University of Waterloo
CS240E, Winter 2021
Assignment 2

Due Date: Wednesday, February 10, 2021 at 5pm
(Programming Due Date: Wednesday, February 24, 2021 at 5pm)

The integrity of the grade you receive in this course is very important to you and the University of Waterloo. As part of every assessment in this course you must read and sign an Academic Integrity Declaration (AID) before you start working on the assessment and submit it **before the deadline of February 10** along with your answers to the assignment; i.e. **read, sign and submit A02-AID.txt now or as soon as possible**. The agreement will indicate what you must do to ensure the integrity of your grade. If you are having difficulties with the assignment, course staff are there to help (provided it isn't last minute).

The Academic Integrity Declaration must be signed and submitted on time or the assessment will not be marked.

Please read `http://www.student.cs.uwaterloo.ca/~cs240e/w21/guidelines/guidelines.pdf` for guidelines on submission. **Each written question solution must be submitted individually to MarkUs as a PDF** with the corresponding file names: a2q1.pdf, a2q2.pdf, ... , a2q5.pdf .

It is a good idea to submit questions as you go so you aren't trying to create several PDF files at the last minute. **Remember, late assignments will not be marked but can be submitted to MarkUs after the deadline for feedback if you email cs240e@uwaterloo.ca and let the ISAs know to look for it. (#3).**

For all questions, you may use facts from calculus, but review briefly the name of the fact and exactly what it is stating. (Refer to how the notes review concavity or l'Hôpital's rule for examples.)

1. (6 marks) Let $A[0..n-1]$ be an unsorted array that stores integers, where each entry is in the range $[0, n^5)$. Entries in A are not necessarily unique. Show how to test whether there are indices i and j such that $|A[i] - A[j]| = 10$. The run-time and the auxiliary space must be in $O(n)$.

2. (5 marks) Assume that you are given a list L that contains n key-value pairs in sorted order. Show how to build an AVL-tree that contains the key-value pairs from L. The run-time and the auxiliary space should be $O(n)$.

3. (3+4 marks) Recall that the `Selection` problem receives as input a set of n items and an integer k with $0 \le k \le n-1$ and it must return the item that would be at $A[k]$ if the items were put into an array A in sorted order.

   a) Argue that any comparison-based algorithm for the Selection problem on n items must have $\Omega(\log n)$ worst-case time.

b) Let T be a scapegoat tree that stores n items. Argue that `Selection`(T, k) can be done in $O(\log n)$ time.

4. (2+5 marks) Consider a version of binomial heaps that has the same structure and order properties, but the following implementation of the operations for a binomial heap P:

   - $P.\mathit{merge}(P')$: Append the list of P' to the one of P. This takes $O(1)$ time.
   - $P.\mathit{insert}(x)$: Create a single-node binomial heap P' that stores x and then call $P.\mathit{merge}(P')$. This takes $O(1)$ time.
   - $P.\mathit{deleteMax}$: First, make P proper with the same routine as in class. Then perform *deleteMax* as in class, i.e., find the maximum among the roots (in the list L of flagged trees), remove the tree T that contains the maximum x_0 from L, split $T \setminus \{x_0\}$ into flagged trees, let P' be the binomial heap with these flagged trees and call $P.\mathit{merge}(P')$. This takes $O(|L| + \log n)$ time, where n is the number of items and $|L|$ is the length of the list at the beginning of the operation.

   a) Show that *deleteMax* has worst-case run-time in $\Theta(n)$.

   b) Argue that with this implementation, the *amortized* run-time is in $O(1)$ for insert and in $O(\log n)$ for deleteMax. Using a potential function is recommended; coming up with a suitable one is part of the assignment.

   (You do not need to analyze the amortized run-time of *merge* as a stand-alone operation, though of course you do need to analyze its effects if it is called within *insert* or *deleteMax*.)

5. (2+2+4 marks) Suppose you implement skip lists with a biased coin flip: the probability that the tower-height is increased is 3/4. Put differently, $P(X_k \geq i) = \left(\frac{3}{4}\right)^i$ (where X_k denotes the tower-height of key k).

   a) What is the expected height of the tower of key k? Give a closed-form solution (no summations) and an exact bound (no asymptotics).

   b) What is the expected length of list S_i (for $0 < i < h$)? Give a closed-form exact bound.

   c) What is the expected height of the skip list? Give a closed-form solution of the form $c \cdot \log n + d$. Points will be deducted if c is bigger than it needs to be. You may assume that n is divisible as needed.

6. (21 marks, Programming, due Feb 24)

   You work in a software company, and your boss is well-acquainted with binary search trees. He recently learned that those can be made to perform better with by balancing, and is now wondering which of the various methods is best. Your job is to implement four versions of binary search trees (AVL-trees, scapegoat trees, binary search-trees with MTF-heuristic and splay trees) and to compare their performance with each other as well as with regular binary search trees.

Details:

- We will provide you with classes `TreeNode` and `BST`, which implement binary search trees. We will use our own copies of these classes during auto-testing, so any changes you make to it should only be for testing-purposes.
- We will provide you with a stub for four classes `AVLTree`, `ScapegoatTree`, `MTFTree` and `Splay` that are extension of `BST`. You need to fill in the details for *insert* for all four classes. Submit file `BSTExtensions.cpp`.
- Submit file *experiment.pdf* that contains a recommendation to your boss as to which implementation you would suggest. Give experimental evidence to support your recommendation, and briefly describe how you conducted your experiments. (The expected total length is around half a page.)

Evaluation (21 marks total):

- 4 marks each for correctness of implementing *insert* in the classes `AVLTree`, `ScapegoatTree`, `MTFTree` and `SplayTree`. This will be mostly determined by automated testing of the resulting structure.
- 5 marks for a recommendation that is well-supported by evidence. There is no one correct answer to this question; your recommendation will be evaluated based on doing reasonable experiments and drawing reasonable conclusions.

A few notes and hints:

- The stubs were written to make your life easier and do not necessarily follow best programming practices (e.g. everything is public, no accessor functions, multiple classes in a file). You are likewise permitted to keep everything public, and all your submitted classes shouild be in one file.
- Class `BST` has a helper-method `print`, which displays a binary search tree in ASCII—you may find this useful for testing.
- Class `BST` also has helper-methods `inOrder` and `preOrder`, which we will use during auto-testing. They do not use parent-references. (Put differently, you need not update parent-references during rotations/rebuilds if you prefer to find the search-path differently.)
- Class `TreeNode` has an integer `field`, which is unused by `BST`. You can use it to store height, size, or whatever else you might find useful.
- For all classes except `ScapegoatTree` the tree for a fixed insertion-order is unique. To make it unique for `ScapegoatTree` as well, use $\alpha = \frac{2}{3}$, and use the rule that when re-building a subtree where n_p is even, the left subtree receives $(n_p - 2)/2$ nodes while the right subtree receives $n_p/2$ nodes.

- We have not seen details of how to build a perfectly balanced tree in $\Theta(n_p)$ time. Our testing-scripts only test for structure (as long as the code returns within a reasonable timeframe), so rebuilding in $\Theta(n_p \log n_p)$ time is acceptable (and should be easy).
- You are allowed to use `std::vector`, `std::list`, `std:stack` and `std:queue` in the insert-routines, but no other external implementations. (You can use whatever you want in your own main-file where you do the experiments—we do not get that code.)
- Nearly all your trees require rotations. Implementing rotations only once (i.e., having a common superclass for those trees) is strongly recommended.
- The pseudocodes in the textbook were written for easy understanding, but are not necessarily the easiest for implementation. You are welcome to deviate from what exactly you update when, as long as the final tree has the same structure.
- Your file will be translated with `g++ -std=c++17 -c BSTExtensions.cpp` (and then linked with our own copies of `BST` and main-routine). Therefore your file `BSTExtensions.cpp` is not allowed to have a main routine, or its main routine should be surrounded by `#ifndef TESTING` as done in the stub.
- We would suggest that you run your program on randomly built trees of various sizes. While you could use the run-time as evaluation-factor, this is notoriously unreliable due to hardware, load, swapping, very small numbers etc. It also does not take into account how long future searches might take. A better idea is to consider height, and/or number of rotations, and/or size of rebuilt subtrees, and/or total access cost, or ... (this decision is up to you).