DM207 I/O-Efficient Algorithms and Data Structures

Fall 2009

Project 2

Department of Mathematics and Computer Science
University of Southern Denmark

November 1, 2009

The aim of this project is to gain practical experience with the effect of the memory hierarchy in the setting of searching.

Search Trees in Arrays

Recall the idea of the heap data structure that if a binary tree is laid out in an array in a breadth-first manner (root, then its children, then its grandchildren,...), then navigation from a node to its children or vice verse can be done by simple arithmetic, i.e., there is no need for pointers. Specifically, if the root is placed at index 0 in the array, navigation can be done using the following rule:

> For a node at entry i, its two children are at positions $2i + 1$ and $2i + 2$, and its parent is at position $(i - 1)/2$ (rounded down, i.e. the division is integer division).

The rule is easily generalized to trees of fixed degree larger than two (deduce it from a drawing).

The idea of the project is to create static balanced search trees of this kind, and investigate what is the best fan-out of the trees for various sizes of trees, when searching for random elements in the trees. By comparing the binary solution with the best solution, this should give an idea of what gains are achievable in the setting of searching by optimizing for I/O-efficiency.

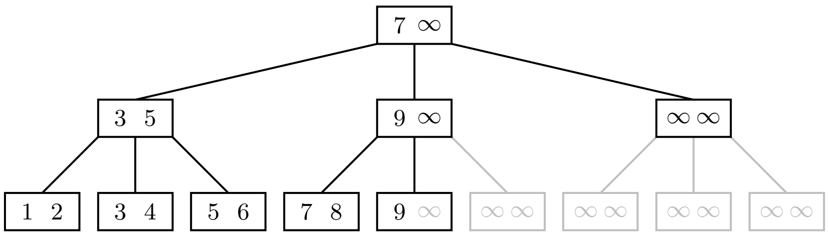
Experiments

Fan-outs of degrees $K = 2^k$ for $k = 1, 2, 3, 4, \ldots, 12$, and sizes of trees ranging from within L1 cache size (i.e., array size below 64 Kb) to above RAM size (i.e. disk should be reached), should be considered.

For ease of construction, the trees should be as follows: The lowest level of the trees should contain the integers from 1 to N. The levels above should have nodes with K children and $K - 1$ guiding elements. The ith guiding element should be the smallest value in the $(i+1)$th subtree, which for this data set can be calculated directly for each level in the tree due to the regularity of the stored keys (deduce the guiding elements from a drawing of the tree). In other words, each level of the tree can be created as a scan of an appropriate part of the array, directly placing the correct values which are found by calculation. For homogeneity, let also the leaves have $K-1$ elements.

Do not just consider tree sizes where the lowest level is full, as this will give far too few possible tree sizes. Rather, consider trees of heap shape, i.e., all levels except the lowest are full, while the lowest level contains N elements and is filled from the left. The (guiding) elements in nodes on levels above the lowest should be as if the lowest level had been padded with the value infinity in all the empty slots in the right part (so the value infinity will appear in the tree in the rightmost parts of the levels above the lowest—if these were removed, the node navigation formula would no longer be correct). As the value infinity, use `MAXINT`.

A tree with $N = 9$ and fan-out 3 will look like the one below, where the greyed out parts represent the padded parts, which will not be stored.



The corresponding array representation is as follows:

7 ∞	3 5	9 ∞	∞ ∞	1 2	3 4	5 6	7 8	9

For each fan-out, try traversing each node during the tree search by both linear search and binary search of the node, to see which is best (this may differ for each fan-out).

For each size, each fan-out of tree, and each node traversal method (linear or binary search), conduct the experiment of repeatedly searching for an appropriate number (a number making the total running time around one minute, possibly larger for the trees on disk) of random values in the the range $1 \dots N$. The cost measure should be wall clock time, which in C can be measured using the `gettimeofday` library call (see `man gettimeofday`). Plot the average running time per search against fan-out size, using e.g. `gnuplot`. A sensible value to plot could be running time divided by $\log N$, which should be a constant (at least when searching a node by binary search) if there were no impact of the memory hierarchy. From this, determine the best fan-out (and node search method) for each size, and consequently determine what gains in running time can be achieved by having a larger fan-out than binary.

Do not forget to test your programs before measuring, such as checking that the search return the value searched for. Measuring on incorrect programs teaches us nothing. The programs should be run on the local disk of the machine, not on a network file system (on the machines at Imada, work in `tmp`), in order to avoid having network latency mask the disk latency.

The program should be implemented in C or C++, and should be compiled using maximal optimization (e.g. option `-O3` for the `gcc` compiler). It is important to actually use the searched keys for something, since the compiler may remove computations and memory accesses that it can deduce have no influence on the outcome of the program. One suggestion is to have a counter to which each found key is added, and then at the end of the program to write out the value of the counter on the screen. Another suggestion is to keep an eye on the assembler-version of the code generated by the compiler, to see what code is actually running in the end (only feasible for small programs).

Pay close attention to minimizing CPU cycles as the CPU time can easily dominate the running time for experiments within cache. This may be at the expense of normal programming conventions. For instance, try to inline functions (although the compiler should do this to a large extent), use global variables, and minimize the number of `if/else`'s (they work against the processors pipelined instruction execution). For the smaller trees, it is quite conceivable that generating random integers will dominate the running time—this may be tested by measuring the time of a lot of calls to the random number generator versus the time for the same number of an elementary operation like addition of one. If this is the case, consider making in advance an array of N random integers in the range 1 to N, and then during testing traverse this array (repeatedly) while using the read values as the search keys. To make the size of this array significantly smaller than the tree, you may for some small integer k (e.g. $k = 16$) instead store N/k random integers in the range 1 to N, and before each traversal of the array generate a random value r and add it to the read search keys in that traversal (subtracting N if the resulting search key is larger than N). The creation of the array of random numbers should not be included in the time measurements for searches in the tree, and neither should the tree/array construction time.

For suggestions of ways to work with arrays past the size of the RAM, see the mails sent out during Project 1. In particular, for Linux on a 64 bit machine, using the `mmap64` library call should make it possible.

Scripting the execution of your entire set of experiments makes them easier for you to control and to redo if needed.

Formalities

Make a report of 6–8 pages describing your implementation and your experiments, on a level of detail such that others could repeat your experiments themselves. In particular, this includes reporting the compiler version, compilation options, and machine characteristics (at least disk, RAM, and cache sizes). Use many plots of your experimental data (not tables), and make sure it is explained what they show. Draw conclusions based on the observed data. Plots should be given as an appendix (not included in the page count above), code should be online available and an url to it should be given in the report.

You should hand in your report (in pdf) using the digital drop-box at the Blackboard page of the course (under menu item "Tools").

Deadline:

Monday, November 30, 2009, at 23:59.