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

September 19, 2014
KSL

Part 1 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, October 24, 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.

The Programming Track

The aim is to implement scapegoat trees and skip lists, investigate properties of these separately and compare them. Note that in the general rules below, we state some implementation requirements in addition to the ones given in this section.

Both data structures must be implemented as outlined in the articles describing them, and in such a way that the same time and space complexity bounds hold, and both must support at least the operations `search`, `insert`, and `delete`.

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.

Instead of measuring time, you must count comparisons, i.e., in the following when we refer to the time complexity of operations, we mean the number of times two keys are compared.[1] As a starting point, investigate the issues below when searches are uniformly random. An easy way to handle this would be the following: Decide on an n, insert the keys 1 through n in uniformly random order, and then search a number of times, choosing the key to search for every time uniformly at random from $\{1, \ldots, n\}$. You can of course also investigate other distributions or worst-case ordering if you wish, e.g., what happens if you insert the numbers 1 through n in sorted order?

Investigate the following:

- Average search complexity.

  For both structures, illustrate this using a graph where you display the search complexity as a function of n. Can you demonstrate that the search time logarithmic? Can you find out what the constant is in front of the logarithm, assuming that we use base 2?

  For skip lists, try to investigate the connection between the choice of p and the constant in front of the logarithm. You can do this for a fixed, large enough n. Does it behave as predicted theoretically?

[1] These data structures are extremely fast, which means that it is basically impossible to get meaningful results by simply timing operations. If trying to time extremely fast operations, actions of the underlying operating system disturb the results so much that results become meaningless. To work around this, one can decide to work with a very large n, so that running times get slowed down. However, n has to be so large that data storage effects, such as cache misses at different levels, start ruining the measurements instead.

- 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.

- For scapegoat trees, investigate frequency and size (number of nodes involved) of restructurings, and the relationship between the two.

Compare and discuss your findings and supplement with you own ideas, if any.

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 three independent problems described below.

The Complexity of Disjoint Sets

Read [1, Chapter 2] and compare with the proof of complexity for Disjoint Sets covered in the course. Make a brief list, explaining the structure (the major steps) in the proof from the course. Then explain which steps are the same as in [1, Chapter 2], and explain in greater detail how the differences lead to the better result in [1, Chapter 2]. You are not supposed to write a lot or repeat calculations from the literature.

Global Rebuilding

For randomized state space exploration, it could be useful to have a data structure supporting the two operations `insert` and `deleteRandom`. To implement the latter efficiently, we decide to store our data consecutively in an array indexed from zero, i.e., if there are n elements, the last element has index $n - 1$. First, assume that n will never be larger than some fixed s, which we then choose to be the size of the array. Then we can implement the operations as follows:

```
def insert(e):
  A[n] = e
  n = n+1
```

```
def deleteRandom():
  i = ⌊random() * n⌋
  e = A[i]
  n = n-1
  A[i] = A[n]
  return e
```

Here, `random` is assumed to return a value in the range $[0, 1)$ uniformly at random.

Now, we no longer assume a known upper bound on n. We need to change the data structure, and we want space usage to be $O(n)$. Let s denote the current size of the array. We decide that when $n = s$, and we want to insert a new element, we move all current elements to a new array of size $2s$. Similarly, if $n = \frac{s}{4}$ and we want to delete an element, we move all current elements to a new array of size $\frac{s}{2}$.

Define a potential function, Φ, and show that both operations are amortized constant time.

Now, we want the operations to be worst-case constant time, and still limit the space use to worst-case $O(n)$.

The idea to obtain this is to spend constant extra time every time we carry out one of the two operations, and use this extra time for gradually moving the elements to a new array (either twice or half as large as the current, depending on how n has been changing). When the new array has been finished, we switch to using that and throw away (deallocate) the old one.

Make the details of this algorithmic idea precise, using pseudo-code as above, and prove the worst-case space and time results.

Properties of Red-Black Trees

You are supposed to know red-black trees; in particular that

- nodes are either red or black,
- any path from the root to a leaf has the same number of black nodes, and
- no red node has a red parent.

Red-black trees can be presented in different variants. Below, we describe the one that we will use now, and it may be a little different from the one you were taught.

For an *insertion*, we add a new leaf in the appropriate place and color it red. If the parent is also red, we apply the operations in Figure 1 until the problem is fixed.

For a *deletion*, we want to physically delete a node that has at most one child. We obtain this as follows: If the key we want to delete is in a node u that has a left child, then we find the predecessor node v, overwrite the key in u by the key in v, and then proceed to delete the node v (which cannot have a right child). In this way, the node we are actually deleting has at most one child. Thus, we can cut out this node by letting its parent refer to its child.

If the node we cut out is red, then we are done. Otherwise, we mark the parent node by "–". One can think of the node being double-black, so that there are still the same number of black nodes on every path. Using the operations from Figure 2 repeatedly, we remove the mark.

First, assume that we only perform insertions.

Divide the rebalancing operations into ones that solve the problem of two consecutive red nodes and ones that may not immediately solve the problem. For the latter, figure out the exact conditions under which the problem is not solved, i.e., what are the colors of children, parent, siblings, etc. to the nodes in the configurations.

Let $\mathcal{C}_I$ denote a configuration in the tree where a black node has two red children, and let $\#\mathcal{C}_I$ denote the number of such configurations in tree. Now, argue that if we define a potential function $\Phi_I(T) = \#\mathcal{C}_I$, then rebalancing after an insertion is amortized constant.

Now, assume that we only perform deletions. Repeat the process above, but now let $\mathcal{C}_D$ denote configurations, where a black node has two black children, and let $\Phi_D(T) = \#\mathcal{C}_D$. Argue, using this potential function, that rebalancing after deletion is amortized constant.

Why do these two results not imply that if we have mixed insertions and deletions, then rebalancing after and update is amortized constant?

Come up with a potential for the mixed case and prove that rebalancing after an update is amortized constant.

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

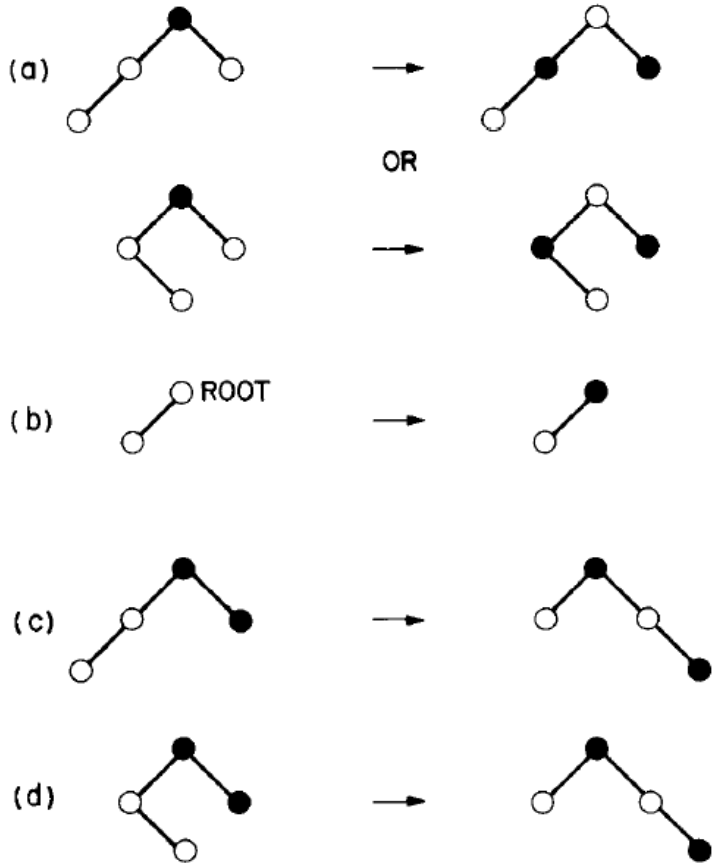

FIGURE 4. The Rebalancing Transformations in Red-Black Tree Insertion. Symmetric cases are omitted. Solid nodes are black; hollow nodes are red. All unshown children of red nodes are black. In cases (c) and (d) the bottommost black node can be missing.

Figure 1: Rebalancing operations after an insertion.

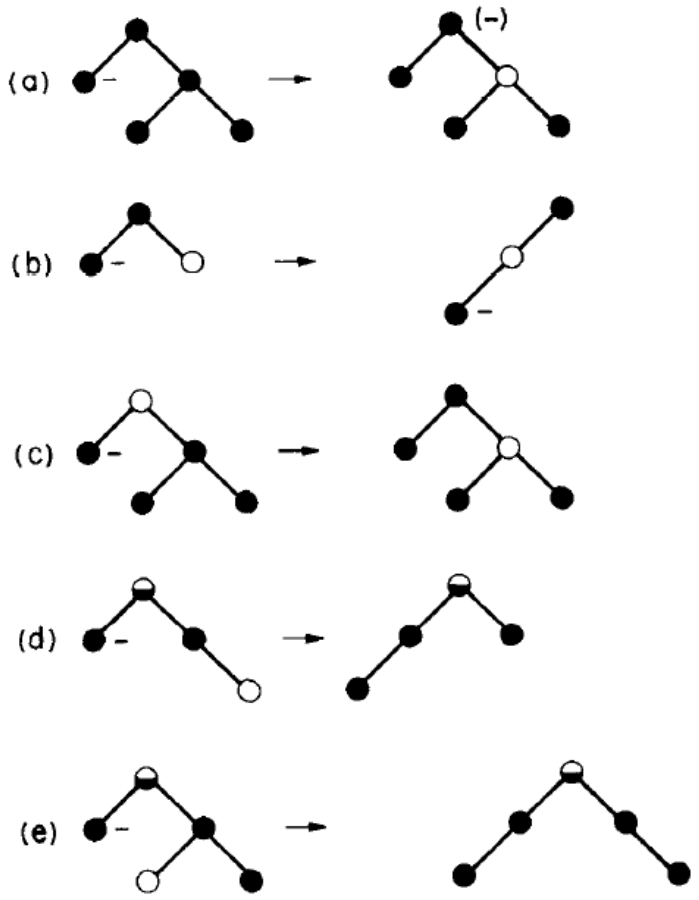

FIGURE 5. The Rebalancing Transformation in Red-Black Tree Deletion. The two ambiguous (half-solid) nodes in (d) have the same color, as do the two in (e). Minus signs denote short nodes. In (a), the top node after the transformation is short unless it is the root.

Figure 2: Rebalancing operations after a deletion.

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] Robert Endre Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF regional conference series in applied mathematics*. SIAM, Philadelphia, 1983.