Justin Adkins
CSCI 2270
Final Project

Priority Queue Implementation Analysis

Purpose:
To analyze the complexity and efficiency of different Priority Queue implementations.

Testing Environment:
I chose to test my implementations under more rigorous guidelines than specified in the write-up because the data I was constructing initially had deviations due to inconsistent run times. I chose to build a much larger data file and test the run times. Using this data file, I was able to get a good feel for how scaling these data structures would affect their performance. This gave me a feel for the "big picture".

In order to accurately calculate when certain implementations were more efficient than others, I also focused on smaller data sets between 50 and 850 data members and ran these tests thousands of times in order to get an accurate mean for each.

I built helper functions to collect data for each of my implementations. These work by creating a file and feeding information to them as tests are ran. The program is built so one can modify number of times run, distance between tested queue lengths, and maximum queue length.

See code comments for other information on helper functions.

Data:

The data set used for this study contained a name, a primary priority, and a secondary priority. I chose to build a much larger data set than provided to test scalability, but both hold the same type of data. Ties with primary priority will be handled by secondary priority. If an occurrence occurs where both priorities tie, the data member who was first added to the queue will keep their position.

Linked List Implementation:
This implementation is in essence a sorted Linked List. For every insertion the run time should be O(n) and removal should be O(1) since we are just deleting the head and reassigning it. My implementation is in line with these standards as shown in my graphical data. Since our testing is the insertion and deletion of n data members, our final complexity is n times O(n) or O(n^2).

Enqueue works by dynamically creating a new object and then searching for its spot in the list, starting at the head. If the primary condition of the objects being compared is equal, the secondary condition is checked. Once the place in the list is found or the end is reached, pointers are re-arranged and the object is now sorted into the list.

Dequeue works by simply deleting the head and reassigning it to the next node in the list.

Heap Implementation:
This implementation follows the implementation of a standard binary heap with the use of an array as an abstraction of a tree. For every insertion and deletion, the complexity is O(log(n)). On n data members, our final complexity is O(nlog(n) + nlog(n)) or O(2nlog(n)). This is in line with our graphical data. We lose efficiency by inserting each member individually and rebalancing every time. Doing some further research on the matter I found that if the heap is randomly populated at first and then sorted the complexity is reduced to O(n + nlog(n)). This is due to an efficiency gain in the complexity of insertion.

****Dynamic Heap Implementation:**

I decided to implement a dynamically sized array as an extra implementation to see how the memory allocation / deallocation would impact performance. How this implementation works is when the array is full, its size is doubled. I made the starting value for the array to be 100 but modifying this wouldn't dramatically affect the scaled result. When an array is using less than half of its available space, it is cut in half. From our graphical data one can see that the impact was minimal. This was surprising since the operation of copying and deleting data seems bulky but considering the limited number of times the operation would need to be run, it does little to the run time.

This expense in run time would be in order to minimize memory as the Priority Queue grows and shrinks. Obviously, in a testing setting this application doesn't seem useful but, in a case where a list is being modified over time it could be important to manage the memory.

STL Implementation:

The STL implementation uses a similar heap design to the one which we wrote but due to some differences in implementation, is slower than our version. Analyzing the "push" method we see that two things happen. The method first calls "push_back", this is a function of the Algorithm library and adds a new data member to the back of a vector. The complexity here is constant unless reallocation is necessary. If that is the case, the complexity becomes linear. The method then calls "push_heap", another function of the Algorithm library which sorts the newly added last element of the vector into the heap. This has a complexity of O(log(n)) which is the same as our Heap implementation.

The "pop" method first calls "pop_heap" which is a function of the Algorithm library that moves the highest value member of the heap to the back of the vector. This function has complexity of O(log(n)) because it first compares and then swaps (if necessary) from top to bottom. The "pop_back" function is then called to remove the last data member of the vector.

The difference in performance is minimal when you look at larger scale. On a short scale, the impacts in performance are due to the multiple function calls and resizing of the vector. The implementation uses a method during dequeue of placing an element in the back and then discarding it, whereas in our implementation, we discard an element and then “heapify”. This movement would lead to a small performance hit as well.

Findings:

Looking at the “Run Time Analysis (5000 Run Average)” graph we can make out some interesting data between the lengths of 0 to 400 data members. The Linked List Implementation is more efficient than the STL Implementation until they intersect between 400 and 500. When considering why this is the case we must look to the efficiency of the STL. The STL will have a complexity of O(nlog^2(n)) where the Linked List will have a complexity of O(n^2). This is in line with my findings since initially the STL is slower, but quickly becomes more efficient as the queue scales.

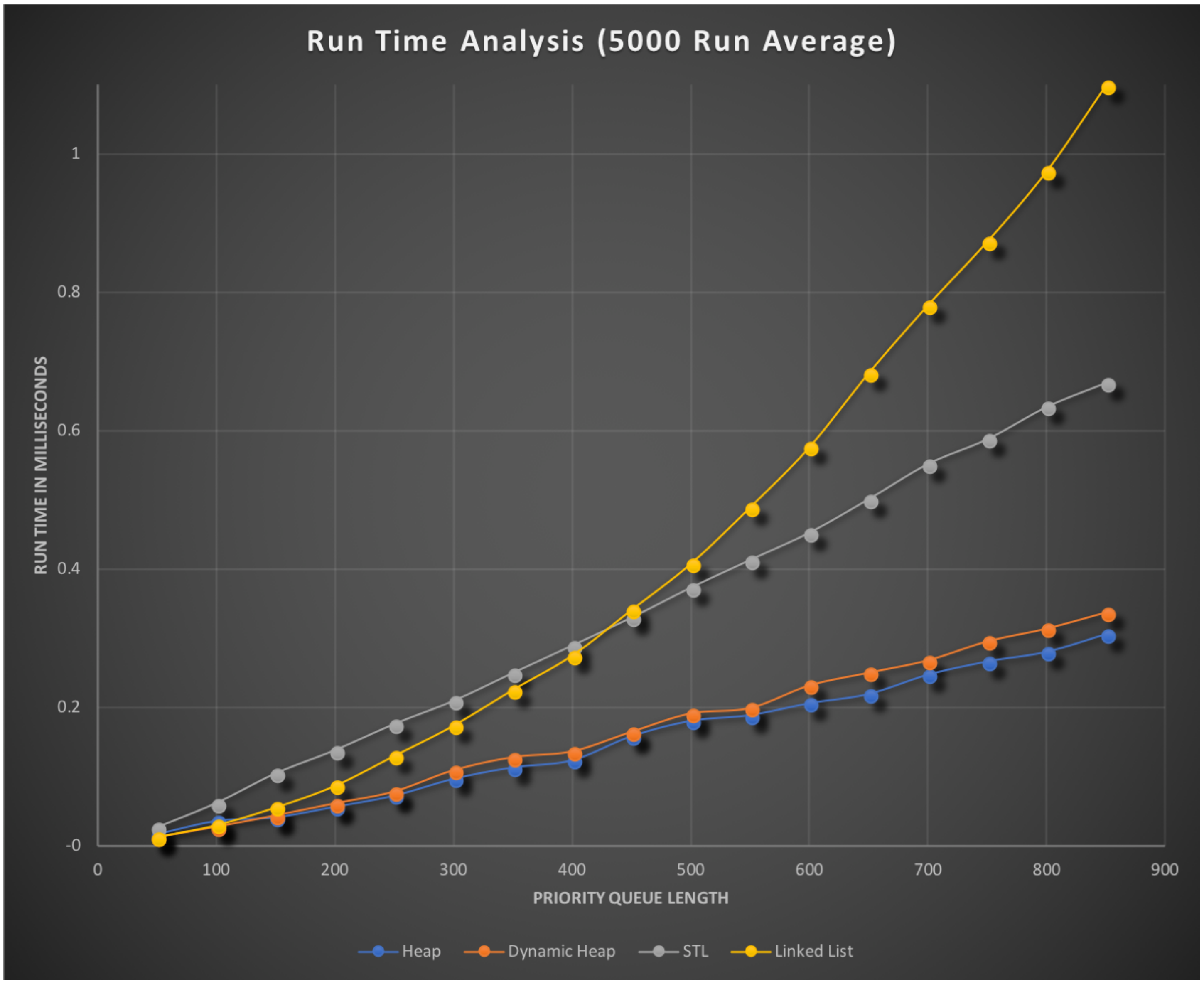
Looking at the “Run Time Analysis 10000 (2000 Run Average)” graph we can see how dramatic the Linked List complexity scales. The graph moves exponentially upward as the other three implementations logarithmically grow. Removing the Linked List from the implementation we can see that the self-implemented Heap is still the most efficient. The STL actually becomes increasingly less efficient to our implementation as the size of data grows, with a run time more than twice that of our implementation at 10,000 data members.

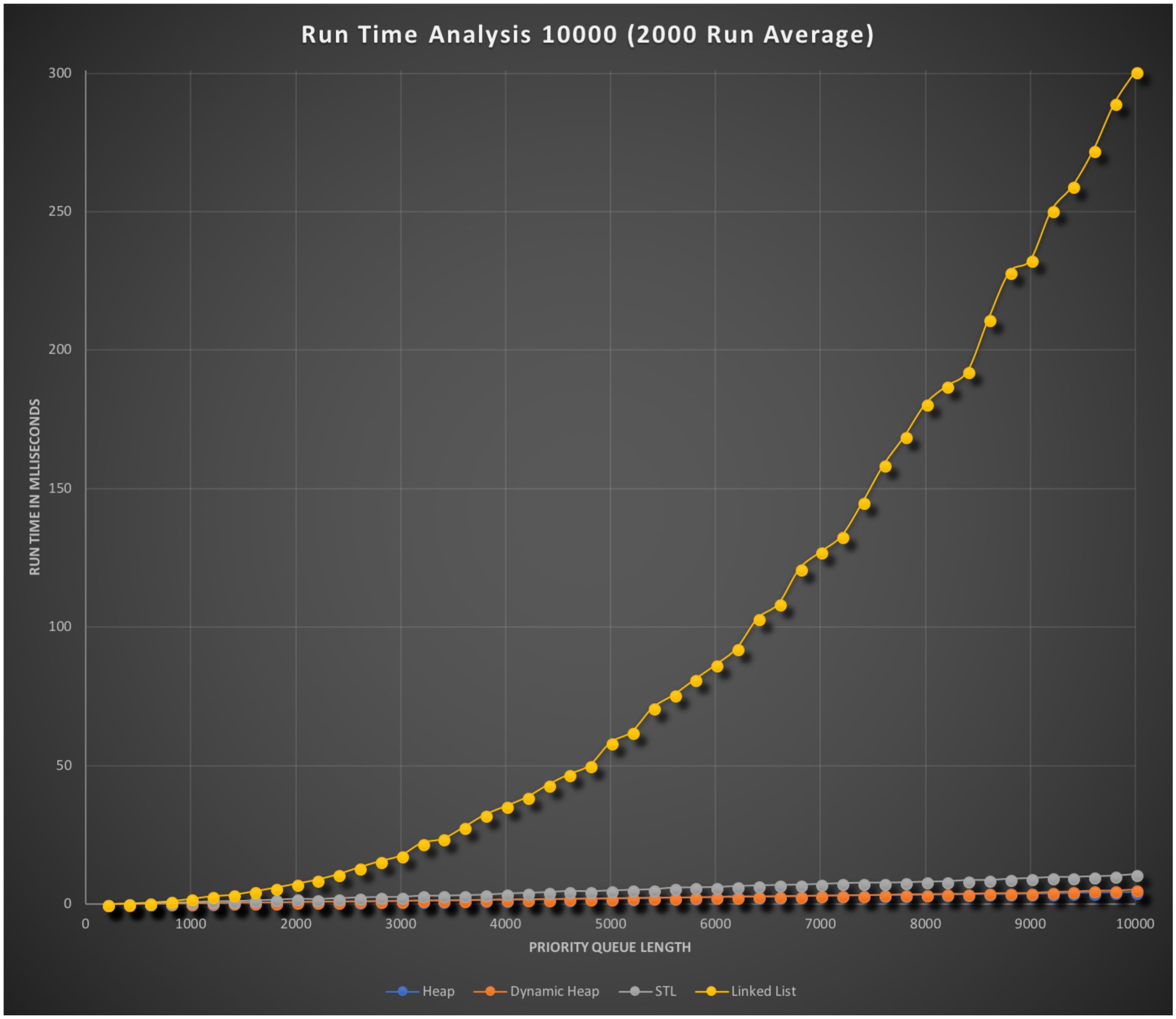
Other Observations:

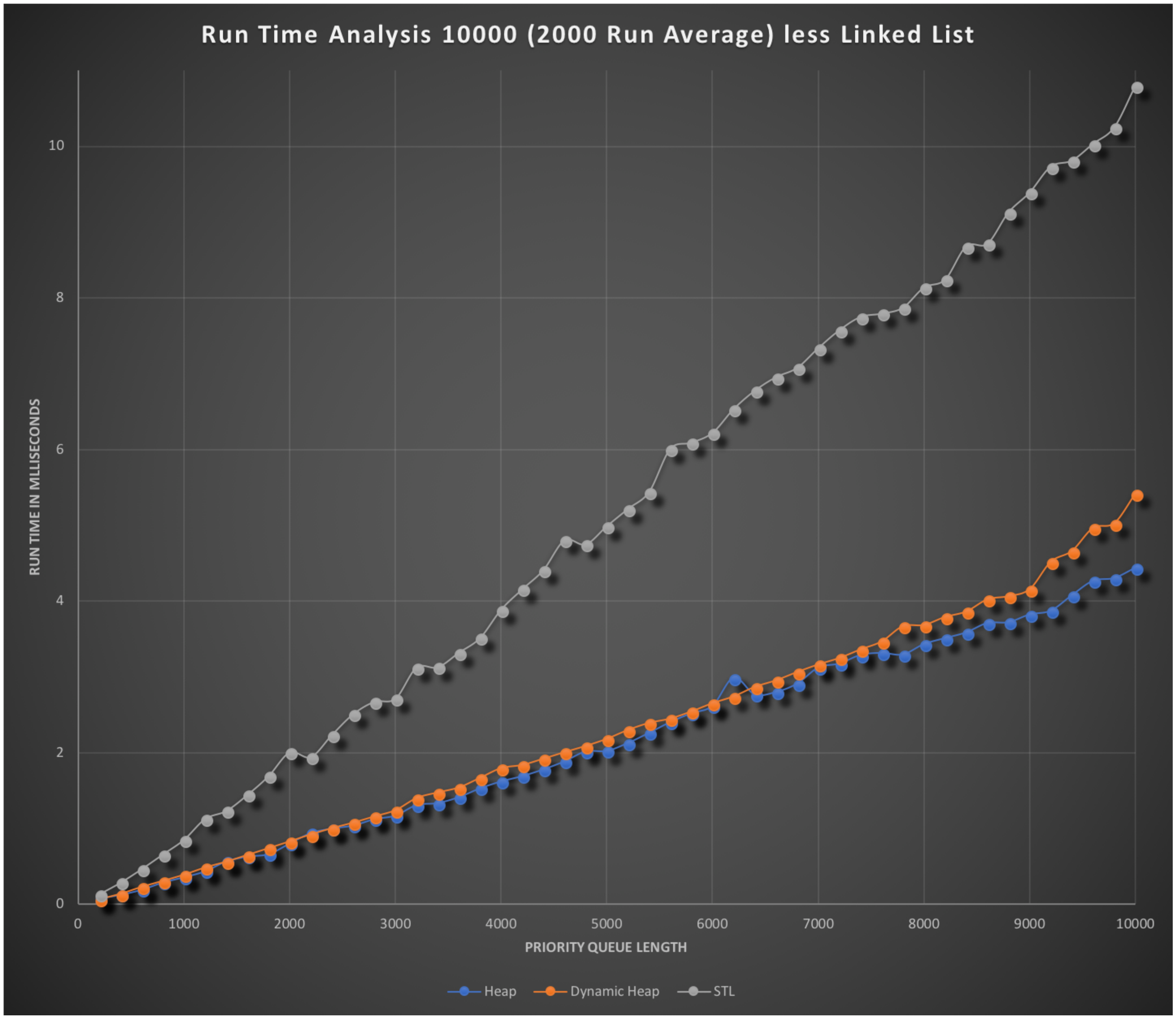
While testing my code over the past few weeks I noticed hardware impacts on processing power that would dramatically slow down my run times. The two that I noted were:

- Low battery
- Sleep mode / closed screen

Both of these must have sent my processor into a slowed state to conserve power. My run times dramatically grew, ruining portions of my data and requiring a retest.


Run Time Analysis (5000 Run Average)
RUN TIME IN MILLISECONDS
1
0.8
0.6
0.4
0.2
-0
0
100
200
300
400
500
600
700
800
900
PRIORITY QUEUE LENGTH
Heap
Dynamic Heap
STL
Linked List


Run Time Analysis 10000 (2000 Run Average)
RUN TIME IN MLLISECONDS
300
250
200
150
100
50
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
PRIORITY QUEUE LENGTH
Heap
Dynamic Heap
STL
Linked List



Run Time Analysis 10000 (2000 Run Average) less Linked List
RUN TIME IN MLLISECONDS
PRIORITY QUEUE LENGTH
0
2
4
6
8
10
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Heap
Dynamic Heap
STL