

CS 14: Data Structures and Algorithms

Lab 8: Advanced Sorting

Peter H. Fröhlich
phf@cs.ucr.edu

Wagner Truppel
wagner@cs.ucr.edu

Out on: February 24, 2003

Due by: March 2, 2003

Instructions: Remember to follow the instructions and policies for assignments given on the course website. This week's assignment consists of **three** separate parts:

- an **in lab** programming exercise (1)
- an **at home** programming exercise (2)
- an **at home** written exercise (3)

The “in lab” part is due at the **end** of your **lab section**. The “at home” parts are due **before 8:00 pm** on the date given above. You have to use the department's electronic turnin service for **everything**, including the **written** part. Put your solution for the latter into a **text file** `solution-8.txt` and turn it in with the other files.

1 In-Lab: Simple Heap Sort

As described in the lecture, we can use the heap data structure from Lab 7 to sort an array:

1. First we go through the array and `add()` each element to the heap.
2. Then we “fill” the array by repeatedly getting the minimum element using `min()`, putting it into the “correct” slot, and removing from the heap using `rem_min()`.

There are n elements in the array, and each `add()` takes $O(\log n)$ time, so the complexity of the first step is $O(n \log n)$. In the second step, each `min()` takes $O(1)$ time, each `rem_min()` takes $O(\log n)$ time, and again there are n elements. So the complexity of the second step is also $O(n \log n)$, making the **overall** complexity of this sorting algorithm $O(n \log n)$.

Your task for this lab is to extend your file `sorting` from Lab 4 with the following function:

```
heap_simple( cs14::Array<T> &a );
```

The function should implement the simple heap sort algorithm described above. While this algorithm is **asymptotically** faster than all the algorithms we discussed in Lab 4, it is **not an in situ** algorithm (the elements are copied into an auxiliary data structure). Due to memory allocations and other “problems,” the constant factors are also much higher than we would want.

2 At-Home: Heap and Quick Sort

Following the lab exercise, your task for this week is to further extend the `sorting` file with the following two functions:

```
heap( cs14::Array<T> &a )
quick( cs14::Array<T> &a )
```

The functions should implement the **in situ** sorting algorithms **heap sort** and **quick sort** as discussed in the lecture and the assigned books.

For heap sort, this means building the heap data structure **inside** the array; in this case it is easier to build a **max heap** and “fill the array” from the end. For both algorithms, you should **decompose** them further into separate functions (e.g. to “sift down” an element for heap sort, to “partition” the array for quick sort); since C++ does not support “nested functions,” we suggest you follow the convention of starting their names with an underscore.

3 At-Home: Performance

Now you will (once again) investigate the **actual** performance of sorting algorithms, both from Lab 4 and from this lab.

You should adapt the “timing programs” from Lab 4 (i.e. `timebubble.cpp`, `timeselection.cpp`, and `timeinsertion.cpp`) to perform the timing of heap and quick sort. Call your programs `timeheap.cpp` and `timequick.cpp`.

As in Lab 4, you will use the Unix `time` command to measure how long it takes each algorithm to perform all the sorting operations. You will also compile the programs using different levels of optimization again to judge their effectiveness (the levels are `-O0`, `-O1`, `-O2`, and `-O3`).

For your written answer, you should include the performance data you collected (each in the form of a table), compare the actual performance of these algorithms to their asymptotic complexity, and explain your “theories” (if you have any) on why certain implementations are faster or slower than others.