

EECS 560 Project 2 Report

Analysis of Priority Queues:

Binary Search Tree

Min-Max Heap

Leftist Heap

Lance Feagan

April 8, 2003

Computing Environment

The computing environment used was a Sun Blade 100 Workstation with a 500 MHz UltraSPARC III processor with 256-KB L2 cache and 1.7GB of PC133 ECC SDRAM. The compilers used were Sun Workshop 6 Update 2 C++ 5.3 2001/05/15 and Sun C++ 5.5 EA2 2003/01/09. The first compiler is a known stable compiler and the second compiler is Early Access (Beta) version 2 of the newest Sun compiler for C++. I am using the standard libraries for the respective compilers. The operating system is Sun Solaris 9 2/03. The Solaris Management Console (SMC) is setup to update the system with all current patches on a weekly basis for all installed programs.

Overall Program Organization

The program is organized into a main program and three template classes. The main program is used to execute the loop that reads the file for input in the first stage and to load the three priority queue (PQ) data structures with random data in the second stage of execution. The three template classes are used for the binary search tree, leftist heap, and min-max heap.

In addition, there are some helper classes that I have created. They are ReadFile, which is used to ease the reading from the data file input.txt. Dexceptions is used for error handling.

Random Number Generation

For random number generation I used the stdlib.h function lrand48 and casted it back to an integer. Rand() does not provide the necessary range, as RAND_MAX is typically 32,768 on most machines. Lrand48 on the other hand has a range of $2^{31}-1$ which is more than suitable for my needs as I only need numbers up to $3 \cdot 2^{20} < 2^{22} < 2^{31}$.

A special class, RandomNumbers, was created to ease the creation of random numbers between 1 and $3 \cdot n$ for this project. It will take in a lower and upper bound when instantiated. Then, you can call a function and pass it a size and an array pointer and it will fill the array with the requested number of elements on the range given upon object creation.

DeleteMax Algorithm

The DeleteMax algorithm for the leftist heap is very simple. First, it does a scan through the whole heap structure to find the largest element. As it is searching it not only keeps track of the largest element, but also a pointer to its parent and a pointer to the largest element. Once the tree has been completely scanned, the largest element is removed and the other child of the parent is merged with the parent, if there is another child. Otherwise, the largest element is merely removed. This situation has a complexity of $n + \log(n)$ in the worst case.

Tables / Graphs

All tables and graphs are contained at the end of this document. The ones with no specific labels are for comparisons and the ones labeled "Timing" are for CPU timing.

Data Analysis

For all of the plots I decided to use a Log-Log scale as this greatly eases analysis. One of the nicest aspects of this type of plot is that you can easily determine the order of the highest-power term in the equation governing the complexity for each of the different data structures for a particular operation. Often times this is not easily seen when other types of plots are used. In addition, it helps to condense the plots so that the large range involved does not cause the plots to become so excessively skewed towards the large values and so minimizing of the small ones as to become useless.

Build: The build operation for all three types of data structures exhibited a linear nature when scaled versus n , the number of elements. Theoretical values for the worst-case complexities for a leftist heap are $2n$ and for a Min-Max heap are $7n/3$. I observed a similar behavior for the two. The Min-Max heap did in general slightly more comparisons than the leftist heap did.

DeleteMin: The data for DeleteMin

DeleteMin is an example of what all three of these data structures certainly excel at. Unlike DeleteMax, which is only a forte of the Min-Max Heap and the BST to a lesser extent. All three data structures seemed to exhibit a slightly less than linear behavior. In fact, they should scale according to $2\log(n)$ for the leftist heap and $2.5\log(n)$ for the Min-Max heap. All three functions clearly exhibited the $O(\log(n))$ behavior that was expected. However, I was a bit surprised to see that the Min-Max heap had the lowest number of comparisons for each DeleteMin. I would conclude that this is a result of the maintenance level being higher on BSTs and Leftist heaps.

DeleteMax:

This function executes in a superior fashion on the Min-Max Heap, constant time, and in a very respectable fashion on a BST, in logarithmic time. Unfortunately, on a Leftist heap this operation is $O(n + \log(n))$. This is definitely not an acceptable situation if you need to perform this operation very much. The graph clearly shows that the Min-Max Heap is almost horizontal, indicating constant execution time. The BST has a slight curve, of slope approximately 0.3, which is fitting with a logarithmic increase in complexity. This makes sense since the BST's complexity is based on the number of levels that need to be traveled down, which also scales in a logarithmic manner. The leftist heap, as is obvious, is scaling with a slope of 1 plus a little bit more, which makes sense for an $O(n + \log(n))$ type relationship.

Insert: The insert function also displays very similar characteristics to what one would expect. The Leftist and Min-Max heaps are both very quick at completing the insert operation. However, the BST is not nearly as effective at completing this task in a small amount of time. It is clearly from the curvature of the graph though that the binary search trees number of comparisons is directly related to the height of the tree. That is why on a Log-Log graph it has a curve that looks like $1-e^{-x}$.

Conclusion

This was a very interesting project. I would like to whole-heartedly thank the grader for helping us get an extension on this project as I am much happier with the thorough analysis I was able to perform. I would have been saddened had I not had more time as I really would not have learned nearly as much.

The three different priority queue structures analyzed here are all very interesting. Based on the data obtained, I would say that a Min-Max heap is certainly the best implementation possible if you know very little about the requests that are going to be made of the data structure, other than knowing that you do not need a general find. A Min-Max heap, as well as a leftist heap, would not make for particularly good data structures toward this type of operation. The leftist heap was by far the weakest data structure in my mind. In particular, the slow execution of the DeleteMax operation, showed the largest weakness of this structure. The BST was a very respectable data structure in comparison with the Min-Max heap other than one thing. The thing in my mind that makes the Min-Max heap the strongest data structure is because it can be constructed in linear time and can be stored in an n -element array. These two properties combined make it an implicit data structure. Neither of the other two data structures are implicit data structures.

Optional Timing Analysis

For the timing analysis, I was lucky enough to have Sun Workshop as my IDE of choice. It, in fact, will do all of the timing analysis for me. To accomplish the timing analysis, I just adjusted the runs so that only one value of k would be executed in each run of the program. This allowed me to use the Workshop Analyzer to see how long Workshop took to execute each function. I continued to use the average of 100 runs. However, I have reported the sum of all 100 runs and used that for my data gathering as the numbers are much more human convenient. It should be noted that I did not do any timing analysis for k greater than 15 as this resulted in extravagant times to complete. Remember, I ran 100 executions at each k , so, this is to be expected. However, I feel that the number of k values attempted does enable us to see the relationship between comparisons and CPU clock timing.

Build: The build operation for all three data structures showed a nearly linear build time. This is evidenced by the Log-Log Scale graph in which the slope is very nearly 1. However, further analysis with a non-Logarithmic scale graph shows a slightly different picture. On it we can see that the BST is clearly having a slightly upward curvature. The Min-Max heap is clearly the fastest at building, especially with the bottom-up algorithm. The leftist heap is right in-between the two as far as performance.

DeleteMin: Once again, a linear relationship based on the Log-Log graph, which is certainly not as the comparisons shows. The comparisons vs. n shows an $\log(n)$ type relationship. This, however, is not surprising, as a real computer has many checks to do to complete each of the algorithms even though not many comparisons may be done. It is also interesting to note that the ordering is reversed, i.e., the Min-Max heap has the fewest number of comparisons, but it is the slowest implementation. However, a closer look at the Log-Log plot shows that the other two heaps are likely have a second derivative that is increasing, whereas it looks as though the second derivative for the

Min-Max heap is decreasing. Perhaps for larger k we would in fact see the Min-Max heap become faster.

DeleteMax: The DeleteMax operation is clearly the dominant function as far as timing goes on the Min-Max heap. However, it should be noted that the functions all seem to have fairly similar slopes on the Log-Log graph, despite the fact that the non-Log-Log scaled graph seems to make it appear as though DelMax for the Leftist heap is taking off like a banshee, which it is. This is, in fact, more fitting with what we would expect since the DelMax function should execute in $O(n+\log(n))$ time for a Leftist heap.

Insert: On the insert operation, once again the Min-Max heap is the clearcut winner as far as performance.

Optional Timing Analysis Conclusions

The timing analysis certainly showed a number of facts clearly.

- 1) Array implementations are incredibly faster than pointer based. This is evidence by the Min-Max heap, which uses an array.
- 2) Some of the operations, such as DeleteMax and DeleteMin, which should execute in very favorable times according to the number of comparisons method for a Min-Max heap, do not execute as well when analyzed more carefully with timing.

From these two facts, it is easy to see that when a practical implementation is going to be done, timing analysis should certainly be done. The number of comparisons clearly does not work well for array based implementations. However, for pointer based implementation it does work well. Why could this be? My suspicion is that the array based implementation are able to better take advantage of the cache on the processor since the array is static and to load it into on-die cache. The pointer based implementations take up much larger data structures for the same sized input set. This means that the computer's processor will spend much more time in a wait state for data to come in from main memory.

This waiting for data to come in from main memory is a very realistic concern. This drive to fit all data within a L3 or better yet L2 cache is a major factor in the high-end server arena and supercomputer arena. The ability to fit an 8MB data set into very rapid cache memory can greatly enhance the performance of many algorithms. The timing analysis also allows the study of other variables effecting the execution times. It allows you to compare different architectures and configurations of machines to one another to see how efficiently they are able to execute instructions. This would make a very interesting analysis to perform with other students in the class.

My conclusion: I would not use the comparisons method for any type of timing analysis. It is inconvenient and is not effective at analyzing algorithms in my opinion.