5DV022 Software Exercise 3 Fall 2008

Solutions to this assignment are due on October 20, 2008 at class time. This is a *laboratory exercise*, and the amount of collaboration allowed is as defined in the documents *Riktlinjer vid labgenomförande* (*Rules for the Preparation of Laboratory Exercises*) and *Hederskodex* (the *Honour Code*), which may be found on the course home page.

Late submissions are subject to penalty as described in the course syllabus.

This exercise may be carried out in groups of at most three individuals.

The report **must** be written in English. Reports written in Swedish will be returned, ungraded, unless prior written arrangement has been made with the course instructor.

Place your report in the mailbox of the instructor or give it to the instructor during class. Do not place it in the mailbox for laboratory reports.

1. General Description

The purpose of this assignment is to examine experimentally the average and worst-case complexity of insertion into and creation of a min-heap. Two fundamentally different measurement strategies will be used. The complexity of building a min-heap will be measured using profiling, while the complexity of insertion into a min-heap will be measured using instruction counting.

2. Specific Required Tasks

The experiment is divided into two distinct tasks. In the first, the complexity of building a min-heap will be measured, while in the second, the complexity of a single insertion will be studied. In both cases, the heap will be implemented as an array, as described in 3.5.11 of the course notes and Section 2.4 of the textbook. A high-level description of the data structure is as follows:

compl_bintree = **record**
tree : array[1..*max_tree_size*] of *value*;
last : 0..*max_tree_size*;
end record

All programs are to be written in C and the tests run on Linux and/or Unix systems. Thus, this data structure will be realized using a `struct`. Note that array indexing must begin with 1, and not 0 (as is the default in C), for the standard addressing rule to work (left child of the vertex at position n in the array is at position $2 \cdot n$, the right child is at position $2 \cdot n + 1$).

The best strategy is simply to ignore the array position indexed by 0; the "waste" of a single memory location is negligible in this context.

2.1 The Complexity of Building a Heap

A min-heap may be constructed in two distinct ways. In the first, the starting point is an empty heap, and the elements are inserted one-by-one. With n the total number of elements, mathematical asymptotic analysis shows that this approach is has time complexity $\Theta(n)$ in the average case, but $\Theta(n \cdot \log(n))$ in the worst. In the second approach, the elements are simply placed into the heap structure without any attention to order; subsequently, the structure is heapified using the procedure described in 3.5.14 of the course notes and Algorithm 2.11 in the textbook. Mathematical asymptotic analysis shows this algorithm to have time complexity $\Theta(n)$ in all cases.

In this part of the experiment, these two heap-building algorithms will be implemented, and their performance profiled and compared to that predicted by the theory. In each case, the algorithm is to be run on both random and worst-case data configurations.

For the average-case measurement, for each of the ten heap sizes $n = 5000$, $n = 10000$, $n = 15000$, ..., $n = 50000$, run the tests on three distinct random data sets. Thus, for the random case, there will be a total of 30 distinct configurations. The following particulars apply:

- For each such n, report the time as the average over the three random configurations; do not measure the performance of each random configuration separately.
- It is important not to measure the time consumed by generating the random numbers. Therefore, it is best to generate three random arrays of 50000 elements each, and use them for the source of data elements. For example, to get the three arrays of 15000 elements each, take the first 15000 elements from each of the three source arrays. The same source arrays should be used for both heap-generation algorithms.
- Use integer data arrays and comparisons. Thus, it will be necessary to generate (non-negative) integer random numbers.

The same algorithms are to be profiled for worst-case performance, but (obviously) on different data sets. For each of the two algorithms, the worst case is achieved with a data set which is in reverse order. The following specific points apply:

- For a heap of size n, the (ordered) data set will be n, $n-1$, $n-2$, ..., 1.
- For repeated insertion, this means that the elements will be inserted in the order given above; *i.e.*, first n, then $n-1$, etc.
- For heapification, this means that the initial data array will have *compl_bintree.tree*$[i] = n - i + 1$.

- For (the unique) data sets of size $n = 5000$, $n = 10000$, $n = 15000$, $\cdots$, $n = 50000$, run each experiment three times. Thus, there will be 30 test runs. For each size of data set, report only the average time over the three runs.

For both the average and the worst-case analyses the following apply:

- Both approaches formally require element swapping to propagate a value upwards in the heap, but this may be realized more efficiently by moving existing elements in the heap downwards, and finally inserting the new element into the vacant slot. See Algorithm 2.8 of the textbook for details.
- The main measurements should be made using the profiling package developed for Software Exercise 1. Small modifications to facilitate data collection are of course allowed.
- For at least one data point for each of the two algorithms, also measure the performance using the `gprof` tool.

2.2 The Complexity of Insertion into a Heap

As shown in [2] and [1], the average-case complexity of insertion into a random heap is constant; *i.e.*, $\Theta(1)$. In this part of the experiment, this complexity will be investigated experimentally. A measurement based upon instruction counting; specifically, counting the number of swaps which are required to position the new element correctly in the heap, will give a meaningful measure of the time complexity. The following particulars apply:

- Use the same data sets as for the heap-building experiment; thus, there will be 30 distinct configurations. For each of the ten heap sizes $n = 5000$, $n = 10000$, $n = 15000$, ..., $n = 50000$, the tests will be run on three distinct random data sets.
- For each of the 30 data sets, run 1000 test insertions, using random values. Each of the 1000 test insertions is to be run on the same 30 data sets; thus, each data item must be restored to its initial value before performing the next insertion.
- It is not necessary to perform the actual swaps. It is enough to examine the heap and determine how far up the new element would move were it to be inserted.

3. The Submission

The format of this assignment should be similar to that of the previous one. Specifically, your submission should be in the form of a report on an experiment. It should contain the following parts.

1. An introduction clearly describing the overall goal and ideas of the experiment
2. A detailed description of the experiment which was conducted. Include a discussion of any problems you encountered, together with an explanation of how you dealt with them. Also, clearly identify the environment in which the experiment was conducted; *i.e.*, the machine and programming environment on which it was run.
3. An analysis of the results of the experiment, together with the conclusions drawn from the experiment. This should include, but not be limited to:
   (a) For the experiments of 2.1, the following:
      (i) For the average case, a graph of the performance of the algorithm as a function of n.
      (ii) For the worst case, graphs of the performance of the algorithms as functions of both n and $n \cdot \log(n)$.
      (iii) For select data points (at least one for each algorithm), a profile based upon data collected using `gprof`. Any large discrepancy between the time reported by `gprof` and by your utility should be analyzed and explained.
   (b) For the experiments of 2.2, a graph is not necessary. Rather, provide a table showing the average number of swaps required as a function of n.
4. An appendix containing the following:
   (a) A source listing of your program.
   (b) Test runs on a small data set (say 20 elements) to show that the algorithm is correct.
   (c) All profiles generated, with each clearly labelled with the case considered.

If you just turn in a program and some data printouts, you will not receive much credit, even though your programs work perfectly. You **must** submit a clear description of what you did and what conclusions you reached. Furthermore, it must be organized as outlined above. This is an experimental investigation, not a programming exercise.

The report must be typeset; handwritten reports will not be accepted.

References

[1] E.-E. Doberkat, Inserting a new element into a heap, *BIT*, **21**(1981), 255–269.

[2] T. Porter and I. Simon, Random insertion into a priority queue structure, *IEEE Trans. on Software Engrg.*, **1**(1975), 292–298.