

Experimental Performance Analysis on Priority Queue Implementations

Geoffrey Foster
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
gfoster@scs.carleton.ca

April 22, 2008

Abstract

Priority Queues are an abstract data-type similar to queues but each item in the queue has an assigned priority and the front of the queue will always have the smallest priority. They are often used in discrete event simulators to schedule simulated events that are to occur at a specified simulated time in the future. Events can be scheduled to occur at any point into the future, from mere moments to centuries. The underlying data-structure for a priority queue could be any number of different data-structures that provide at least the three operations of insert, peek and remove_min. By implementing and testing different data-structures for priority queues we can see how well they perform compared to one another and under what circumstances.

1 Introduction

A priority queue is an abstract data-type that provides three operations, insert (with an associated priority), remove (the element with the highest priority) and peek (at the element with the highest priority without removing it).

Priority queues are often used in discrete event simulators. Events are added the queue with their simulation time as the priority. Events in the simulation are executed repeatedly by taking the first element in the queue and performing some designated action on it. Additionally, Dijkstra's algorithm can use a priority queue for a more efficient implementation and the A* search algorithm does use a priority queue.

For this paper, four different implementations of priority queues were constructed and their experimental running times compared for the insert and the delete_min operations. The peek operation was not analyzed because for each of the implementations there is an $\mathcal{O}(1)$ running time for the peek and in the implementations it is merely a pointer lookup/follow. The four different underlying data-structures for priority queues that we will look at are: heater, minimum binary heap, pairing-heap and randomized meldable priority queue.

In section 2.1 we examine the experimental running time of n insertion operations. Then, in section 2.2 we examine the experimental running time of the removal of n elements. In

section 2.3 we examine how each of the priority queues handles a random ordering of insert and remove operations. Finally, in section 3 we look at the memory usage of the four different implementations after the insertion of n elements.

2 Experimental Running Time

For the following experiments three different data-sets were generated of size n . The first was in increasing order, going from 0 to $n - 1$. The second was in decreasing order, going from $n - 1$ down to 0. The last was made up of randomly generated data created using the *C rand* function (with duplicates allowed).

All tests performed on a quad-core 2.83 GHz Intel Xeon (“Harpertown”) with 2 GB DDR2 RAM, 2x6 MB L2 Cache and a 1333 MHz FSB. The code was written in C and compiled using GCC 4.0.1 with -O3 optimizations and built as 64-bit binary. Each test was performed twenty times and results were averaged.

2.1 Insertion of n elements

The first operation we will look at is the insert operation. The randomized meldable queue has an expected $\mathcal{O}(\log n)$ runtime for insertion [4] as does the heater [1]. A minimum binary heap has a running time of $\mathcal{O}(\log n)$ [2] for insertions with an amortized running time of $\mathcal{O}(1)$. Likewise, a pairing-heap has been shown to have an amortized insertion time of $\mathcal{O}(1)$ [5].

For the following experiments all n elements from each of the data-sets described in section 2 were inserted, one by one, into the priority queue. Figure 1, Figure 2, and Figure 3 show the results for the increasing, decreasing and randomly generated data-sets respectively.

Each of the four implementations performs similarly, whether dealing with increasing, decreasing or randomly generated data, up until $n \approx 10^5$ at which point very noticeable changes begin to occur.

The min-heap benefits the most from increasing data because its best case insertion time of $\Omega(1)$ occurs for all n insertions. The pairing-heap has very similar performance to the min-heap. The randomized meldable queue performs poorly which is likely because the expected number of meld operations performed is $\mathcal{O}(\log n)$ for all n insertions. The heater has similar performance because it will have to walk down the tree for an expected $\mathcal{O}(\log n)$ steps in order to do an insert operation for all n insertions.

The min-heap is generally hurt the most from decreasing data because every insert operation will result in a worst case scenario requiring $\mathcal{O}(\log n)$ time. This is because each time an element is inserted as a leaf it will always have to be rotated up to become the root requiring $\log n$ rotations per insertion. The heater and randomized meldable queue were all able to handle decreasing data much better than the increasing data. With 10^6 insertions requiring at most 70 seconds. The increasing order data required more than 10x that requiring at most 800 seconds. For a randomized meldable queue this is because each insert will be a single operation where the new priority value (that is less than all others currently in the queue) becomes the new root and the original tree becomes either the left or right child (chosen at random).

The min-heap and pairing-heap are able to handle random data quite well. The randomized meldable queue has decent performance and appears to have roughly the average of the increasing and decreasing data-sets performance whereas the heater continues to have rather poor results.

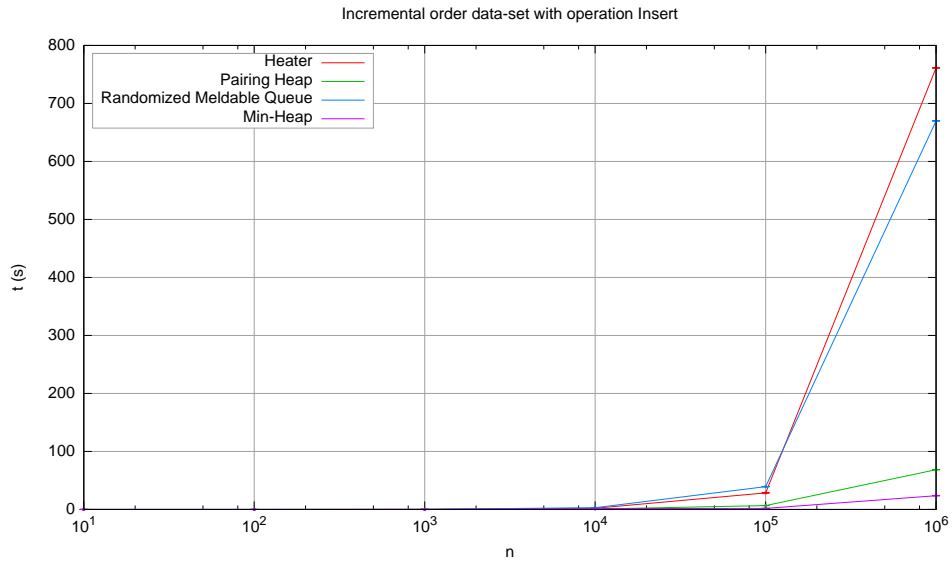


Figure 1: Insertion of n incremental ordered elements

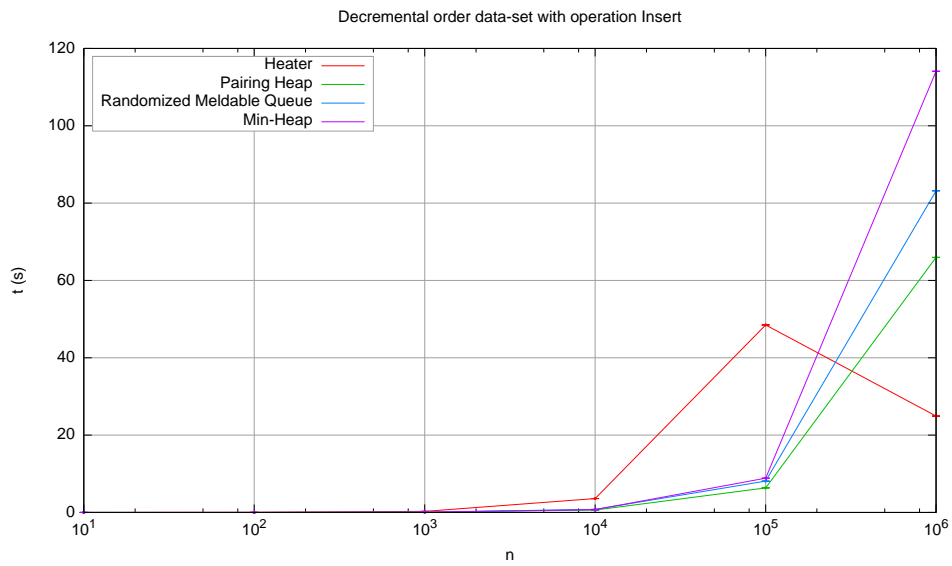


Figure 2: Insertion of n decremental ordered elements

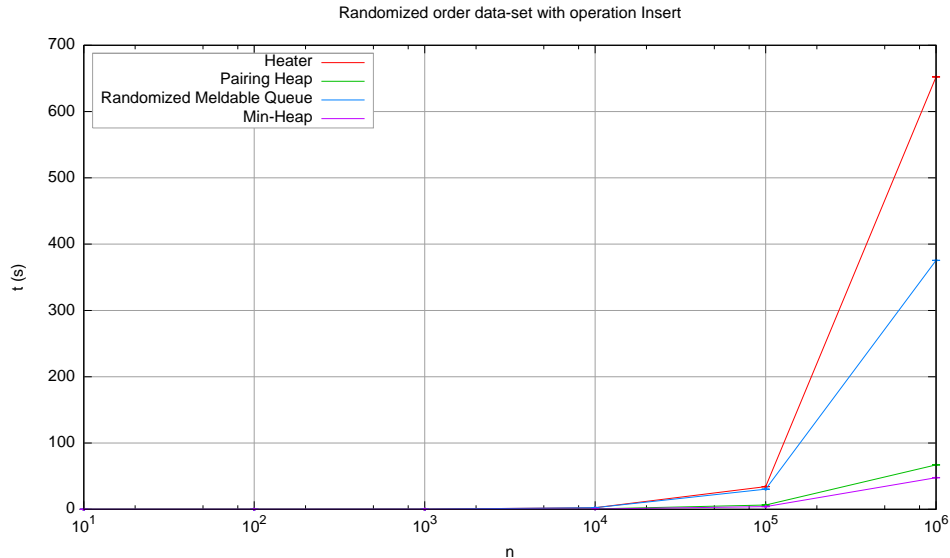


Figure 3: Insertion of n randomly ordered elements

2.2 Removal of n elements

Next we test the speed at which n elements are removed from the priority queue. This was done by taking the n insertions done in section 2.1 and removing them one by one.

The randomized meldable queue has an expected $\mathcal{O}(\log n)$ runtime for `delete_min` [4] as does the heater [1]. A minimum binary heap has a running time of $\mathcal{O}(\log n)$ [2] for deletions with an amortized running time of $\mathcal{O}(1)$. A pairing-heap has been shown to have a `delete_min` time of $\mathcal{O}(\log n)$ [3].

Figure 4, Figure 5, and Figure 6 show the results for the increasing, decreasing and randomly generated data-sets respectively. Like the insert operation, the `delete_min` operation for each of the implementations performs similarly until $n \approx 10^5$.

The min-heap continues to provide good performance, which is expected because of its balanced binary tree structure. The pairing-heap performs well when the data was inserted in either decreasing or increasing order but appears to perform the worst when it was constructed with random data. With random data the randomized meldable queue provides results very similar to that of the min-heap. With incremental data it performs quite poorly. The heater still sees poor performance compared to the other implementations.

2.3 Random order of n insertions and n removals

This set of testing models a more realistic usage of a priority queue where insertion and removal operations happen at anytime without any specific order. This type of occurrence would be seen in a discrete event simulator where the removal of an event from the queue and subsequent processing of that event could result in additional items being added to the queue. Using random data this is further extended to model the addition of events to the queue that are to occur in the near future and far future.

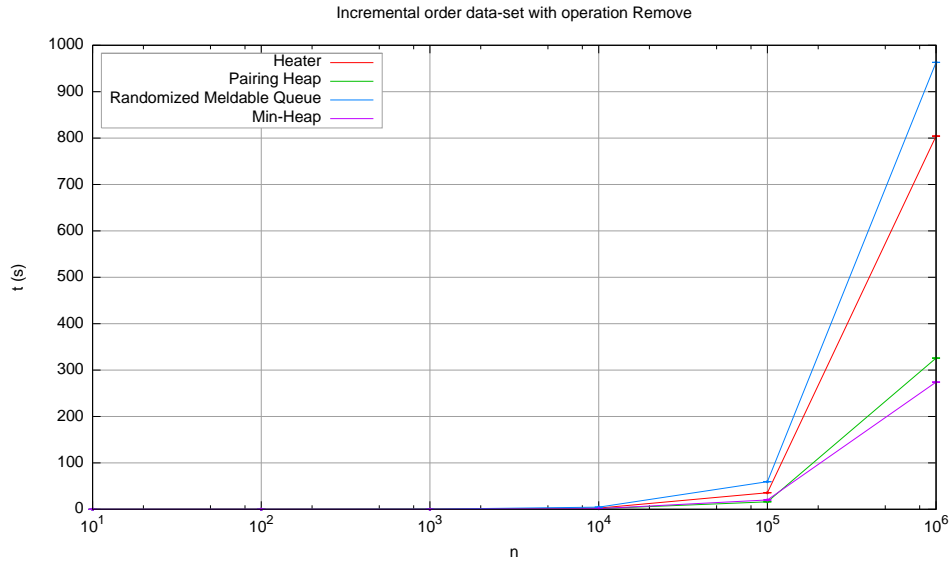


Figure 4: Removal of n elements inserted in incremental order

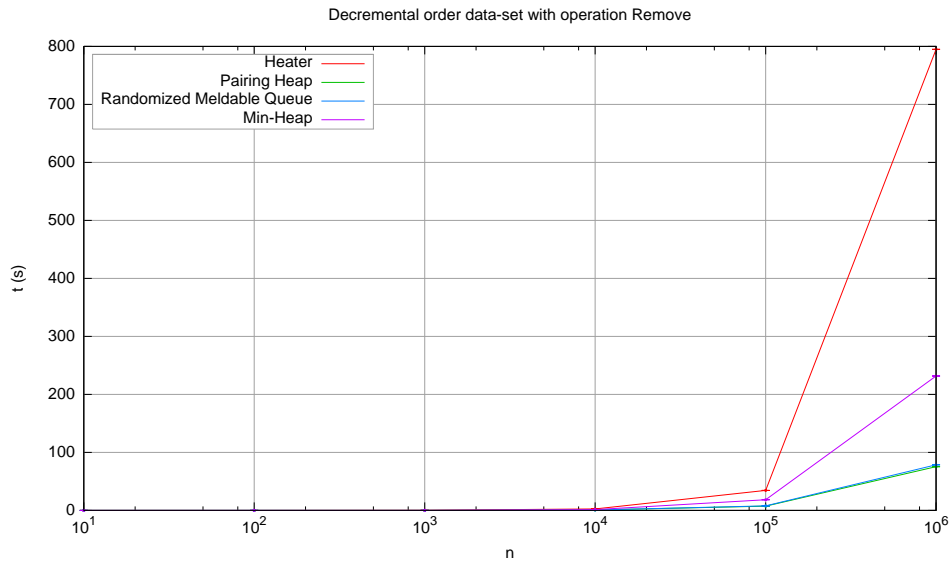


Figure 5: Removal of n elements inserted in decremental order

In order to test this scenario an array of size $2n$ was created where each item in the array was either an insert or a delete_min operation chosen at random but with the following restrictions:

- the first operation must be an insert
- the last operation must be a delete_min
- a delete_min can only be done if there is at least one item in the priority queue

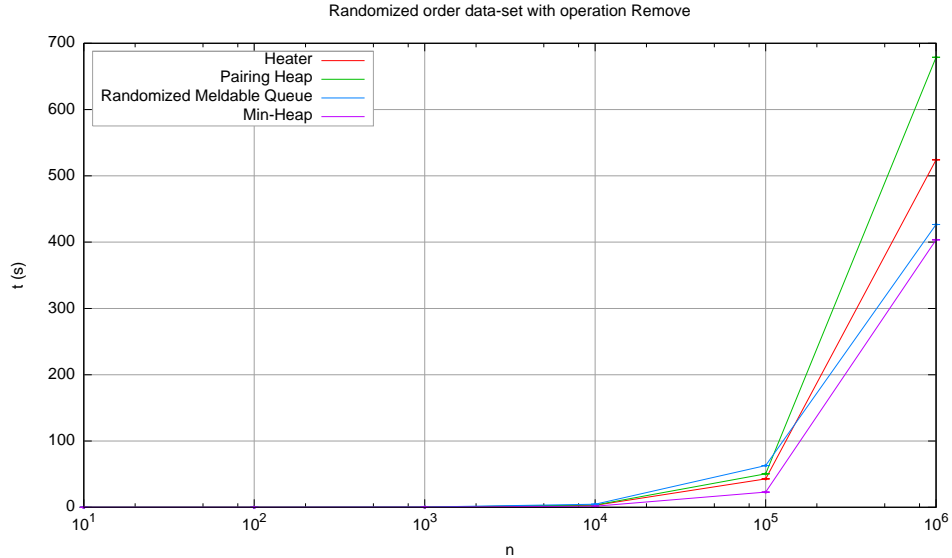


Figure 6: Removal of n elements inserted in random order

- the number of insertion operations must be n and the number of delete_min operations must be n

The minimum binary heap and the pairing-heap continue to handle themselves quite well with all data-sets. The randomized meldable queue continues to suffer with increasing data (Figure 7). This was also observed with both the insert (Figure 1) and remove (Figure 4) operations. In the decreasing data and random data cases it performs similarly to that of the min-heap and pairing-heap. The heater continues to disappoint, performing poorly in all cases.

3 Memory Usage

The min-heap implementation used can be over-zealous in its allocation of memory. It stores elements in an array until that array is full and then doubles in size. This could potentially make the min-heap require twice as much memory as it actually would need but also limits the number of times the array must grow.

Heaters, pairing-heaps and randomized meldable queues are all tree based structures but a randomized meldable queue uses less memory than the other two because it only stores a pointer to the left and right children, a pointer to the data and the priority. A heaters stores this same information but also a parent pointer and a random double value, k . This will make a node in a heater store about 1.5x as much memory as a node in a randomized meldable queue which can be seen in Figures 10 through 12. Similarly, a pairing-heap stores 3 pointers, the left child, a previous pointer and a sibling pointer as well as a pointer to the data and a priority.

Figures 10 through 12 also show that no matter the order of data being inserted (increasing, decreasing or random) the amount of memory will stay the same. This is obviously the case because no matter the ordering of the data there will still be n elements inserted, each taking up their required amount of memory.

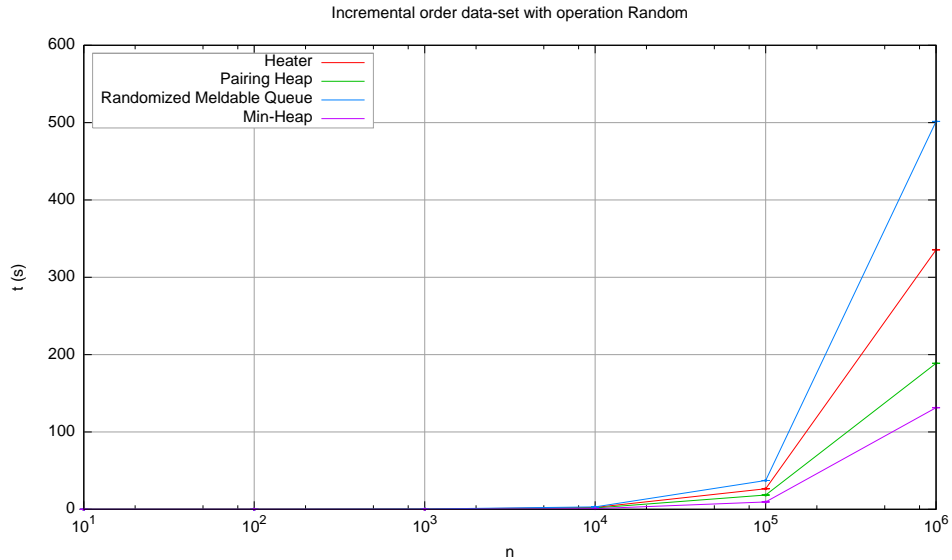


Figure 7: Randomly ordered insertion and removal of n incremental ordered elements

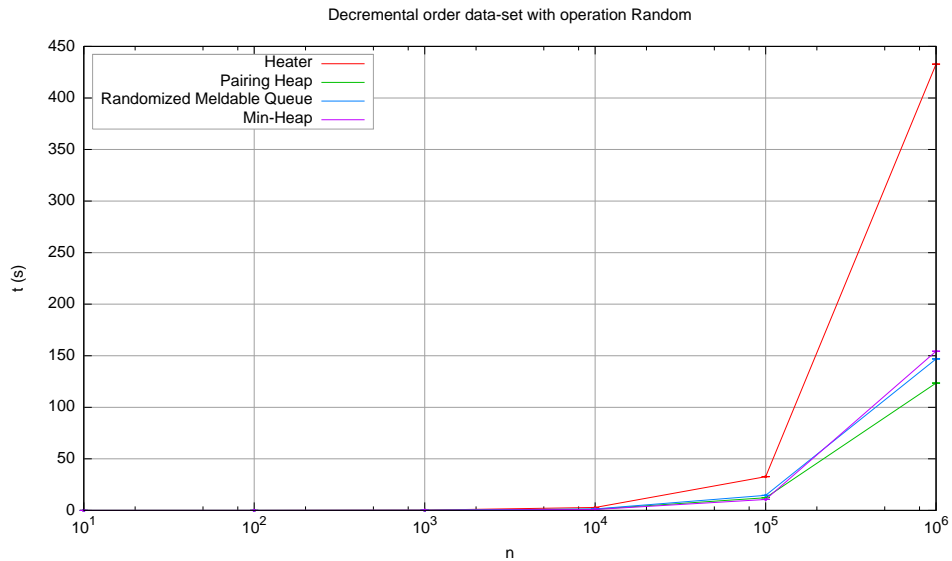


Figure 8: Randomly ordered insertion and removal of n decremental ordered elements

4 Conclusions

A randomized meldable queue was by far the easiest to implement and contains the fewest lines of code. It also provides decent experimental performance when operations consist of randomly chosen operations of insert and remove. It also requires little extra memory compared to a min-heap. A min-heap is both memory efficient and performs well in most circumstances and only suffered when data was inserted in a decreasing order of priorities. Although less memory efficient than a min-heap or randomized meldable queue, the

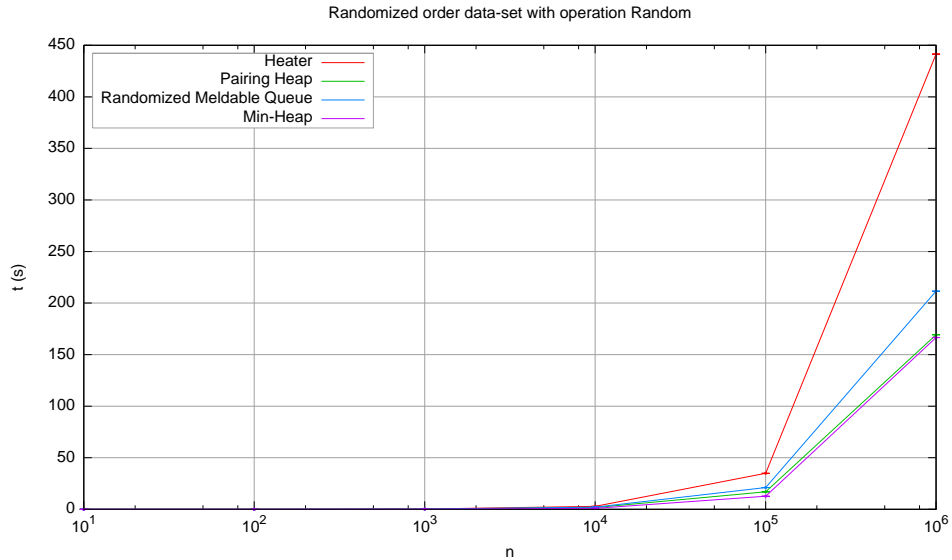


Figure 9: Randomly ordered insertion and removal of n random ordered elements

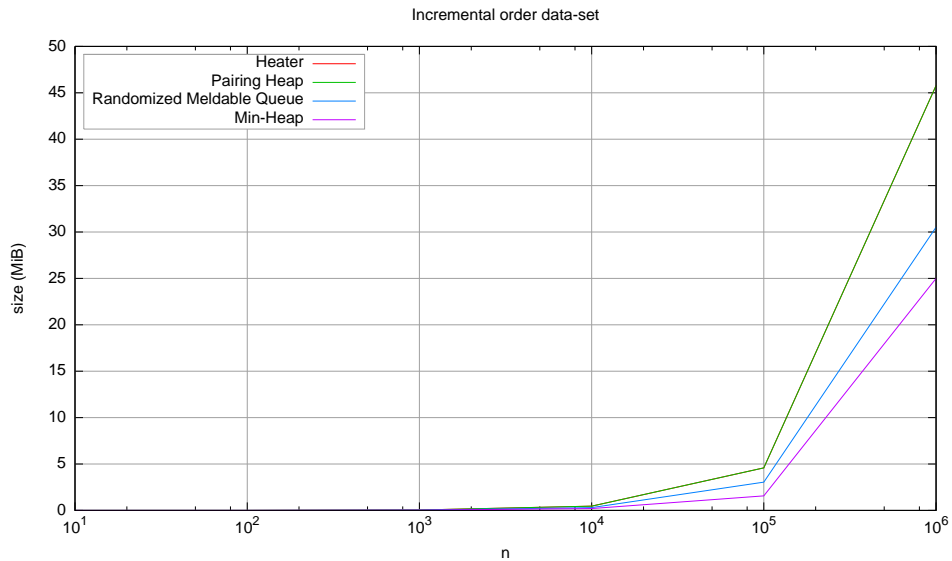


Figure 10: Memory usage when inserting incremental ordered data

pairing-heap has shown to have good performance in most situations often nearly matching or even exceeding the performance of a min-heap. A heater has shown to be the least efficient in terms of performance and equivalent to the pairing-heap in terms of memory usage which suggests that it is not a very good choice for a data-structure for priority queues.

The min-heap has an advantage over the tree based structures. It requires very few memory allocations (only when the heap is full). The tree based structures require a memory allocation for each insert performed. Likewise, when an element is removed from the

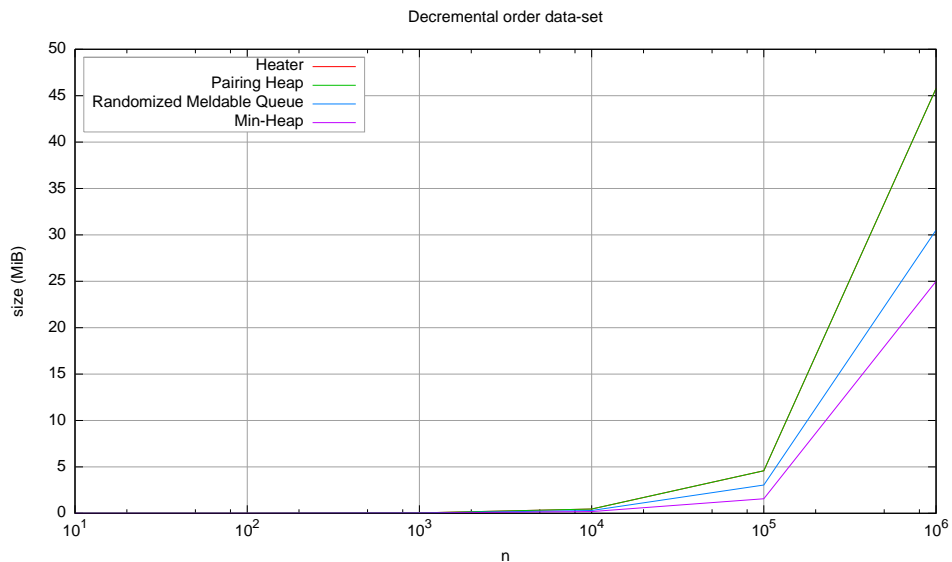


Figure 11: Memory usage when inserting decremental ordered data

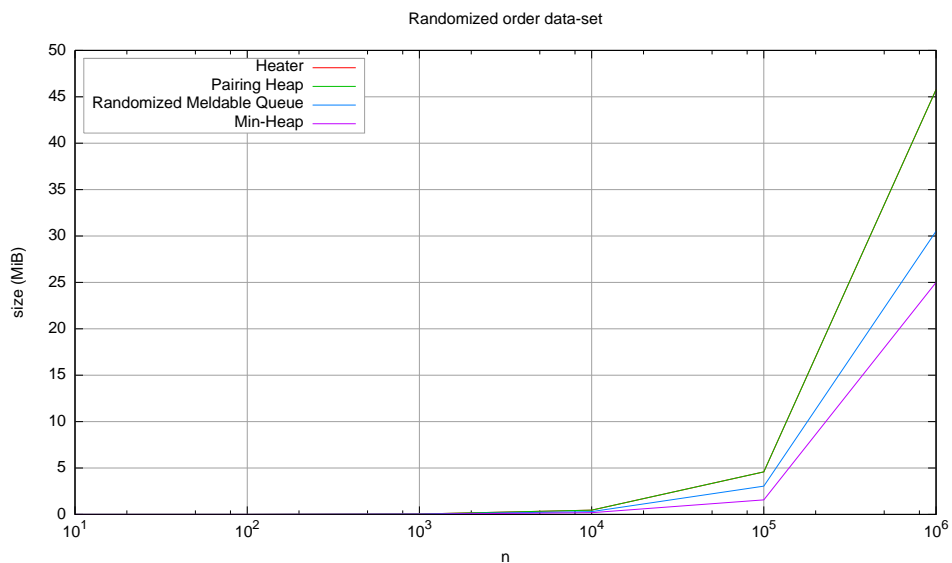


Figure 12: Memory usage when inserting random ordered data

tree structures they must make a *free* call in order to release the memory used. The tree based structures could benefit from customized memory allocation routines that use a memory pool in order to decrease the number of allocations and deallocations required.

The different implementations of priority queues can be more useful depending on the circumstances. For example, in a discrete event simulator decreasing order data will never occur since events will never be scheduled to occur in the past. But both insert and delete_min operations will be performed on the queue in a “random” order. Knowing this we can see,

from Figure 7 that a min-heap would likely have the best performance in this situation.

In general, a minimum binary heap seems to be the most obvious choice for a priority queue because of both its memory and time efficiency.

References

- [1] Prosenjit Bose, Luc Devroye, and Pat Morin. Topics in data structures (course notes). 2008.
- [2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2004.
- [3] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, March 1986.
- [4] Gambin and Malinowski. Randomized meldable priority queues. In *Theory and Practice of Informatics, Seminar on Current Trends in Theory and Practice of Informatics, LNCS*, volume 25. 1998.
- [5] Iacono. Improved upper bounds for pairing heaps. In *SWAT: Scandinavian Workshop on Algorithm Theory*, 2000.