## Choosing data structure for scheduling

Having a scalable and efficient simulation environment is very dependent on the data structure we use to maintain the events and the agents for scheduling. MUSE has a two tier scheduling system. The very top tier is the scheduler and it maintains the agents and knows which agent to process at any given time. The second tier is in the agent. All incoming events to a given agent must be stored and correctly scheduled in increasing fashion according to the time of the delivery. The heap data structure seemed a great fit for both tiers. The heap data structures under consideration are the Fibonacci heap (Fredman and Tarjan) and the Binary Heap. Binary heaps are heaps that are implemented with binary trees (Wikipedia). Fibonacci heaps have very impressive runtime results, however these results are amortized. The following table shows the runtimes of both binary and Fibonacci heaps.

|  |  |  |
| --- | --- | --- |
| Standard Operations | Fibonacci Heap | Binary Heap |
| Insert | O(1) | O(log\*n) |
| Get Min | O(1) | O(1) |
| Delete Min | O(log\*n) <amortized> | O(log\*n) |
| Decrease Key | O(1) <amortized> | O(log\*n) |
| Delete | O(log\*n) <amortized> | O(n) |
| Merge | O(1) | O(m log(n+m)) |

Fibonacci heap showed impressive runtimes, but we wanted to know just how much we have to amortize before we realize the gains. Binary heap on the other hand has good runtimes and no amortized costs. The two tiers make more use of different operations. Hence, there is a good chance that we would end up using a combination of the two heaps in MUSE. The first task we have done is identifying which operations were frequent in each tier. The first tier, once we add the agents we should never remove until the end of simulation. Therefore the only operation we want to compare is the *decrease key* operation. Decreasing the key in short is just an operation to reorder an element in the heap. We can draw an early conclusion here and say that Fibonacci heap should be used, but it is better to let the numbers speak. In the second tier, we frequently made use of the *insert, get min, and delete min*, whenever there was a rollback we also used the *delete* operation.

For binary heap implementation we will be using the *priority\_queue* from the C++ STL containers. Fibonacci heap we have found a nice C++ implementation. To get the source for the fibonacci heap implementation follow this reference (Kühl).

### 1.1.1 Fibonacci vs. Binary testing procedure

We have to find a good heap for both tiers, and we already discussed the heavily used operations for both tiers. With the first tier we want to test the key decreasing. To get a good idea we have a couple of controlled variables. We have fixed the time steps to 400. This allowed us to see a nice difference in performance between the two heaps and the time to run the tests was reasonable. The basic idea is to keep increasing the number of agents, starting from 100 and ending at 100,000 agents. At each time step we will iterate over the number of agents and randomly (P = .5) increase or decrease the value of the agent’s key, and call the *decrease* operation on the key. We will keep track of the time it takes to execute and take the average of five runs for each increase in the number of agents. Fibonacci heap implementation has a *change(element, key)* method which we can use. However, the priority queue does not implement a way to change the key, so the solution is to pop the top element and then update the value and push it back into the heap. This makes the runtime from *O(log\*n)* to *O(log\*n+log\*n)*. We simply added the runtimes for *delete min* and *insert* to get the updated runtime.

The second tier deals with events. To actually see something meaningful we fix our time steps to 5000 iterations. We will slowly increase the number of events in the heap starting from 100 events per time step all the way to 100,000 events per time steps. There are two cases to test in the second tier. First case is just going to be a test to see how long it takes to insert *X* number of events and then *delete min* until the heap is empty again. The second case is testing how long it takes to delete arbitrary elements from the heap. We will use the Iterators and just keep calling the *delete* operation and see which has the best time. The STL container *priority\_queue* does not support Iterators. In order to get elements in the back, we would have to remove all the elements and store the valid ones into a temporary storage. Once we remove the invalid elements, we would then push all the elements from the temporary storage back into the priority queue. Like the first tier we will run each five times and get the average time. The big deciding factor for the second tier will be the first case, as this is the most frequent operations. However, since *priority\_queue* does not support Iterators, we must add the second case into the comparison test.

### 1.1.2 Fibonacci vs. Binary data collection, results, and discussion

The table below is the collection data when we compared the two heaps for tier one. Keep in mind that that the execution times are the average of five runs and represent execution time in seconds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agents | Time steps | Fibonacci execution time | Binary execution time | Speedup |
| 100 | 400 | 6 | 16 | 2.66 |
| 1000 | 400 | 78 | 222 | 2.84 |
| 10000 | 400 | 949 | 2721 | 2.86 |
| 100000 | 400 | 14090 | 34179 | 2.42 |

The graph above shows clearly the trends we expected. The results above were expected because for binary the best and worst case is *O(log\*n+log\*n).* However, fibonacci heap has a best amortized run time of *O(1),* but the worst case is *O(log\*n).* Derived proof of worst case times for fibonacci heap can be found in the reference (Fredman and Tarjan). From these results the choice for tier one is fibonacci heap.

The table below is the collection data when we compared the two heaps for tier two. Here are the execution times of case one and case two combined as discussed earlier.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time | Speedup |
| 100 | 5000 | 2 | 1 | 2 |
| 1000 | 5000 | 23 | 18 | 1.27 |
| 10000 | 5000 | 300 | 203 | 1.47 |
| 100000 | 5000 | 3236 | 1794 | 1.80 |

The results for tier one were expected, however the results from the test for tier two revealed surprising information. The amortized cost of the *delete min* operation proved to be too great. Since *priority\_queue* did not support Iterators, we had to be fair and add the time to actually remove arbitrary elements from the heap. We purposely took the naïve approach and just popped all elements into a temporary storage and pushed in the valid elements back into the heap. Fibonacci still proved to be slower than the binary heap. From the results we can clearly conclude that the amortized run times claimed by fibonacci heaps would require very large data in the heap and for our purposes was not needed. Hence, for tier two, we will use the binary heap, but can we do better?

For tier two, you can see from the speedup column in the table that while there was speedup, they were not great. This was due greatly to the performance hit we take from *priority\_queue* and its lack of iterators. Earlier we showed the runtime to delete arbitrary elements from the binary heap took worst case *O(log\*n+log\*n).* We believe the speedup would be even greater if we had the worst case to the original runtime of *O(n).* This need to improve our performance was the motivation behind the development of the *BinaryHeapWrapper*. This implementation used a vector and represented it as a binary heap. Most importantly it allow us to remove arbitrary elements at *O(n).* *BinaryHeapWrapper* was specialized for MUSE and the disadvantage being reusability, the binary heap implementation could only be used within the MUSE framework. We ran the tests for tier two and compared *BinaryHeapWrapper* against *priority\_queue* and fibonacci heap*.* The following table shows the results from the experiment. Since we have two binary heaps, we decided to change the names in the table to the actual class names.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time | BinaryHeapWrapper execution time |
| 100 | 5000 | 2 | 1 | 1 |
| 1000 | 5000 | 23 | 18 | 13 |
| 10000 | 5000 | 300 | 203 | 125 |
| 100000 | 5000 | 3236 | 1794 | 1041 |

We did not show the speedup in the table above, but we can clearly see a dramatic increase in performance from the custom built binary heap.

From these experiments and results we were able to show that although fibonacci heap had great amortized times, there are cases where the benefits do not outweigh the cost. However, there was a case where fibonacci turnout to be very beneficial and hence us using it for tier one. Using fibonacci heap for tier one, means that fibonacci heap is really great when you have a case where you have frequent key changes and minimum popping from the heap. This was the exact scenario for tier one. Lastly, we showed that priority\_queue actually turned out to be faster for tier two over fibonacci heap, but its lack of iterator support motivated the need for a custom built binary heap. The BinaryHeapWrapper turned out to have the best time and speedup out of all three heaps and thus was chosen as the heap for tier two.