Lab 10 Report
Performance Comparison of
Binary Heap, Leftist Heap and Skew Heap

Yan Xia
KUID: 2330720

April 21, 2013

Contents

1 Motivation **3**

2 Overall Organization of Experiment **3**

3 Data Generation **3**

4 Summary of Results **4**
4.1 Experiment Results 4
4.2 Average Calculation 5
4.3 Plots Generation 5

5 Observation and Conclusion **6**

1 Motivation

The goal of this lab is to implement and compare the performance of three priority queues. Specifically, min binary heap, min leftist heap and min skew heap were implemented, and timing experiments were then performed on these priority queues using inputs from random number generator.

The array-based implementatin of binary heap takes advantage of the complete tree structure property of binary heap and enables efficient performance of basic operations, such as insert and deleteMin. However, it seems difficult to implement the merge operation using array-based priority queues. To enable efficient merge operation, lefist heap and skew heap were developed, which use the pointer-based children pointer implementation.

2 Overall Organization of Experiment

1. Implementation: Binary heap, leftist heap and skew heap were implemented. Basic operations, such as buildHeap, insert and deleteMin, were tested by building these priority queues via either the heapify operation or insertion. Then, five consecutive deleteMin operations were performed to test its correctness. The priority queues after each deleteMin operation were printed, along with their level order traversals. These test ensures that we have the priority queues correctly implemented.

2. Timing on building the priority queues: For each type of priority queue, we attempted to build the priority queue with different sizes from 50000 to 400000. For each size, we built the heap with randomly generated numbers between 1 and $4 \times n$, where n is the size of priority queues. The same procedure was repeated 5 times using different numbers to seed the random number generator. Binary heap were built by the bottom-up heapify operation, while leftist heap and skew heap were built by insert operations. The time of building the priority queues were recorded by the provided Timer class and reported by the program. Then, the average running time of these 5 repetitions were calculated to generate a plot of average running times versus heap sizes for the three priority queue data structures. In each repetition, we used the same seed for all three priority queues to ensure that the same sequence of numbers were used.

3. Timing on insert/deleteMin: We then performed timing tests on the insert and deleteMin operations for the three types of priority queues. For priority queues of different sizes, insert/deleteMin operations were performed $n \times 10\%$ times, where n is the size of the priority queues. Insert and deleteMin were chosen randomly with equal probability. If insert operation was chosen, a random number between 1 and $4 \times n$ were generated and inserted into the priority queue. The same procedure were repeated for 5 times, using the same seeds from the process of building the priority queues. The times of insert/deleteMin operations were recorded and the average times were calculated the same way as in buiding test. Finally, the average insert/deleteMin times were ploted against heap sizes for the three priority queue data structures.

3 Data Generation

The execution time of building and insert/deleteMin with different sizes were outputed by the program for each of the three priority queue data structures. These data are reported in Table 2 and Table 3. The average of running times with different seeds were calculated and used to generate plots in Section 4. Machine information is summersized in Table 1.

OS	Mac OS X 10.8.3
Kernel Version	Darwin 12.3.0
CPU	Intel Core i7 2.3 GHz
Number of CPUs	4
Architecture	x86_64
Compiler(Version)	llvm-g++ (4.2)

Table 1: Machine Information

4 Summary of Results

4.1 Experiment Results

The results outputed by the program are listed in Table 2 and Table 3. We simply use repetition index (0 - 4) to seed the random number generator.

Size ($\times 10^4$)	Heap	Seed				
		0	1	2	3	4
5	Binary Heap	0.001	0.001	0.001	0.001	0.001
	Leftist Heap	0.014	0.012	0.011	0.012	0.012
	Skew Heap	0.010	0.009	0.009	0.009	0.009
10	Binary Heap	0.002	0.002	0.002	0.002	0.002
	Leftist Heap	0.025	0.025	0.025	0.025	0.025
	Skew Heap	0.020	0.020	0.020	0.021	0.020
20	Binary Heap	0.004	0.004	0.004	0.004	0.004
	Leftist Heap	0.052	0.052	0.051	0.053	0.053
	Skew Heap	0.044	0.045	0.042	0.045	0.045
40	Binary Heap	0.008	0.007	0.008	0.007	0.007
	Leftist Heap	0.108	0.108	0.107	0.113	0.112
	Skew Heap	0.099	0.105	0.096	0.106	0.103

Table 2: Summary of Building Times

Size ($\times 10^4$)	Heap	Seed				
		0	1	2	3	4
5	Binary Heap	0.001	0.001	0.001	0.001	0.001
	Leftist Heap	0.003	0.003	0.003	0.003	0.003
	Skew Heap	0.002	0.002	0.002	0.002	0.002
10	Binary Heap	0.001	0.001	0.001	0.001	0.001
	Leftist Heap	0.007	0.007	0.007	0.007	0.007
	Skew Heap	0.004	0.004	0.004	0.004	0.004
20	Binary Heap	0.003	0.003	0.003	0.003	0.003
	Leftist Heap	0.016	0.016	0.016	0.016	0.016
	Skew Heap	0.010	0.010	0.010	0.010	0.010
40	Binary Heap	0.006	0.006	0.006	0.006	0.006
	Leftist Heap	0.037	0.036	0.038	0.036	0.036
	Skew Heap	0.025	0.025	0.026	0.026	0.025

Table 3: Summary of insert/deleteMin Times

4.2 Average Calculation

For each type of priority queues with different sizes, the averages of execution times from five repetitions were calculated for both building and insert/deleteMin operations. These average values are summarized in Table 4.

Size ($\times 10^4$)	Binary Heap		Leftist Heap		Skew Heap	
	Buiding	Insert/DeleteMin	Buiding	Insert/DeleteMin	Buiding	Insert/DeleteMin
5	0.001	0.001	0.012	0.003	0.009	0.002
10	0.002	0.001	0.025	0.007	0.021	0.004
20	0.004	0.003	0.052	0.016	0.045	0.010
40	0.007	0.006	0.109	0.037	0.105	0.025

Table 4: Summary of Average Times

4.3 Plots Generation

Subsequently, we plotted average building times vs. heap sizes (Figure 1) and average insert/deleteMin times vs. heap sizes (Figure 2) for each type of priority queues. Plots are generated by Python matplotlib.

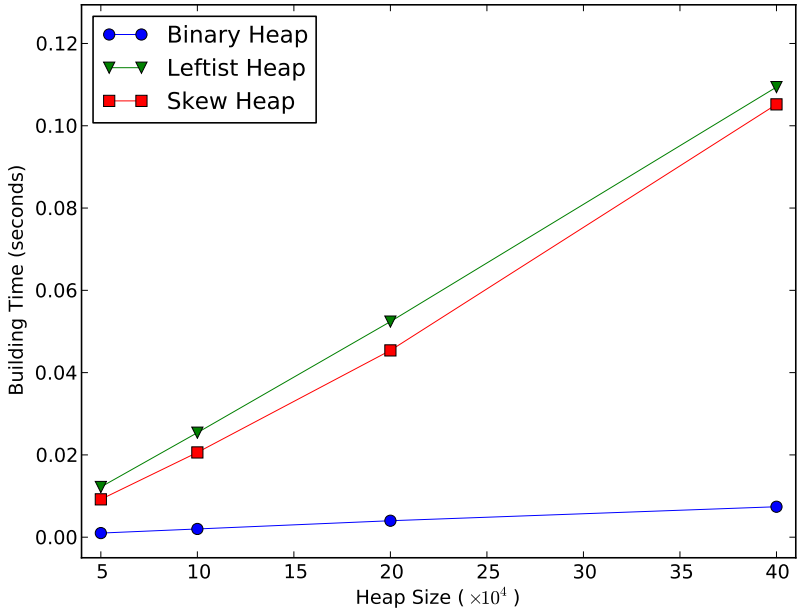

Figure 1: Average Building Time vs. Heap Size

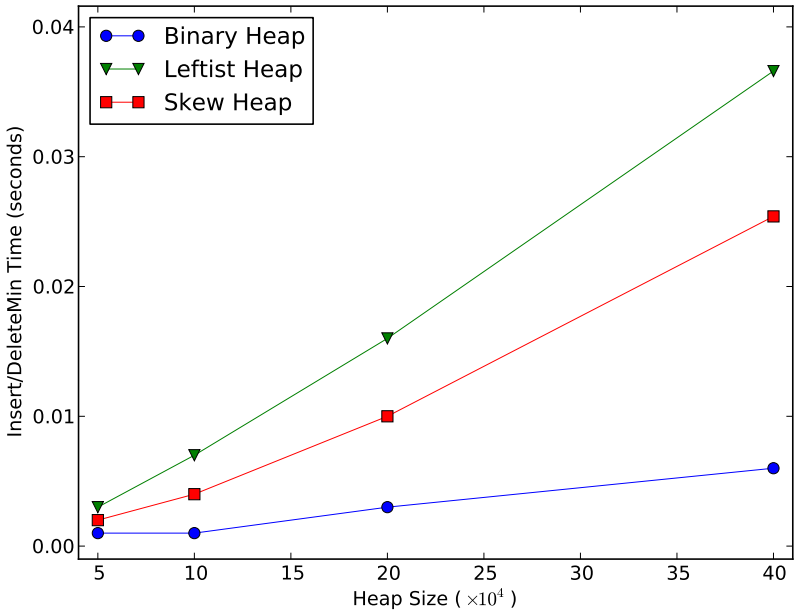

Figure 2: Average insert/deleteMin Time vs. Heap Size

5 Observation and Conclusion

In order to make sense of the performance differences shown in the Figuer 1 and Figure 2, we compiled a table summarizing the worst case time complexity of different operations of the three types of priority queues (Table 5).

Classical binary heap is a very efficient data structure for build, insert and deleteMin operations because of the complete tree structure property and the array-based implementation.

However, the array-based binary heap does support the merge operation efficiently. One method of merging two binary heaps requires the concatenation of two binary heaps and rebuilding the heap completely afterwards, and therefore have a worst case time complexity of $\mathrm{O}(n)$, which is unsatisfactory.

As a result, leftist heap was developed to support efficient merge operation with sub-linear complexity. This sub-linear complexity is achieved by introducing a leftist tree structure property, in which nodes keep track of their ranks in the tree and ensure that its left subtree always has higher rank than the right subtree. This property favors the formation of "left-heavy" binary trees, and by only merging at the right subtree, it can be proven that the merge operation has $T_w(n) = \mathrm{O}(\lg n)$. Build, insert and deleteMin operations in leftist heap can be reduced to the merge operation and has the complexity shown in the table.

A skew heap is a self-adjusting version of a leftist heap that is simple to implement. Skew heaps are binary trees with heap order, but there is no structural constraint on these trees. Unlike leftist heaps, no information is maintained about the rank the the nodes, and skew heaps always perform swapping after merging. Though the worst case running time of insert/deleteMin operations is $\mathrm{O}(n)$, it was shown that the amortized cost for M consecutive operatons is $\mathrm{O}(M \lg n)$, which corresponds to a $\mathrm{O}(\lg n)$ amortized cost per operation. In skew heap, no extra space is required to maintain rank informations and no tests are required to determine when to swap children, the simpler implementation generally leads to faster running time for random input sequences. The main difference between leftist heap and skew heap are summarized in Table 6.

Operations	Binary Heap	Leftist Heap	Skew Heap
Build	$O(n)$	$O(n \lg n)$	$O(n \lg n)^*$
Insert	$O(\lg n)$	$O(\lg n)$	$O(\lg n)^*$
DeleteMin	$O(\lg n)$	$O(\lg n)$	$O(\lg n)^*$
Merge	$O(n)$	$O(\lg n)$	$O(\lg n)^*$

* Amortized cost per operation shown

Table 5: Worst Time Complexity of Operations

	Leftist Heap	Skew Heap
Tree Structure	lefist tree	binary tree
Implementation	children pointer	children pointer
Node Structure	need info on rank	no info on rank
Merging	verify rank info	no rank info
Re-attachment	may swap children	always swap children

Table 6: Comparision between Leftist Heap and Skew Heap

The data structure analysis above correlates very well with the experimental observations. As shown in Figure 1, in the building process binary heap out-performs both leftist heap and skew heap, due to both the efficient bottom-up heapify procedure ($O(n)$) and the array-based implementation. On the other hand, the performance of leftist heap and skew heap are quite close. Skew heap performed slightly faster than the leftist heap, potentially because of the simpler implementation of the insert operation.

Figure 2 demonstrates the performance of insert/deleteMin operations of three types of priority queues. Though the worst case complexity of all three priority queues is $O(\lg n)$, the array-based implementation of binary heap still makes it considerably faster than the pointer-based leftist heap and skew heap. The gap between leftist heap and skew heap becomes more evident. A potential reason is that only insert operations are used in building the priority queues, while both insert and deleteMin operations were perfromed in the second test. The deleteMin operation merges two large subtrees after deleting the root, and the merging process requires more numbers of computing ranks than the insert operation, in which we merge a one-node tree to a large tree. The increased number of computing rank operations further deteriorates the performance of leftist heap.

To further investigate the source of performance differences between leftist heap and skew heap, we profiled the program using GNU gprof(version 2.22.52.0.1). The profiling results showed in Table 7. From the table, it is obvious that the slower performance of leftist heap was caused by the extra 1.29 seconds of computing rank operations.

	Leftist Heap	Skew Heap
Merge	1.32 s	1.78 s
Compute Rank	1.29 s	
Total	2.61 s	1.78 s

Table 7: Profiling of Lefist Heap and Skew Heap

In conclusion, in this lab we implemented three types of priority queues and attempt to compare their performances on building the priority queue and insert/deleteMin operations. Binary heap performed best in both tests because of its array-based implementation and the complete tree structure property. Both leftist heap and skew heap were developed to support efficient merging. Because of the lack of balanced structural property and pointer-based implementation, their insert/deleteMin operations are slower than the binary

heap, even though the time complexity are the same ($\mathrm{O}(\lg n)$). In theory, leftist heap should perform better than skew heap because of the leftist tree property. The rank of nodes that the leftist tree property maintains provides a tigher control of the topology of the heap, and it can be proven that the worst case running time for insert/deleteMin is $\mathrm{O}(\lg n)$ for leftist heap by only performing operations on the right subtree. Skew heap introduces randomness into the algorithm by always swapping the children after merging, and its worst case running time could be as high as $\mathrm{O}(n)$. It is conceivable that for certain input sequences leftist heap will perform better than skew heap because the performance guarantee provided by the leftist tree property. However, it can be proven that the amortized cost per operation is actually $\mathrm{O}(\lg n)$ for skew heap assuming random inputs, and because its simpler implementation, skew heap normally out-performs leftist heap in practice.