DM207 I/O-Efficient Algorithms and Data Structures

Fall 2015

Project 2

Department of Mathematics and Computer Science
University of Southern Denmark

December 2, 2015

The first part of this project is practical and its aim is to gain practical experience with the effect of the memory hierarchy on search trees. The second part is theoretical and its aim is to train creativity and 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.

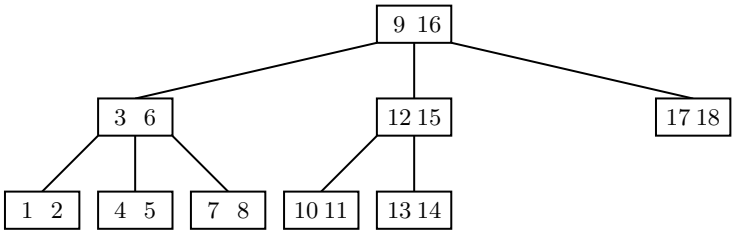
All internal nodes of the tree have the same number of children F (except the last internal node, which may have fewer). The tree is node-oriented (like binary search trees are usually defined, but unlike B-trees which are usually defined in a leaf-oriented fashion): an internal node (except the last) contains $F-1$ actual elements, while 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 the nodes numbered $F{\cdot}i{+}1, F{\cdot}i{+}2, \ldots, F{\cdot}i{+}F$, and its parent is the node numbered $(i-1)/F$ rounded down (i.e. the division is integer division).

A node number i refers to a leaf iff $i \geq N$, where N is the number of internal nodes. For simplicity of code for searching inside a node, we in this project assume that all internal nodes (including the last) contain $F-1$ elements, hence the number n of elements stored must be

a multiple of $(F-1)$ and is given by $n = N(F-1)$. The entire tree is implemented simply as an array of these n elements. If we by $\mathrm{A}[k,l]$ denote the array entries from $\mathrm{A}[k]$ to $\mathrm{A}[l]$ (inclusive), then node number i is the array entries $\mathrm{A}[i(F-1),(i+1)(F-1)-1]$. To illustrate, 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 implementation

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		

1.2 Tasks

The idea of this part of the project is to create static balanced search trees of the kind described above, 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 (if any) are achievable in the setting of searching by optimizing for I/O-efficiency. We will only consider trees in RAM.

The programming language should be Java. The elements should be `int`'s, hence the entire data structure should be an array of n `int`'s.

1. Implement a DFS traversal which visits the elements of the tree in (multi-way) search tree order.

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

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

4. Let the possible fan-outs be $F = 2^i$ for $i = 1, 2, 3, 4, \ldots, 12$, and let the possible tree sizes be given by $n = (F-1)\lceil n'/(F-1)\rceil$ for $n' = 1.8 \cdot 2^j$ and $j = 17, 27$. For each size n, each fan-out F, and each node traversal method (linear or binary search), perform the following experiments: 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 be around 30 seconds), 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 of the search phase (not the DFS phase) using `http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis()`. For each size, plot for each traversal method the *average* running time per search as a function of $i = \log F$. If some of the curves are much larger than the rest, make extra plots containing only the rest of the curves. 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.

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 RAM. For instance, optimize/simplify the navigation calculations (maybe even hardcode the values of F into the programs), avoid small utility functions, and do not use more `if/else`'s than necessary (they work against the processor's pipelined instruction execution). Generating random integers is quite expensive compared to, say, additions of `int`'s. To avoid this masking the actual search time, the search keys should be generated by creating before the search phase an array of k random integers in the range 1 to n, and then during the search phase traverse this array (repeatedly) while reading values from it and using these as the search keys. To make the size of this array significantly smaller than the array of the tree, choose $k = n/20$ and before each traversal of the array generate a single new random value r and add it to the read search keys s in that traversal, using $(r + s) \bmod n$ as the actual search key. The creation of the array of random numbers should not be included in the time measurements for the search phase.

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

You will probably need to increase the heap size by using the option `-Xmx` (as in `java -Xmx4G JavaProgram`).

2 Theoretical Part

2.1 BFS on trees

A BFS-numbering of a tree is a labelling of the nodes of the tree with numbers $1, 2, 3, \ldots$ such that

1. the root has label 1,
2. nodes in BFS-level i have smaller labels than nodes in BFS-level $i+1$,
3. for any two nodes in BFS-level $i+1$, their labels are in the same order as the labels of their parents.

The nodes in BFS-level i are the nodes whose path to the root contains i edges.

2.2 Task

Give an $O(\text{Sort}(E))$ external algorithm for computing a BFS-numbering of a tree (given as a list of edges oriented downwards from the root), where E is the number of edges in the tree.

You should give a clear description of your algorithm, as well as arguments on its correctness and its I/O-cost.

[Hint: You may be inspired by the algorithm for DFS on trees. Also, the fact that $\sum_i \text{Sort}(N_i)) \leq \text{Sort}(\sum_i N_i)$ may be useful (but should be proved if used).]

3 Formalities

The first part of your report should describe your implementation and your experiments, on a level of detail such that others could repeat your experiments themselves (based on the project text and your description, i.e., you may assume the project text is known). State the Java version, compilation options, and machine characteristics (such as RAM and cache sizes). Use plots (not tables) to report your experimental data as described above, and make sure it is clear to the reader what they show. Discuss and draw conclusions based on the observed data. Source code should not be included in the report, but the source files should be part of the hand-in.

The second part of the report should contain a clear description of your algorithm, as well as arguments on its correctness and its I/O-cost.

You should hand in your report (in pdf) and your source files as one zip-file using *SDU Assignment* at the Blackboard page of the course.

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

- The clarity and comprehensiveness of the writing and the structure of the report.
- The thoroughness of the experiments in the first part—execution as well as discussion.
- The correctness of the algorithm in the second part.
- The total amount of work done.

Deadline:

Thursday, January 7, 2016, at 12:00.