IE 170 Laboratory 4: Sorting

Dr. T.K. Ralphs and Dr. J.T. Linderoth

Due Feb 20, 2006

1 Laboratory Description and Procedures

1.1 Learning Objectives

You should be able to do the following after completing this laboratory.

1. Understand heaps and the heap property.
2. Understand how to implement a heap with an array.
3. Understand the heapsort algorithm.
4. Gauge the impact of initial ordering on the effectiveness of different algorithms.
5. Understand the relationship of merging and sorting.
6. Understand how to select the sorting algorithm most appropriate for a given situation.

1.2 Key Words

You should be able to define the following key words after completing this laboratory.

1. Heap
2. Heap sort
3. Insertion sort

1.3 Scenario

MegaStore.com is a giant conglomerate that is in the business of acquiring companies. It then incorporates the acquired company's catalog of products into its own. **MegaStore.com** has acquired the company **ExoticBeer.com** (from Laboratory #1). Thankfully, you have been retained by **MegaStore.com** as their Chief Algorithm Officer (CAO). As you learned in Laboratory 1, searching for a specified item (or word) in a dictionary can be done quickly if the list of items is sorted. However, not all of the companies that **MegaStore.com** acquires have nicely sorted catalogs. In fact, during one hostile takeover, a company even scrambled their own catalog data out of spite. Therefore, you will need to develop a sorting algorithm that can be used for sorting key words and/or merging various lists of keywords together.

1.4 Design and Analysis

In this lab, we will reinforce two basic concepts that we have emphasized before. The first principle is that of choosing the proper algorithm for a given situation. We will compare two algorithms for sorting, one of which is efficient when the list is almost completely sorted, and one of which is efficient for randomly ordered lists.

Second, we will continue to emphasize the basic design principle of implementing a class based on a given interface, rather than on a client. Therefore, your main assignment involves simply implementing a `heap` class with the public interface given in the file `heap.hpp`. You will then write a client program that uses this class to perform the functions listed below.

1.5 Program Specifications

1.5.1 Program(s) Function

Your program should perform the following functions:

1. Read in a list of key words from a file.
2. Sort the items in the file either using either insertion sort or heap sort, according to a command-line argument.
3. Write the sorted items to a different file.
4. Print the time necessary to perform the sorting operation.

1.5.2 Algorithms

The basic heap sort algorithm was described in Lecture 7. You will implement an array-based `heap` class with the public interface described in the file `heap.hpp`. For comparison, an implementation of insertion sort has also been provided for you in the file `heap.cpp`.

1.5.3 Data Structures

The basic data structure required for this laboratory is a heap.

2 Laboratory Test Files

The test files for this laboratory are in the zip archive `Lab4.zip` available on the course Web site. The archive will unpack into a directory called `Lab4` with subdirectories `data`, and `shell`. The shell directory contains the files `heap.hpp` and `heap.cpp` that can be used to create your program. In the `data` directory are three very large files containing keywords to be sorted. The file `input.sorted.txt` contains a list of words that is *almost* completely sorted. The file `input.jumbled.txt` contains the same list of words in random order. Finally, the file `input.merge.txt` contains two sorted lists concatenated together, one of which is much smaller than the other. These files will be used for testing and analysis.

3 Laboratory Assignments

3.1 Programming (50 points)

1. Implement a heap class supporting the given interface.
2. Write a client program that uses your heap class and behaves as specified above.
3. Verify that your program compiles and runs correctly using the test files provided for the lab.

3.2 Analysis (30 points)

1. (10 points) This question will compare heap sort and insertion sort. To answer this question, you should use the file `input.jumbled.txt`. To simulate input files of different sizes, you can simply change the first line in the file to be n (some number less than the actual number of lines in the file). That way, your subroutine for reading in the file will stop after reading only n words instead of reading the whole file. Run both insertion sort and heap sort for "simulated" files of size 100, 1000, and 10,000. Create graphs that show the growth in running time as a function of file size. How well do the graphs conform to the expected growth in running times based on worst case?
2. (10 points) Predict the running times that would be expected for each algorithm running on a list of size 100,000. Compare your prediction to the actual running time for heap sort (insertion sort will probably take too long to verify its running time).
3. (5 points) Compare heap sort to insertion sort on the file `input.sorted.txt`, which is "almost" sorted. Report your running times. What did you find?
4. (5 points) Now compare insertion sort to heap sort for the file `input.merge.txt`, testing different sizes for the smaller list to be merged into the bigger list as follows. The big list has 929061 entries in it. Setting the first line of the file to $929061 + k$ effectively causes a list of size k to be merged into a list of size 929061. Find the value k at which heap sort becomes more efficient than insertion sort. Report on your experiments.

3.3 Follow-up Questions (20 points)

1. (5 points) 6.1-4
2. (5 points) 6.1-6
3. (10 points) 7.4-5 (just give a rough argument for why the *best case* running time is in $\Theta(nk + n \lg(n/k)$.