DM207 I/O-Efficient Algorithms and Data Structures

Fall 2011

Project 2

Department of Mathematics and Computer Science
University of Southern Denmark

November 9, 2011

The focus of this project is search trees in external memory. The aim of the project is to gain practical experience with the effect of the memory hierarchy in this setting, as well as train theoretical argumentation.

The project is to be done in groups, preferably of size two (group size one is allowed).

1 Experimental Part

1.1 Search Trees in Arrays

We will consider static multi-way search trees of fan-out F, implemented in a way similar to the idea behind the heap data structure.

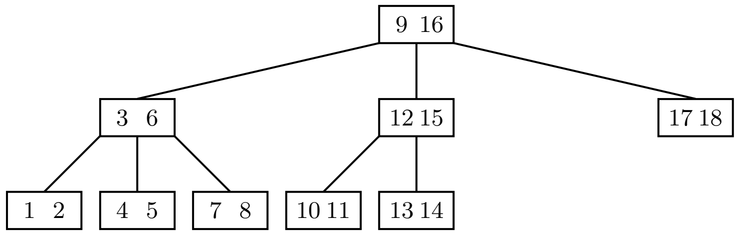
All internal nodes of the tree have the same number of children F. The tree is node-oriented (like a binary search tree is normally defined, but unlike a B-tree, which is normally defined in a leaf-oriented fashion): an internal node contains $F - 1$ actual elements, a leaf contains none and is really just a nil-pointer in an internal node. The elements stored in the tree fulfil the standard (multi-way) search tree order. The shape of the tree is the same as for the heap structure: the tree is perfectly balanced, and on the lowest level, all nodes are placed left-most possible.

If a tree is numbered in a breadth-first manner (root, its children left-to-right, its grandchildren left-to-right,...), then similarly to the heap structure, navigation from a node to its children or to its parent can be done by simple arithmetic on node numbers—i.e., there is no need for pointers. Specifically, if the root has number 0, navigation can be done using the following rule:

> For node number i, its F children are at positions $F \cdot i + 1, F \cdot i + 2, \ldots, F \cdot i + F$, and its parent is at position $(i - 1)/F$ rounded down (i.e. the division is integer division).

A node number i refers to a leaf (or another non-existing node) iff $i \geq N$, where N is the number of internal nodes. Since all internal nodes contain $F - 1$ elements, the number n of elements stored must be a multiple of $F - 1$. The entire tree is just an array of $n = N(F - 1)$

elements. If we by A$[k, l]$ denote the array entries from A$[k]$ to A$[l]$ (inclusive), then node i is the array entries A$[i(F-1), (i+1)(F-1)-1]$. A tree with $F = 3$, $N = 9$, and the first $n = 9(3-1) = 18$ natural numbers as elements, has the following tree structure



and the following array representation

Array index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	9	16	3	6	12	15	17	18	1	2	4	5	7	8	10	11	13	14
Node number:	0		1		2		3		4		5		6		7		8	

1.2 Tasks

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.

1. Implement a DFS traversal of a tree of $n = (F-1)N$ elements, which visits its elements in (multi-way) search tree order.

2. Based on linear search, implement a search procedure of a node A$[k, l]$, which for a search key s returns the smallest $i \in \{k, k+1, \dots, l\}$ for which $s \leq$A$[i]$ if such i exists, and else returns $l + 1$.

3. Based on binary search, implement a search procedure of a node A$[k, l]$, which for a search key s returns the smallest $i \in \{k, k+1, \dots, l\}$ for which $s \leq$A$[i]$ if such i exists, and else returns $l + 1$.

4. Let the possible fan-outs be $F = 2^i$ for $i = 1, 2, 3, 4, \dots, 8$, and let the possible sizes be given by $n = (F-1)\lceil n'/(F-1)\rceil$ for $n' = 1.8 \cdot 2^j$ and $j = 14, 19, 25, 28$. For each size n, each fan-out F, and each node traversal method (linear or binary search), perform the following experiments on a computer with the available RAM reduced to 256Mb (see instructions for reducing RAM in the text of the last project): First create a tree containing the integers from 1 to n by using your DFS procedure (and a counter) to fill them into an empty array of length n. Then repeat an appropriate number of times (defined as a number making the total running time around ten seconds, possibly larger for the trees on disk) the action of getting a new random integer in the range 1 to n and searching for it in the tree. Measure the total running time (wall clock time).

For each size, plot for each traversal method the *average* running time per search as a function of $\log F$. More precisely, plot average 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 traversal method for each size, and consequently determine if any 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 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, C++, or Java, and should for the first two languages 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 possibility 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.

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 do not use more `if/else`'s than necessary (they work against the processors pipelined instruction execution). For the smaller trees, it is quite conceivable that generating random integers for the search keys 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.

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

In Java, you will need to set the heap size to around 2Gb by using the option `-Xmx` (as in `java -Xmx2G JavaProgram`).

2 Theoretical Part

2.1 Task(s)

1. Prove statement 3 of Theorem 1 in the note *Amortized Analysis of (a,b)-Trees* by Rolf Fagerberg (October 2009)
2. Extra (not mandatory): Prove the tree navigation rule stated on the lower half of page 1 in this project text.

3 Formalities

In your report, the first part should describe your implementation and your experiments, on a level of detail such that others could repeat your experiments themselves. This for the experiments includes reporting the compiler version, compilation options, and machine characteristics (such as disk, RAM, and cache sizes). Use plots of your experimental data (not tables), and make sure it is explained what they show. Discuss and draw conclusions based on the observed data. Plots should be given as an appendix (not included in the page count below), code should be online available (not included in the report) and an url to it should be given in the report. This first part should have around 3–6 pages (excluding plots).

The second part of the report should contain your proof of statement 3 of Theorem 1 (and possibly the proof of the extra task), and should have around 2–5 pages.

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

For groups of size more than one: For formal reasons, you will need to designate who wrote which part of the programs and report.

The project will be evaluated by pass/fail grading. The grading will be based on:

- The clarity of the writing and of the structure of the report.
- The thoroughness of the experiments—execution as well as discussion.
- The correctness of the proof.
- The total amount of work done.

Deadline:

Monday, December 12, 2011, at 23:59.