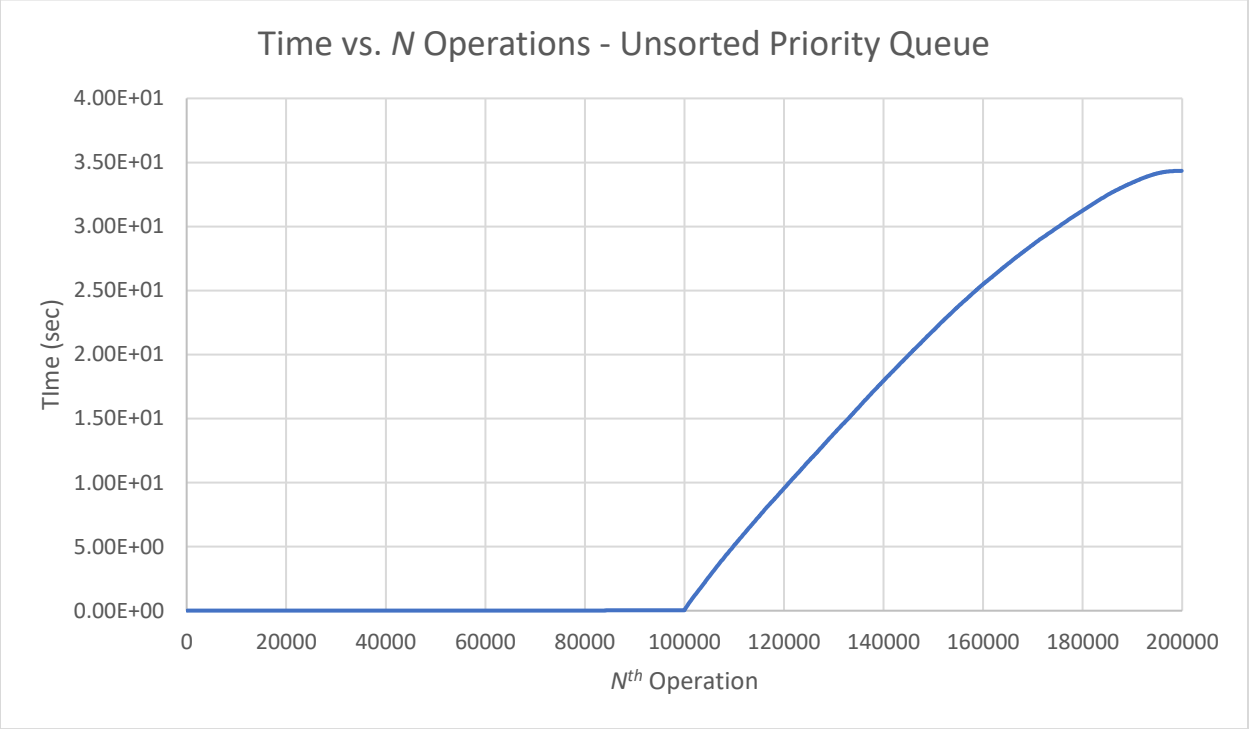

Heap Priority Queue Insert Only:

- 1,000,000 Inserts



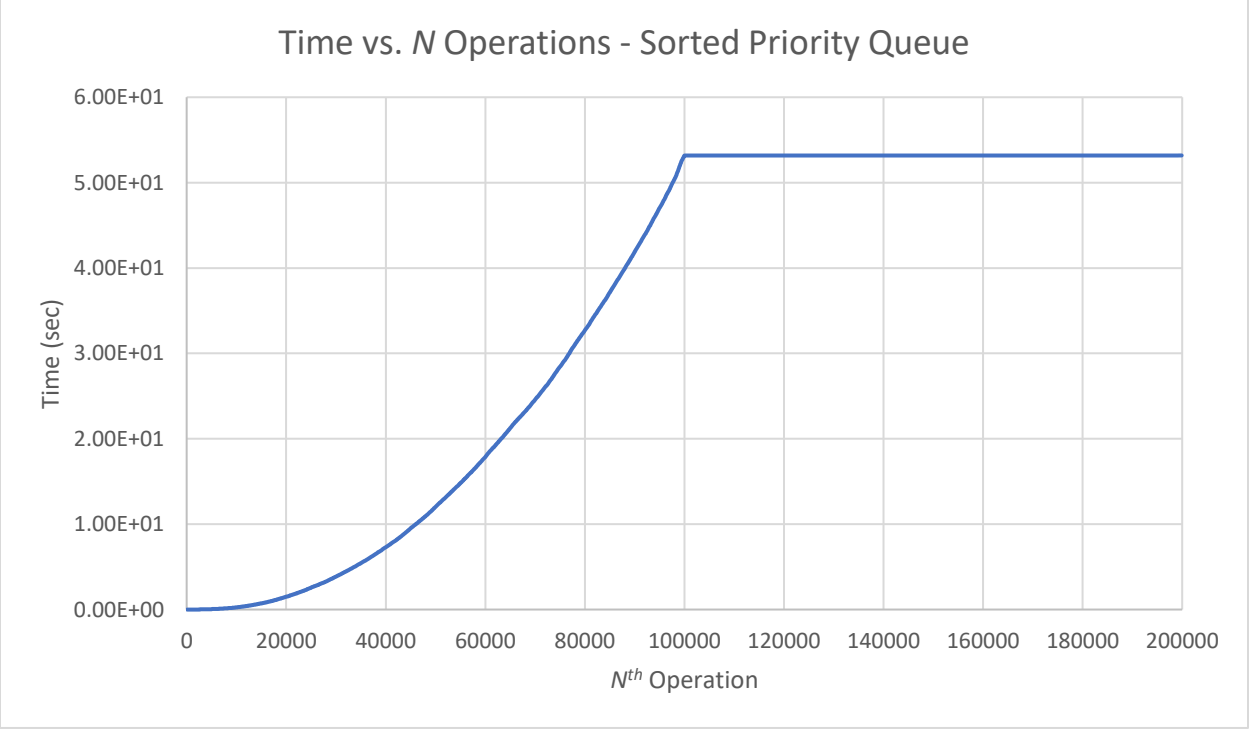
Unsorted Priority Queue Insert and Remove:

- 100,000 Inserts and Removes



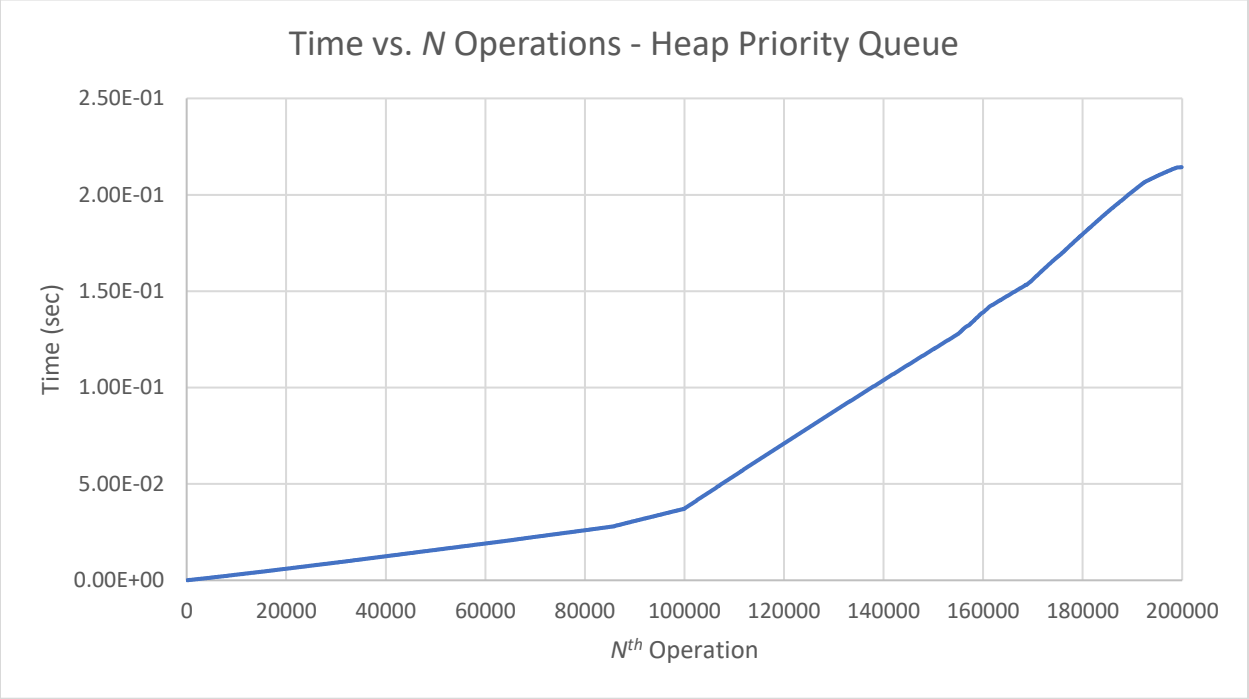

Sorted Priority Queue Insert and Remove:

- 100,000 Inserts and Removes

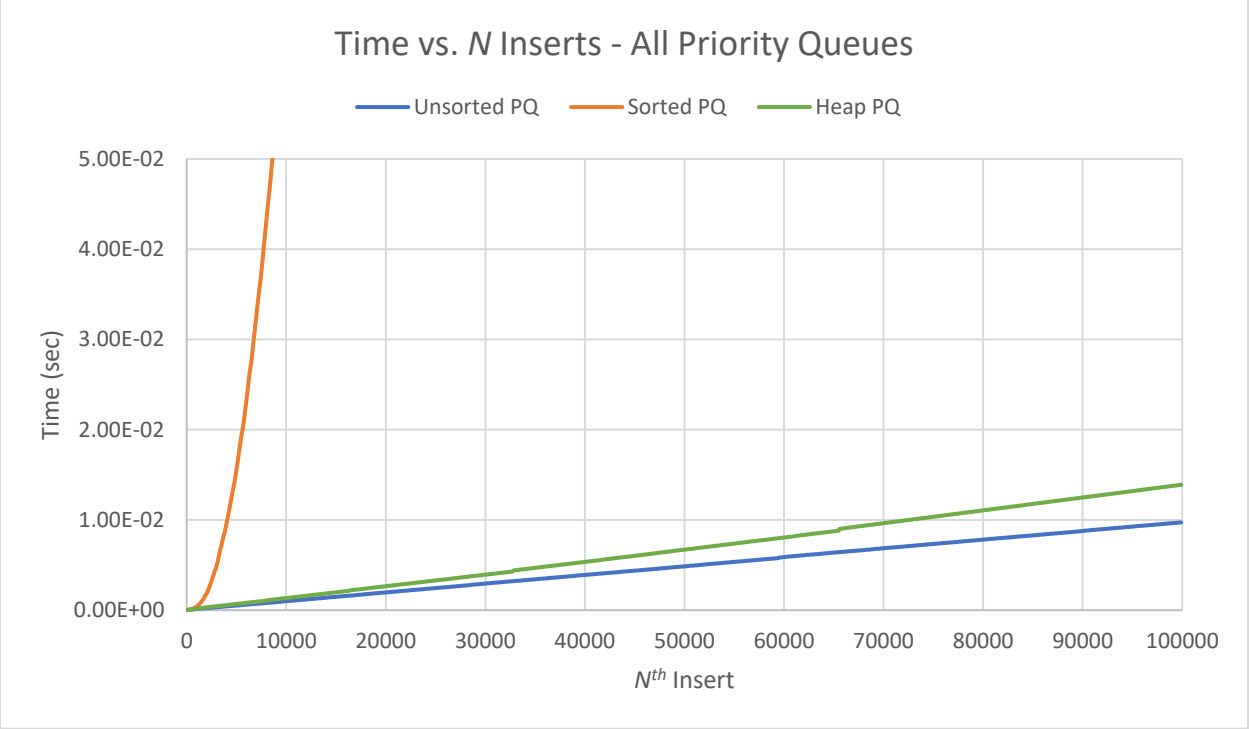


Heap Priority Queue Insert and Remove:

- 100,000 Inserts and Removes

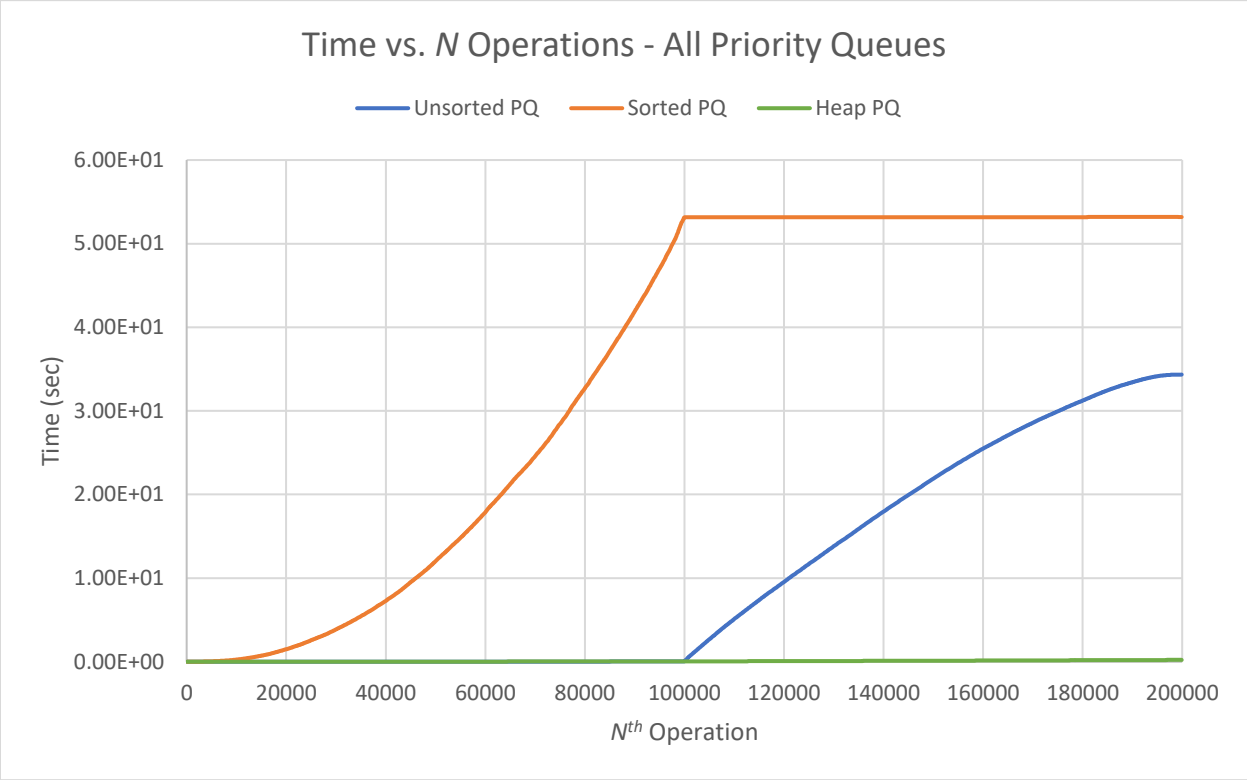



All Priority Queues Insert Only (Comparison)[3]:




[3] Y-axis for this graph has been limited to show the differences between all three queues' time complexities. In doing so, the chart does not show the whole line function of the sorted priority queue.

All Priority Queues Insert and Remove (Comparison):




**Conclusion:**

In terms of raw time for inserting elements, the unsorted priority queue was the best performing out of the three. This follows the theoretical analysis that the unsorted priority queue has an amortized cost of *O(1)* for inserting because of the fact that the queue only inserts the element to the tail of the linked list, causing the operation's time complexity to be constant no matter the size of the queue. However, when the queue removed all elements in the list, the sorted priority queue removed the minimum key values in the fastest time since its remove function has a Big-Oh of *O(1)*. This is because the sorted queue already had its elements arranged from highest priority to lowest priority allowing the queue to only need to remove the head of the linked list which is a constant operation. This implementation causes a very speedy dequeue for the sorted priority queue.

When combining both insert and remove operations for testing, it was found that the heap priority queue performed the best for the 100,000 inserts followed by 100,000 removes test. Although the heap priority queue's time complexity for inserting is not as good as the unsorted queue's (*O(logn)* vs. *O(1)*), and its time complexity for removing is not as fast as the sorted queue's (*O(logn)* vs. *O(1)*), its combined operation performance is the fastest, completing the 100,000 inserts and 100,000 removes in less than 0.25 seconds vs. the other two priority queues as can be seen in the graph above. This is because of the larger time complexities of the unsorted queue's remove function and the sorted queue's insert function. Since the unsorted queue's remove function is *O(n)*, it's amortized cost of inserting and removing is *O(n)* which is larger than the heap queue's amortized cost of *O(logn)* for inserting and removing elements. Likewise, the sorted priority queue's insert function is *O(n)* and its amortized cost for inserting and removing elements is *O(n)* which is also larger than the heap queue's

*O(logn)* insert and remove cost. The theoretical analysis of the three queues match the experiment conducted in this project as the heap priority queue performed the best because of its *O(logn)* complexity of sorting the elements in the queue using a vector as a binary-tree representation.