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

March 17, 2022
KSL

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

Kim Skak Larsen
Spring 2022

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, Spring 2022. 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

Monday, May 8, 2022 at 23:55.

Randomized Search Trees and Partial Persistency

There are two independent tasks: Implementing randomized search trees and implementing a partially persistent list structure (to be defined).

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.

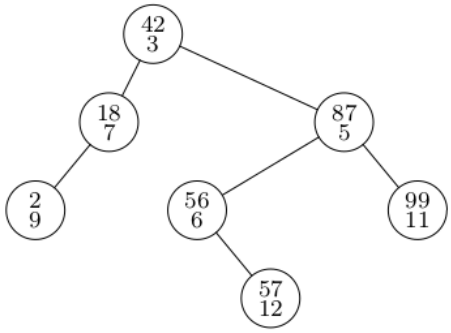

Figure 1: Example Randomized Search Tree. Keys are listed above priorities in the nodes.

Before stating the tasks, we describe some background material not covered in lectures.

Theory of Randomized 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 (64-bit) integers. Thus, it is unlikely that keys get the same priority, but it is fine if they do.

Operations on the tree are implemented using the following outline. 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. (You may 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 all 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 we used for an alternative to leftist heaps discussed in the exercises: The node, x, with the smallest priority should be the root node of the result. Consider the three trees consisting of this node's left and right children, x_{left} and x_{right}, and the other argument to the merge operation, y. Either x_{left} or x_{right} will contain the middle range of keys. Let z denote the one of x_{left} and x_{right} that contains the middle range keys and let z' denote the other tree. We give the node x the following two children, ordered appropriately according to the keys: z' and the tree resulting from a recursive merge of z and y.

Deletion

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

Randomized Search Trees

The operations for randomized 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.

A Partially Persistent List Structure

In the following, we describe a structure which is not very complicated, but a little more than trivial. The reason for its definition is that it enables you to demonstrate that you can handle several types of pointers in the set-up for the technique; see Fig. 2 for an example structure.

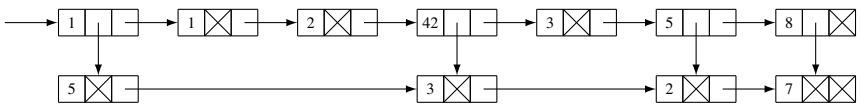

Figure 2: An example list structure.

In the following, we refer to the designated value that indicates that a pointer is not pointing to another storage space as `nil`. This is also sometimes referred to as

`null` or `none`).

The basic structure consists of items with three fields: an integer field `key`, and pointer fields `assoc` and `next`.

We want to implement a singly-linked list (based on the `next`-fields) where elements can, optionally, have a pointer (`assoc`) to an item with additional information. If there is no such item, then the `assoc`-field is `nil`.

Furthermore, all the items with associated information are also linked via their `next`-fields, in the same order as the items referring to them, and their `assoc`-field is always `nil`.

The `next`-field of the last item in each list is `nil`.

To access the list, we have a reference to the first element.

You must provide operations `newversion`, `search`, `insert`, and `update` in a partially persistent version of this data structure, using the node-copying technique from [1] (that is, you should, for instance, *not* use the fat node technique).

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, as well as the key from the associated item, if it exists.

The operation `insert` takes a key k and an integer i as arguments, and optionally also a second key k'. It inserts 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. If there is no second key, the `assoc`-field should be set to `nil`, and otherwise, there should be a reference to a new item with key k'; this item should of course be inserted correctly into the list of associated items.

The operation `update` takes a key and an integer i as arguments, and optionally also a second key k'. It updates the key in the ith element to the given key in the newest version, and also the key in the associated item, if present.

To handle the optional keys in the function calls, you may decide that keys are nonnegative and use -1 to indicate that there is no associated key. However, you are *not* allowed to create items for nonexisting associated keys. In other words, you should be able to create a structure looking like the one in Fig. 2.

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. Try to determine if there is an optimal number of extra pointers to choose.

To get reliable results, you must grow your structure to a relatively large size, but you will discover that from experiments. It is a good idea to fix the number of associated elements, at least probabilistically, e.g., create an associated element with probability $\frac{1}{3}$.

Specific Execution Requirements and Input/Output Formats

You can let your programs take options such that you can define your own behaviors as well. However, when executed without any options, the programs should follow the directions below.

Your programs must read from `stdin` and write to `stdout`. Keys are integers.

It is part of the assignment for you to design and clearly describe input/output formats for testing the implementation of the two tasks of this project. As a minimum, one should be able to test semantic correctness, e.g., for the dictionary, is an inserted/deleted key present or not? It would be even better if the formats also allowed for (some) testing of internal correctness, e.g., just because one can confirm that an inserted key is in the randomized search tree, this does not guarantee that it is implemented correctly; for instance, it could be an implementation of a red-black tree. ☺

General Requirements and Rules

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

Exam Rules

You can do this project alone or in pairs, referred to as a group (consisting of one or two people). This is an exam project, and cooperation outside of your group 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.

On Making the Deadline

I always set deadlines so you have plenty of time for a project compared to how long it takes, enabling you to schedule your work where it is most convenient for you. I do not give extensions without formal and objective reasons such as long-term, documented illness, for instance. However, if you notice a problem with the deadline shortly after it has been announced, please alert me and I will consider changing it. If you are interested in why I try to follow this policy and why I think it is a matter of fairness not to extend deadlines, you can read about that here:

`https://imada.sdu.dk/~kslarsen/Blog#18`

The Report

The front page of your report must include your full name and student e-mail address (the prefix to `@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 everything, i.e., the report in pdf-format, named `report.pdf`, all relevant programs, test files, and `manual.txt`.

You upload your files using Itslearning (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 (often not handled well). To be safe, also avoid spaces and all special characters not normally occurring in file names.

If possible, please collect your files into one (archive) file before uploading. 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.