Assignment 1 — *Applied Algorithms, T. II/2022–23*
(due: mon feb 06 @ 11:59pm)

Ground Rules: Do all problems below. Solve them either by yourself or in teams. For written questions, please typeset your answers. Everything will be handed in on Canvas. You can look up things on the Internet; refrain from copying solutions straight-up.

Most of the problems weren't designed with gradeability in mind. They are generally open-ended. Their main function is for you to have fun and think more deeply about the topic. Therefore, you are encouraged to go above and beyond what is required.

Problem 1. *Resizable Arrays.* Resizable arrays are traditionally implemented using geometric expansion and shrinking. It is simple to code up but has $O(n)$ worst-case complexity. In class, we looked at Sitarski's Hashed Array Tree (HAT) data structure. You'll investigate how HAT fares in this problem.

Our goal is to characterize performance differences between geometric expansion/shrinking and Sitarski's HAT. The following are dimensions of interest:

(a) Append latency—how long does each append take?

(b) Access latency—how long does it take to access an index?

(c) Scan throughput—how long does it take to read every element of the array from left to right (this is known as a scan pattern)?

(d) Overall throughput–how long does it take to insert N elements?

(*Hint:* You'll need `rdtsc` to study per-operation latency.)

Problem 2. *Space Usage of Skip Lists.* In class, we saw skip lists constructed using an unbiased coin–that is, a key "grows" one level higher when the coin turns up heads and stops growing when the turn turns up tails.

(i) Prove a sharp, high-probability space bound for a skip list with n keys. (*Hint:* The overall space requirement is $\Theta(n)$ with high probability.)

(ii) Suppose we use a p-biased coin instead (i.e., with probability p, the coin turns up heads). How does this affect the running time (search) and space? What is the best value of p?

Problem 3. *Skip Lists.* There are two subtasks here.

(a) Let's implement the skip list and compare it against any reputable built-in ordered map. Please design your own experiments. What are some questions you would like to know and how would you experimentally answer them?

(b) Searching in a skip list typically starts from the top-left corner. In this way, a search takes $O(\log n)$ whp., where n is the number of keys stored in the skip list. If we keep bi-directional pointers for each vertical column, we will be able to go up and down any column freely. This allows us to start from the smallest key and work our way to the desired key.

You'll extend the skip list in the following way. Maintain a "pointer" to the smallest key. Make up/down pointers bi-directional. Design and implement a search algorithm that runs in $O(\log d)$ time, where d is the number of elements smaller than the key you're searching for.

Problem 4. (a, b) *tree.* We'll work with a $(2, 3)$ tree.

(a) Starting with an empty tree, insert the following keys (in this order):

733, 703, 608, 846, 309, 269, 55, 745, 548, 449, 513, 210, 795, 656, 262

Show the resulting tree.

(b) Show the steps that happen when we delete 309 from this tree.

Problem 5. *B-Tree Speed.* Traditionally, balanced binary search trees (BBSTs) are the default choice for in-memory operations and B-trees are reserved (exclusively) for disk-resident data. For in-memory scenarios, the obvious downside of the B-tree is that it is more bloated than its BBST counterparts. However, B-tree is known to play well with the memory hierarchy (caches and transfers between disk and memory).

To narrow down the design space, we'll use $a = \lceil b/2 \rceil$—that is to say, your (a, b) tree will be a typical B-tree. For a large number of keys (say, tens of million keys), what is the best b for performance? Using this "optimal" b, how does it compare to a typical ordered-map in terms of space usage and speed?