CMPS 101 - Programming assignment 2 - S13
Comparison of sorting algorithms (version 1)
Handed out 04-24-13 Due 11:55pm Thursday 05-02-13.

April 24, 2013

1 Introduction

Sorting is a well investigated problem in Computer Science and many different algorithms have been developed for accomplishing this task. There are simple algorithms with runtimes in $O(n^2)$ and specialized algorithms which make restrictive assumptions on data to obtain $O(n)$ runtime performance. In this assignment, you will implement and compare the performance of two common sorting algorithms.

For this assignment you may optionally work in groups of two. Partnerships must rotate – you may not use the same partner as a previous assignment (i.e. written 2). If you work in a group of 2, both student's are responsible for ensuring that both partners completely understand the code and C-constructs used. Every file should begin with a block comment indicating who the partners are (names and UCSC accounts) and any outside help received on that part of the program. Significant outside help must also be acknowledged in the README file. Only one partner should submit the zip file described below, the other partner should watch the submission to ensure it is done correctly and on time. The partner not submitting the program should instead just submit a copy of the partnership's README file (*not* zip'ed).

The three goals of this assignment are:

- Use the heap data structure to implement a priority queue and heap sort in the C language.
- Compare the running times of these algorithms.
- Improve your C programming and ADT skills.

2 Assignment Description

Your are to implement heap sort (with a priority queue ADT) and insertion sort. These method should sort a group of integer keys stored in an array (there may be duplicates). You will then need to test your algorithms to make sure they work correctly. Finally you will need to run experiments to compare the performance of these algorithms.

Most applications would have other information associated with the keys that is of interest. For example, one might want to sort a bunch of name-salary pairs by salary to see which names earn the top salaries. However, *for this assignment* we will just be sorting the keys (salaries).

Details on implementation are as follows.

- File `insertionSort.h` file should contain the following prototype: `void insertionSort(int keys[], int numKeys);` that is implemented in file `insertionSort.c`. This function should sort the `numKeys` integers in the array `keys`.
  One way to do this is with the array-based insertion sort algorithm in the text.
  A different way is to use your linked list ADT and insert the keys in the array one-at-a-time into a sorted linked

list. Then copy over the sequence of keys (e.g. by repeatedly putting the first element on the list into the next slot in the array, and then deleting the first element). Of course, if `insertionSort.c` uses your linked list module, it will include `list.h` and you must submit `list.h` and `list.c` with this program. Note that regular C passes arrays by address, so changes made to the array will be seen by the caller.

- Your `heap.c` and `heap.h` files should implement the heap ADT. The following description is for a max-heap, you may implement either a max-heap or a min-heap. Your `heap.h` file should declare the handle type `heap` (probably a pointer to a struct[1] containing an interger max-size, an integer current-size, and a pointer to an array of integers) and the prototypes
  - `heap newHeap (int maxSize);`
  - `int isFull(heap h);`[2],
  - `int isEmpty(heap h);`[3],
  - `int maxValue(heap h);`
  - `void deleteMax(heap h);`
  - `void insert(heap h, int priority);`

  and optionally `heap buildHeap (int maxSize, int[] data, int numData);`.

  Note that both `newHeap` and `buildHeap` also create new heaps, but the heap created by `newHeap` starts empty while the heap created by `buildHeap` it is initialized to contain the `numData` priority values stored in the `data` array. These functions should all be implemented in your `heap.c` file. As you implement these functions, create and extend a `heapDriver.c` program to test them out.

- Your `heapSort.h` file should contain the prototype `void heapSort(int keys[], int numKeys);` that is implemented in file `heapSort.c`. As with your insertion sort routine, the array `keys` is used both to pass in the unsorted keys and to pass the keys back in sorted order (i.e. the caller will see that the keys in their array are now in sorted order).

- `sortPrint.c` will contain a main program to test your sorting algorithms. In a sense, it is the driver program for the insertionSort and heapSort modules. `sortPrint.c` should take a file name as a command line argument. This file should have one integer per line. The first line indicates how many keys are in the file, and each other line contains a single key. `sortPrint` should read the file and create one copy of the keys for each sorting algorithm (stored into an array for that algorithm). For each sorting algorithm, `sortPrint` should call the algorithm on the appropriate array to get the keys in sorted order. `sortPrint` should verify that the keys are sorted (by comparing each $A[i]$ with $A[i+1]$) and print each sorting of the keys (labeled by sorting algorithm) to the standard output.

- `sortComp.c` contains a second main program which will be used to compare runtime performance. Like `sortPrint.c` this program should take an input file name as a command line argument and create an array of keys for each sorting method to sort. However, rather then printing the results and testing correctness, `sortComp.c` should get timing information as described below and print the input size (number of keys) and how much CPU time each algorithm took in a readable format.

Measuring the time

We will measure the amount of time taken by a sorting algorithm using the **clock()** function. To use this function, the `sortComp.c` file must include `time.h`. The `time.h` file exports the `clock_t` type, a `clock()` function that returns the current number of "clock ticks", and a `CLOCKS_PER_SEC` conversion factor that will let convert clock ticks into CPU seconds. The following outline shows how they are used.

[1] For information hiding, the struct should be declared in the `heap.c` file.

[2] You may have `isFull` return a boolean type rather than an int. You may restrict your heaps to hold only maxSize elements, and don't need to implement the "array doubling" technique briefly discussed in class.

[3] You may have `isEmpty` return a boolean type rather than an int.

```
#include <stdio.h>       /* printf */
#include <time.h>        /* clock_t, clock, CLOCKS_PER_SEC */

int main ()
{
  clock_t startTime,  duration;
  // read your data and initialize array of unsorted keys
  startTime = clock();
  // run the sorting algorithm
  duration = clock() - startTime;
  printf ("program took %f seconds.\n", ((float) duration ) / CLOCKS_PER_SEC);
  return 0;
}
```

`clock()` returns the number of 'ticks' (see the Wikipedia entry for "jiffy") used by your program's process. Since the frequency of ticks depends on the particular system you are running one (there could be 1000000 ticks per sec), a `CLOCKS_PER_SEC` constant is provided to convert a number of ticks into seconds. The calculated runtime is likely vary with different inputs, and might even fluctuate due to the other activity on your system. Therefore It is advisable to measure the run time several times with different input files and take the average of the various time values obtained.

Experimental procedure

You will need data of required size to perform your experiments. One good way to obtain this data is from the web site 'http://www.random.org'. You can use statistical packages like *R* or even use random number generators in c, or java to generate random data. Measure the performance for input files with 10, 100, 1000, and 10000 integers to be sorted. You should run your sortComp experiment at least three times for each input size with different input files. You can add in more sizes if you would like to get a clearer picture. Then *comment on the observations in README file for your submission*. Which of the algorithms is more efficient according to your data?

3 Submittal Information

You will need to submit the .c files, .h files, a single makefile that compiles the code and creates the three programs (`sortComp`, `sortPrint`, and `heapDriver`), and a README file which describes the contents and purpose of all files. You will need to make sure that the programs compile and run correctly on the UCSC unix systems. You will lose points for compile issues, and should not expect that graders will take much time trying to fix them. If the program does not compile, graders may choose to grade the code on non-execution aspects and this will significantly cost your grade.

You will be submitting the following 11 files: `insertionSort.c`, `insertionSort.h`, `heapSort.c`, `heapSort.h`, `heap.c`, `heap.h`, `heapDriver.c`, `sortPrint.c`, `sortComp.c`, `makefile` and a `README`. If you use your list ADT (or any other modules) also submit the module's `.c` and `, .h` files. zip all the files together into a file called lastNameFirstInitialProg2.zip and then submit this zip file through ecommons.

3.1 Grading Guide

10 points total:

- 1 point for correct submission with all files (including the driver program) and a README file.
- 1 point for separate ADT Files with information hiding, good comments (including pre and post conditions) and modularity.
- 1 point for well organized, readable code and generally good style.

- 1 point for good memory management (free-ing the arrays and struct you malloc or calloc).
- 1 point for correct insertion sorting.
- 3 points for correct heap sorting
- 2 points for proper experimentation and documentation of results in the README file

3.2 Hints

There is a lot to do in this assignment so try to get started early.

Visualize the program top-down to understand what needs to be done, but implement bottom-up, checking that each small part of the implementation is working well before moving on to the next small chunk.

If C is giving you trouble, you might find it easier to debug your algorithms in another language (e.g. Java) and then translate your code to C.

Implement a simple `printHeap()` function that prints the state of the heap so your driver can print the state of the heap and you can ensure your heap ADT is working correctly.

Although the insertionSort and heapSort functions take in an array of keys and return the sorted keys in the same array, they can be implemented by copying the keys (into a heap or inserting into a list) and then re-writing the keys in sorted order back into the array.