Perry Gowdy
EECS 560
Lab 8
11/02/14

Lab 8 Leftist Heap vs. Skew Heap

The purpose of the experiment in lab eight was to compare the execution times of two different heap structures and their execution times. The skew heap was the first type of heap that we implemented, and it utilizes a merge function that recursively merges the original tree and a subtree. The other heap that we used was a leftist heap. In order to qualify as a leftist heap, the heap must always be either the null set or for every node that is apart of the heap, the rank of the left child must be greater than or equal to the rank of the right child. The heaps that we are to compare are very similar in most ways. Both insertion methods of the heaps require you to insert the method and then simply call the merge function. Deleting the minimum in both of the heaps is basically the same as well. Traversing the heaps is, also, of course, the same.

In my specific implementation, I created a general folder labeled Gowdy_Lab 8that had within it three sub-folders. The three folders were named LeftistHeap, SkewHeap, and Timing. This division in the experiment allowed to me to compile and execute the two methods individually and create a general heap program for each that had a main menu and could read from a data file. In the timing folder, I would run each program and, using a Timer class that was given to us in lab, output the amount of time each program took to complete. The main.cpp in the Timing folder had code that allowed me to generate all the results that was necessary to complete the lab at one time.

Data generation for the hash table was created using the random number generator that is apart of the standard C++ library. Using values between 0 and 4n, where n was the number of values that were generated, I generated a set of values for five different seeds. The quantity of numbers that was to be inserted into the table, or the number that n was equal to, was 50,000, 100,000, 200,000, and 400,000. These numbers were generated and then loaded into an array so that they could be used in all of the data sets and so that the heaps had consistent testing rather than being executed on a different data set each time.

The experiment was able to be further categorized by using different seed values for the the various random numbers. Because a computer is not truly capable of creating a random number, the srand() constructor takes in a seed value as a parameter and utilizes it in the algorithm that will generate a random number, the rand() function. I used a total of five different seed values across the four different load factors. The seeds values were 1, 2, 3, 4, and 5. Using the same seed value guaranteed a fair comparison of the heap implementations as the number used in the random number generation would be the same. By using a total of five different seeds, I generated in total eighty different values overall for the experiment or twenty different values for each number that n was equal to.

In addition to simply testing how quickly each heap could insert a certain number of random values into a table, tests on how quickly random insertion and delete minimum deletions at 10% of the size of the heap were also done. Generation of a random number between 0 and 1 was executed and if the number was between 0 and .5, the data structure performed a delete minimum operation. If the value generated was between .5 and 1, then a random integer was generated between 1 and 4n and then inserted into the heap. This allowed for another metric to judge the efficiency of the data structure.

The execution times of the two implementations were very close and there seemed to be only marginal difference between the two different implementations. By taking the average of each of the two methods I came up with the following results:

Perry Gowdy
EECS 560
Lab 8
11/02/14

	Seed	1	2	3	4	5	Average	
Number of Operations: 50,000								
Skew		0.016497	0.0136	0.012257	0.0105974	0.010013	0.0125922	
Leftist		0.00908	0.007677	0.007169	0.006199	0.00618	0.007261	Random Numbers Between 1 – 200,000
Skew Operations		0.002531	0.002149	0.001865	0.00172	0.001748	0.0019068	
Leftist Operations		0.002382	0.002101	0.001756	0.001626	0.001669	0.0020026	

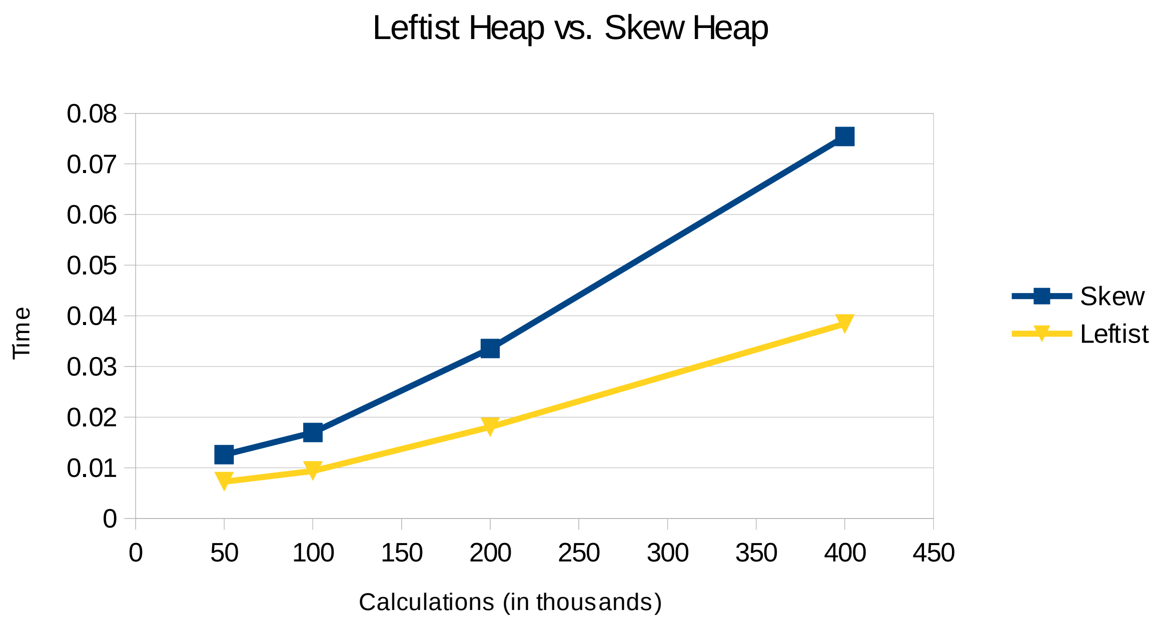
	Seed	1	2	3	4	5	Average	
Number of Operations: 100,000								
Skew		0.019999	0.018341	0.016489	0.014829	0.015111	0.0169538	
Leftist		0.01087	0.009935	0.008772	0.008602	0.008673	0.0093704	Random Numbers Between 1 – 400,000
Skew Operations		0.005157	0.004509	0.004093	0.004014	0.004084	0.0041532	
Leftist Operations		0.004568	0.004142	0.003934	0.004082	0.00404	0.0043714	

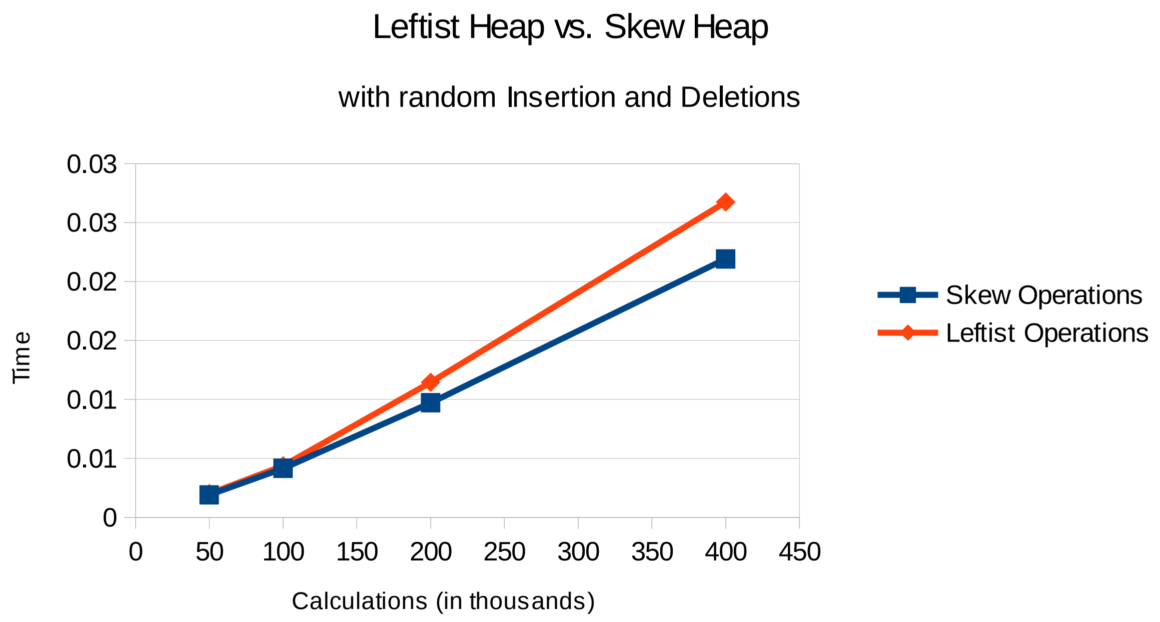
	Seed	1	2	3	4	5	Average	
Number of Operations: 200,000								
Skew		0.032247	0.034336	0.032811	0.033164	0.035204	0.0335524	
Leftist		0.017319	0.018495	0.017572	0.018138	0.018526	0.01801	Random Numbers Between 1 – 800,000
Skew Operations		0.011326	0.011327	0.011576	0.011371	0.01166	0.0097156	
Leftist Operations		0.009544	0.00987	0.009739	0.00969	0.009735	0.011452	

	Seed	1	2	3	4	5	Average	
Number of Operations: 400,000								
Skew		0.075853	0.078027	0.0743	0.077684	0.07112	0.0753968	
Leftist		0.038034	0.038796	0.038026	0.040276	0.036908	0.038408	Random Numbers Between 1 – 1,600,000
Skew Operations		0.026635	0.026643	0.026781	0.026898	0.026761	0.0219162	
Leftist Operations		0.021677	0.021907	0.021878	0.022176	0.021943	0.0267436	

Perhaps the most useful column is the column in which the average times are computed. Here we can see that the skew heap took longer, on average, to perform n number of insertions than did the leftist heap. However, when we compare how long the random operations (insert and delete minimum) took, we see that the Leftist heap took longer, on average to perform these tasks. The graphs below give a more quick and clear picture of exactly these results:

Perry Gowdy
EECS 560
Lab 8
11/02/14





As we can see in the graphs and the raw data itself, the two implementations have advantages depending on what operations are being performed and how many elements they are being performed

Perry Gowdy
EECS 560
Lab 8
11/02/14

on. In terms of simple construction and raw insertion power, the leftist heap seems to have an advantage on the skew heap. This advantage grows to become even more prevalent as the number of elements reaches the 400,000 mark. Here we can see a difference of around .035 seconds. However, in when performing random numbers of deletions and insertions, it seems that the skew heap has the advantage over the leftist heap and has a much lower execution time. This difference in execution time is less prevalent than in the simple insertion test and we see that even at the largest number of elements 10% of 400,000, we find a difference of a measly .005. Hardly a big difference in execution time.

One can conclude that, overall, the leftist heap is a more efficient data structure when it comes to the insertion of a large number of elements. However, in a situation in which random operations were performed, the skew heap would barely beat out the leftist heap. The leftist heap is a safer bet, overall.