University of Waterloo
CS240E, Spring 2026
Written Assignment 2

Due Date: Tuesday, June 9, 2026 at 5:00pm

Be sure to read the assignment guideliness (`https://student.cs.uwaterloo.ca/~cs240e/s25/assignments.phtml#guidelines`). Submit your solutions electronically to Crowdmark. Ensure you have read, signed, and submitted the **Academic Integrity Declaration** AID01.

Grace period: submissions made before 7:59pm on June 9 will be accepted without penalty. Please note that submissions made after 7:59pm **will not be graded** and may only be reviewed for feedback.

Question 1 [1+1+7=9 marks]

Let $T(w)$ be the number of times that we print a '*' when calling Algorithm 1 with a bitstring w of length n. Now let $T^{\text{best}}(n)$, $T^{\text{worst}}(n)$, and $T^{\text{avg}}(n)$ be the best, worst, and average of this (over all bitstrings of length n).

a) Prove an O-bound on $T^{\text{best}}(n)$.

b) Prove an Ω-bound on $T^{\text{worst}}(n)$.

c) Show that $T^{\text{avg}}(n) \in O(T^{\text{best}}(n))$.

Algorithm 1: *silly-algo*(w, n)

Input: bitstring w of length n

```
1 i ← 0, while w[i] == 0 do i ← i + 1
  // w[i] is leftmost non-zero, possibly $
2 for k = 1 to (i+1)^2 do print "*"
```

Hint: If you find part (c) difficult, then for partial credit state a summation-formula for $T^{\text{avg}}(n)$, simplify it as much as possible, and give as tight an upper bound for it as you can manage. You may use without proof that $x \geq 4 \log x$ for $x \geq 16$. The course-notes have bounds for some summations that you may use without proof (but state the page-number).

Question 2 [6+6=12 marks]

Consider the following algorithm to find the minimum in a binary search tree.

Algorithm 2: *findMin*(root r)

```
1 if (r is null) then return "empty tree"
2 while r.leftChild != null do r ← r.leftChild
3 return r.key
```

Let $T^{\text{avg}}(n)$ (for $n \geq 0$) be the average-case number of executions of the while-loop in *findMin* for a tree with n nodes. Here the average is taken over all binary search trees that store $\{0, \ldots, n-1\}$, and $T^{\text{avg}}(0) = T^{\text{avg}}(1) = 0$.

a) Show that for $n \geq 2$ we have $T^{\text{avg}}(n) \leq 1 + \frac{1}{C(n)} \sum_{i=0}^{n-1} C(n-i-1)C(i)T^{\text{avg}}(i)$, where $C(n)$ is the number of binary search trees that stores $\{0, \ldots, n-1\}$.

Note: the expected level of rigour is to be as precise as we were with our average-case time examples in class (Section 1.5 in the notes).

b) Show that $T^{\text{avg}}(n) \in O(\log n)$.

Note: it is recommended to show $T^{\text{avg}}(n) \leq 2 \log n$, and that you consider a 'good case' where the left subtree has size at most $n/2$.) You may use without proof that $C(n) = \sum_{i=0}^{n-1} C(i) \cdot C(n-i-1)$, and you may assume that n is divisible as needed.

Note: if you choose to use induction, the expected level of rigour is as in our bound on the expected length of a random downward walk in a binary tree (Lemma 2.3 in the notes).

Aside: recall that the average height of binary search trees is in $\Theta(\sqrt{n})$, meaning we *cannot* argue the bound on $T^{\text{avg}}(n)$ via the average height.

Question 3 [6+3=9 marks]

a) Assume that you are given a non-empty linked list L that contains n key-value pairs in sorted order. Design an algorithm to build a binary search tree that contains exactly the items of L and that is *perfectly size-balanced*, i.e., for every vertex z we have $|size(z.left) - size(z.right)| \leq 1$. The run-time must be $O(n)$ and the algorithm should use $O(\log n)$ auxiliary space (i.e., space other than the input linked list L and the output-tree T that you are building).

b) Prof. Quirky thinks that he can do the above with a comparison-based algorithm even if list L is in unsorted order. Show that this is not possible.

Question 4 [2+3+2+4+3+4=18 marks]

Motivation: Scapegoat trees as defined in class store at each node z the size of the subtree rooted at z. This is not actually required; this assignment will guide you towards a variation that operates without storing the size of the subtree.

Define a *light scapegoat tree* to be a binary search tree that is *height-balanced*, by which we mean that the height is at most $\lfloor \log_{4/3} n \rfloor$ (or equivalently, every node has depth at most $\lfloor \log_{4/3} n \rfloor$). You may assume that a light scapegoat tree has a field *size* with its number of items. However, a light scapegoat tree does not store its height, and a node knows *nothing* except its key and left and right subtree. (In particular it knows neither its depth, nor its parent, nor the size of its subtree, nor the height of its subtree.)

Algorithm 3 below gives (an incomplete) pseudo-code to insert in such a tree:

a) Show that (after inserting the new leaf z) the tree can be height-unbalanced only if the depth of z is too big.

Algorithm 3: *LightScapegoatTree::insert*(k, v)

```
   // Current tree is height-balanced
1  z ← BST::insert(k, v)
2  if height-unbalanced(z) then
3  |  p ← lowest-small-ancestor(z)
4  |  completely rebuild the subtree at p as a perfectly size-balanced tree
```

b) Design algorithm *height-unbalanced*(z) to test whether the tree is now no longer height-balanced. The run-time must be $O(\log n)$, where n is the current size of the tree.

c) Show that if the tree is not height-balanced after *BST::insert*, then there exists an ancestor x of leaf z such that

$$size(x) < \left(\tfrac{4}{3}\right)^{d(x,z)}.$$

Here $size(x)$ denotes the number of items in the subtree rooted at x, and $d(x, z)$ denotes the distance from z to x, i.e., the number of levels that x is above z. (So $d(z, z) = 0$, $d(parent(z), z) = 1$, etc.).

d) Design algorithm *lowest-small-ancestor*(z). This must achieve the following: You are given the newly inserted leaf z, and you know that the tree is now not height-balanced. You must find an ancestor x of z such that $size(x) < \left(\frac{4}{3}\right)^{d(x,z)}$. Furthermore, your x must be the lowest ancestor of z that satisfies this, i.e., must minimize $d(x, z)$. The run-time must be $O(size(x) + \log n)$.

e) Suppose now that you have access to the subroutine *lowest-small-ancestor*(z) from the previous question that does the following: find an ancestor x of z with $size(x) < \left(\frac{4}{3}\right)^{d(x,z)}$, and among all those, return the one that minimizes $d(x, z)$.

Argue that the rest of the insertion-routine is correct, that is, show that after rebuilding the subtree (at the node p returned by *lowest-small-ancestor*) to be perfectly size-balanced, the resulting binary search tree is height-balanced.

f) Let p be the node returned by *lowest-small-ancestor*. Show that before rebuilding the subtree at p we had

$$|size(p.left) - size(p.right)| \in \Omega(size(p)).$$

For all parts, you may use results of previous parts even if you did not prove them.

Continuing the "motivation": We are *not* asking you to show that *LightScapegoatTree::insert* has $\Theta(\log n)$ amortized time, but with the above results it would be very easy to do so using the potential function from class.