601.433/633 Introduction to Algorithms **Fall 2025**
Homework #3 **Due:** October 13, 2025, 11:59pm

Remember: you may work in groups of up to three people, but must write up your solution entirely on your own. Collaboration is limited to discussing the problems – you may not look at, compare, reuse, etc. any text from anyone else in the class. Please include your list of collaborators on the first page of your submission. You may use the internet to look up formulas, definitions, etc., but may not simply look up the answers online.

Please include proofs with all of your answers, unless stated otherwise.

1 More counters (14 points)

We saw in class that if we have a binary counter which we increment n times the total cost (measured in terms of the number of bits that are flipped) is $O(n)$, i.e. the amortized cost of an increment is $O(1)$. What if we also want to be able to decrement the counter? Throughout this problem we will assume that the counter never goes negative – at every point in time the number of increments up to that point is at least as large as the number of decrements.

Show that it is possible for a sequence of n operations (increments and decrements) to have amortized cost of $\Omega(\log n)$ per operation (so the total cost is $\Omega(n \log n)$). This should hold even if we start from 0 and the counter never goes negative.

2 Heaps (13 points)

(a) (5 points) Draw all possible binary min-heaps on the keys $\{1, 2, 3, 4, 5\}$. (You may hand-draw if you want to, but please make sure your drawings are legible).

(b) (8 points) How many different binomial heaps are there on the keys $\{1, 2, 3, 4, 5, 6\}$? Prove your answer.

3 Range Queries (13 points)

We saw in class how to use B-trees as dictionaries, and in particular how to use them to do *insert* and *lookup* operations. Some of you might naturally wonder why we bother to do this, when hash tables (which we will talk about later) already allow us to do this. While there are many good reasons to use search trees rather than hash tables, one informal reason is that search trees can in some cases be either used directly or easily extended to allow efficient queries that are difficult or impossible to do efficiently in a hash table.

An important example of this is a *range query*. Suppose that all keys are distinct. In addition to being able to insert and lookup (and possibly delete), we want to allow a new operation *range*(x, y) which is supposed to return the number of keys in the tree which are at least x and at most y.

In this problem we will only be concerned with 2-3-4 trees (B-trees with parameter $t = 2$). Given a 2-3-4 tree with n elements, show how to implement *range*(x, y) in $O(\log n + k)$ time, where k is the number of elements that are at least x and at most y. Prove that your solution is correct and that it has the appropriate running time.

4 Union-Find (30 points)

In this problem we'll consider what happens if we change our Union-Find data structure to *not* use path compression. We will still use union-by-rank, but Find operations will no longer compress the tree. More formally, consider the following tree-based data structure. Every element has a parent pointer and a rank value.

Make-Set(x): Set $x \to parent := x$ and set $x \to rank := 0$.

Find(x): If $x \to parent == x$ then return x. Else return $Find(x \to parent)$.

Union(x, y):

Let $w := Find(x)$ and let $z := Find(y)$.

If $(w \to rank) \geq (z \to rank)$ then set $z \to parent := w$, else set $w \to parent := z$.

If $(w \to rank) == (z \to rank)$, set $(w \to rank) := (w \to rank) + 1$

In this problem we will analyze the running time of this variation.

(a) (10 points) Recall that the height of any node x is the maximum over all of the descendants of x of the length of the path from x to that descendant. Prove that for every node x, the rank of x is always equal to the height of x. Hint: use induction.

(b) (10 points) Prove that if x has rank r, then there are at least 2^r elements in the subtree rooted at x (we did this in class for the more complicated data structure which uses path compression, but now you should do it for this version without path compression).

(c) (10 points) Using the previous two parts, prove that every operation (Make-Set, Union, and Find) takes only $O(\log n)$ time (where n is the number of elements, i.e., the number of Make-Set operations).

5 Hashing (30 points)

Let $H = \{h_1, h_2, \dots\}$ be a collection of hash functions, where $h_i : U \to \{0, 1, \dots, M-1\}$ for every i and we assume that $|U| = 2^u$ and that $M = 2^b$ (the same setup as in class when we designed a universal hash family). Recall that H is a *universal hash family* if $\mathbf{Pr}_{h \sim H}[h(x) = h(y)] \leq 1/M$ for all $x, y \in U$.

Consider the following, slightly different definition. We say that H is a 2-*universal hash family* if $\mathbf{Pr}_{h \sim H}[h(x) = a \wedge h(y) = b] \leq 1/M^2$ for all $x, y \in U$ with $x \neq y$ and $a, b \in \{0, 1, \dots, M-1\}$.

(a) (10 points) Prove that any 2-universal hash family is also a universal hash family.

(b) (10 points) Prove that for every u and b with $u > b \geq 1$ there is some universal hash family from U to $\{0, 1, \dots, M-1\}$ (with $|U| = 2^u$ and $M = 2^b$) which is *not* a 2-universal hash family. Hint: think about the constructions from class and the textbook.

(c) (10 points) Give a universal hash family from $U = \{0, 1, 2, 3, 4, 5, 6, 7\}$ to $\{0, 1\}$ that contains at most four functions (and prove it is universal). Is this also a 2-universal hash family? Why or why not?