Department of Mathematics and Computer Science
University of Southern Denmark, Odense

October 28, 2014
KSL

Part 2 of the exam project in
DM803 – Advanced Data Structures

Kim Skak Larsen
Fall 2014

Introduction

In this note, we describe one part of the exam project that must be solved in connection with the course DM803 – Advanced Data Structures, Fall 2014. It is important to read through the entire project description before starting the work on the project; also the sections on requirements and how to turn in your solution.

Deadline

The deadline for this part of the project is

Friday, November 28, 2014 at 12:00 (noon).

Tracks

For the project in this course, you can choose between two tracks, i.e., you choose one of them and focus only on that track. The tracks are named as follows:

- The *programming* track
- The *proof* track.

Below, we described the content of the two tracks with regards to this part of the project.

Before doing so, we describe some background material needed for both tracks.

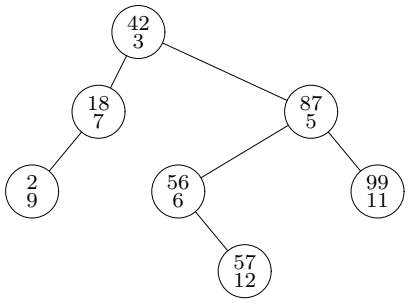

Figure 1: Example Priority Search Tree. Keys are listed above priorities in the nodes.

Priority Search Trees

The design idea is the following: Every time we want to insert a new key in the search tree, we also choose a priority, uniformly at random. The new node we insert will contain both the key and the priority. We require that the tree is a search tree with respect to the keys and is heap-ordered with respect to the priorities. In Figure 1, we show an example, listing keys above priorities in the nodes. Here, we have used "small" priorities for illustration, but in a real implementation, priorities should be chosen from a very large domain, e.g., all 32- or 64-bit integers. For simplicity, we will assume that no keys get the same priority.

Operations on the tree are implemented using the following outlines. We point out that there are many smaller decisions to make while realizing these ideas in an actual program. The order of the implementations of the operations is important, since, as we very often do in data structures, one operation is implemented by using other operations.

The data structure takes up $O(n)$ space and all operations have complexities of the order of the height of the trees.

Search

As in any standard search tree.

Insertion

Given a tree and a key, choose a priority uniformly at random, form a node, and insert the node at the correct location according to the search tree invariant. Use single rotations (which preserve the search tree invariant) to move the node up until the tree is again heap-ordered.

Split

Given a tree and a key k, split should return two trees: one with all keys smaller than k and one with all keys larger than k. (We assume, for simplicity, that k is *not* in the tree.)

The implementation idea is to temporarily insert k, but instead of a random priority, we give it a priority smaller than any priority currently in the tree. After the insertion, return the left and right subtree of the root as the result.

Merge

Given two trees where the keys in one tree are smaller than all keys in the other, we want to merge the two trees into one. We use an idea similar to the recursive procedure from weekly note 2 for leftist heaps: The node with the smallest priority should be the root node of the result. Considering the three trees consisting of this node's left and right children and the other argument to the merge operation, either its left or its right child will contain the middle range of keys. Let the root node keep the child that does *not* contain the middle range of keys, and replace its other child tree (the tree with the middle range keys) with the (recursive) merge of that tree and the other argument of the merge operation.

Deletion

Instead of the node to be deleted, place the merge of its two children.

The Programming Track

There are two independent tasks: Implement priority search trees and implementing partially persistent doubly-linked lists.

Both data structures must be implemented as outlined in this note and the articles describing them, and in such a way that the same time and space complexity bounds hold.

Note that in the general rules below, we state some implementation requirements in addition to the ones given in this section.

Priority Search Trees

The operations for priority search trees are described above. After implementing them, you should investigate their properties. Your philosophy should be that you want to investigate and understand, and then you want to communicate the results as clearly as possible to someone else afterwards. Thus, in particular, you should include graphs showing the connections for all your findings; not just tables with measurements. Below are some requirements and ideas, but you can supplement with additional questions.

Investigate the following:

- Average search complexity.

  You can simply use the depth of the node a key resides in as a measure for the time complexity of searching for that key. Investigate average search complexity as a function of n.

- Variation in search complexity.

  For a large number of searches, record for each i how many of these searches had time complexity i, and illustrate by graphing the percentage of searches of complexity i as a function of i. If this does not give a good illustration, consider accumulating, i.e., at i, give the percentage of searches of complexity i or smaller. You can also consider relating i to $\log_2 n$, either additively or multiplicatively, if that gives better information.

Partially Persistent Doubly-Linked Lists

The basic structure consists of items with three fields: an integer field `key`, and pointer fields `next` and `prev`. To access the list, we have a reference to the first element. Subsequent elements are reached via the `next` pointer, and the last element has a `next` pointer which is `nil` (also called `null` or `none`). If an element A's `next` pointer points to B, then B's `prev` pointer points to A.

You must provide operations `newversion`, `search`, `insert`, and `update` in a partially persistent version of this data structure, using techniques from [1].

The operation `newversion` changes to a new version.

The operation `search` takes a version number v and an integer i as arguments and returns the key of the ith element of the vth version.

The operation `insert` takes a key k and an integer i as arguments, and insert the key k as the new ith element in the list, i.e., between the $(i-1)$st and ith element of the current newest version.

The operation `update` takes a key and an integer i as arguments, and updates the key in the ith element to the given key in the newest version.

Experimentally determine the connection between the number of extra pointers and the space used, i.e., space as a function of extra pointers. Do this both when you count space as the number elements created in all versions and when you count space as the total size of memory allocations in all versions. For the latter, you may count this in terms of number of fields, i.e., you can assume that integers and pointers take up the same amount of space, and that an element takes up space equal to the number of fields in it.

Other Remarks

For the report, other than the investigations described above, you need only discuss possible important choices in your implementation. There is no reason to go through code that is simply implemented as outlined in the papers.

Remember to read the general rules, where there are also explicit requirements to programs and reports.

The Proof Track

In this track, you must solve the four independent problems described below.

Priority Search Trees

First argue that given a collection of nodes with keys and priorities, the appearance of the tree is unique.

We want to establish expected running times of the operations. Let $k_1, \ldots, k_n$ be all the keys in the tree in sorted order, and let $p_1, \ldots, p_n$ be the corresponding

priorities. Consider the subsequence $k_i, \ldots, k_j$. Argue that

$$k_j \text{ is an ancestor of } k_i \iff p_j = \min\{p_i, \ldots, p_j\}$$

Let $\mathbf{1}_X$, where X is a random Boolean variable, denote the value one if X is true and zero otherwise. Clearly, the expected depth of a node is the expected number of ancestors it has. Thus, the expected depth of k_i is

$$\mathrm{E}\left[\sum_{j=1}^{n} \mathbf{1}_{k_j \text{ is an ancestor of } k_i}\right] = \sum_{j=1}^{n} \mathrm{E}\left[\mathbf{1}_{k_j \text{ is an ancestor of } k_i}\right]$$

From your first argument from the above, you can compute the expected value of $\mathbf{1}_{k_j \text{ is an ancestor of } k_i}$.

Using Harmonic numbers, give a logarithmic upper bound on the expected depth of k_i.

Why does this not necessarily imply that the expected height of a priority search tree is $O(\log n)$?

Now, we want to prove it, and you may use the following outline for that. Let $H(n)$ denote the expected height of a tree with n nodes. In such a tree, some key will have the smallest associated priority. This node will be the root of the subtree and that will determine the number of nodes going into the left and right subtree. We let X_n denote the number of nodes in the largest of these, i.e., the maximum of the number of nodes in the left and in the right subtree. Then the expected height of the tree can be written as

$$H(n) = 1 + \sum_i \text{Prob}[X_n = i] \cdot H(i)$$

Argue that the right-hand side is bounded by

$$1 + \text{Prob}[X_n \leq \tfrac{3}{4}n] \cdot H(\tfrac{3}{4}n) + \text{Prob}[X_n > \tfrac{3}{4}n] \cdot H(n)$$

Bound the probabilities to get a recursion equation for H and show that $H(n) \in O(\log n)$.

Layered Heaps

A layered heap maintains a collection of fixed-length tuples. Here, we will just say that the tuple length is three. The purpose of the data structure is to be able to

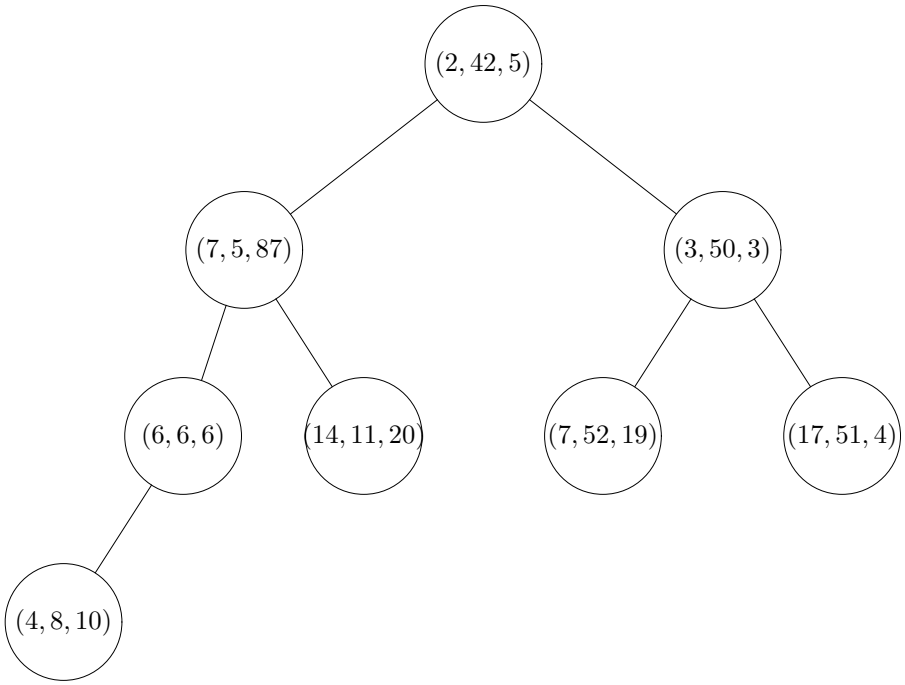

Figure 2: Example Layered Heap.

efficiently (and separately) extract the tuple with the smallest position 1 value, the smallest position 2 value, and the smallest position 3 value. Thus, we have three different `deletemin` operations, $\texttt{deletemin}_i$, $i \in \{1, 2, 3\}$.

For the structural invariant, we organize the data in the usual heap shape, as known from introductory data structures. The ordering invariant is the following: The root layer is referred to as a position 1 layer, the children of the root are in a position 2 layer, then we have position 3 layer, and then we start over, continuing down the heap. At a position i layer, any node must have a tuple having the smallest position i value of any tuple in its subtree. Figure 2 shows an example.

Develop $O(\log n)$ algorithms; first for `insertion` and then for $\texttt{deletemin}_i$, using the usual heap strategy of first of all respecting the structural invariant, and then fixing the ordering invariant. The height is obviously logarithmic, so you just have to ensure that you can progress up (respectively, down) in the tree in each step after a constant amount of work, maintaining that you always have at most one problem node.

Explain the algorithms, or give them in pseudo-code (or implement them), and

argue briefly why the algorithms are correct.

Broad Heaps

Consider a heap where we let nodes have degree $d > 2$. Everything else is unchanged with regards to structural and ordering invariants.

Express the complexity of the operations

`insert`, `deletemin`, and `descreasekey`

as a function of both d and n.

Fibonacci heaps were basically invented to give Dijkstra's algorithm the complexity $O(n \log n + m)$ in a graph with n vertices and m edges. However, Fibonacci heaps are a bit cumbersome with lots of pointers, so in this assignment, we want to avoid them. Using standard binary heaps, the complexity of Dijkstra is $O(m \log n)$. Recall where this expression comes from in terms of how many operations of the different types must be carried out.

Now, we will use Dijkstra's algorithm on families of graphs where the number of edges grows faster than the number of nodes. Assume that we have the dependency $m = nf(n)$, where $f(n) \in \omega(1)$. Write up the complexity of Dijkstra's algorithm for this case and choose a degree for the heap (that may depend on n) such that Dijkstra's algorithm runs in time $o(m \log n)$, i.e., strictly better than $\Theta(m \log n)$.

Beyond Partial Persistency

In what we covered from [1], updates where always performed in the newest version. Here we consider a more flexible approach. Assume that the newest version is version i, and let $1 \leq j < i$. We allow that one can switch to version $i+1$, make updates based on version j, and stop. The process can then be repeated. Making updates based on version j means that you do not follow pointers with version numbers strictly between j and $i + 1$. In this way, the versions no longer form a linear sequence, but instead form a tree structure, e.g., in what we described above, version j will now have both versions $j + 1$ and $i + 1$ as successors in what we could call the version tree.

Go through the algorithms and determine if this gives rise to any differences or problems, and discuss if they can be overcome.

Assuming that this can be worked out, we still have that one constant time step in the ephemeral structure is simulated by an amortized constant time protocol in

the persistent structure. However, how are amortized results from the ephemeral structure transfered? For instance, considering the result that rebalancing after an insertion in a red-black tree is amortized constant, can that be transfered to a similar positive result for this more flexible persistency structure?

General Requirements and Rules

Here we list general requirements, procedures for turning in, and exam rules.

Exam Rules

This is an exam project, and cooperation is not permitted. It will be considered cheating and will be treated as such. You have a duty to keep your notes private and protect your files against reading and copying by others. Both parties involved in a possible plagiarism can be held responsible.

There will be given what we judge to be more than sufficient time for solving the project. Still, we strongly encourage you to plan your work such that you will finish some days before the deadline.

Solutions that are turned in after the deadline will not be accepted. Downtime on the system or the printers will not automatically result in an extension; not even if it is the last hours before the deadline. Neither will own or children's illness without a statement from your physician, etc.

The solution

Depending on which track you choose, the solution may consist of just a report or may consist of a program, test material, and a report.

The Report

The front page of your report must include your full name, the first 6 digits of your CPR number, and student e-mail address (the prefix to `nn@student.sdu.dk`).

There are not supposed to be any of the following, but if there are possible omissions, known errors, etc. they should be described in the report. It is often a good idea to do this in a separate section instead of mixing it in with the rest of the report.

The report should in the best possible manner account for the entire solution, i.e., if your solution includes programs, the report must contain a description of the most important and relevant decisions that have been made in the process of developing the solution and reasons must be given where this is appropriate. You must also explain how possible programs have been tested. Test examples or references to test examples and test runs can and should be included in the report to the extent that this is helpful to understand your solution and to be convinced that it is working correctly.

Programs

When you have to implement algorithms, you must implement them in such a way that you obtain the asymptotic complexities claimed in the course material. If you happen to find a library, a package, or similar that implements the algorithms that you are supposed to implement, then you are of course not allowed to use them. You should make the implementation yourself from scratch using the basic features of the programming language you are working with. The safest approach is not to use libraries and avoid non-trivial built-in features.

Files and directories should be named and organized logically. Programs must be well-structured with appropriately chosen names and indentation and tested sufficiently.

Programs that are turned in must compile and run on IMADA's machines.

The preapproved programming languages you can choose from are the following:

- C or C++
- Java
- Python

If you have other preferences, you could likely obtain permission to use an alternative. Contact the lecturer.

You are very welcome to develop your programs at home, but it is your responsibility. This includes technical problems at home, lack of access to relevant software, moving data to IMADA via e-mail, USB keys, etc. and converting to the correct format, e.g., between Windows and Linux.

You must turn in a file `manual.txt`. Here you must explain how to compile and run your code on IMADA's computers. Be careful not to hardwire references

to your home directory etc. into the code such that it will not be possible for us to compile or run your program directly. Your goal should be to make it as easy as possible for us to run your program on additional tests to the ones you have provided. You have to make everything really clear, e.g., which directory one should be in, the order in which commands should be carried out, etc. You are advised to make things as simple as possible. The safest is usually to have all source files and executables in one directory. Of course, your report and test files can be in subdirectories.

Turning In

You must turn in on paper *and* electronically. The details are given below. All material that is turned in both on paper and electronically must be identical.

On Paper

You must turn in your report on paper at IMADA's secretaries' office in your lecturer's letter box. The office may be closed for very short periods of time. If, for some unexpected reason, the office must be closed for longer periods of time close to the deadline, an announcement will be made outside the office, giving instructions as to where you turn in.

Electronically

Electronically, you must turn in everything, i.e., the report in pdf-format, named `report.pdf`, and, if you have chosen the programming track, all relevant programs, test files, and `manual.txt`.

You upload your files using "SDU Assignment" in Blackboard (which will give you a receipt). You should avoid Danish (and other non-ascii) characters (such as æ, ø, and å) in your directory and file names (Blackboard does not handle this well). To be safe, also avoid spaces and all special characters not normally occurring in file names.

You may upload your files individually or collect your files into one (archive) file (recommended) before uploading. If you choose to do the latter, you may use `zip`, `bzip2`, or `tar` (with or without `gzip`).

References

[1] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.