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.

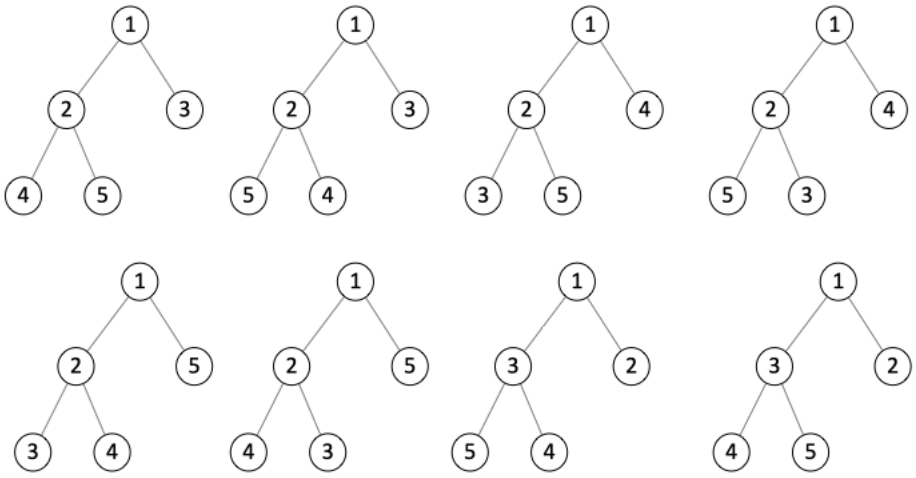
Solution: Let $m = \lfloor \log(n/2) \rfloor$. Note that $n/4 \leq 2^m \leq n/2$. We first do 2^m increments (starting from 0). After this, the counter has a 1 in the $(m+1)$st bit, and a 0 in the other m bits. For the remaining $n - 2^m \geq n/2$ operations, we alternate decrements and increments.

Note that in each of the final $n - 2^m$ operations we have to flip all $m + 1$ bits. Thus the total cost for all n operations is at least

$$\sum_{i=1}^{n-2^m} (m+1) \geq \sum_{i=1}^{n/2} \lfloor \log(n/2) \rfloor \geq \Omega(n \log n)$$

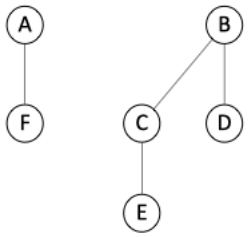
2 Heaps (13 points)

(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). **Solution:**



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

Solution: 45. A binomial heap on $\{1, 2, 3, 4, 5, 6\}$ must have exactly one B_1 and one B_2, since it has 6 elements. Thus it has the following structure:



There are $\binom{6}{2} = 15$ ways of filling in nodes A and F (any two elements can be in those nodes, but for any two elements the smaller one must be in A). For each such choice of keys in A,F, the key in B is fixed (it must be the smallest of the remaining elements). Then there are three ways of filling in D, and for each such choice, C and E are fixed. Thus overall there are $15 \times 3 = 45$ possible binomial heaps.

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.

Solution. We can perform a lookup on x to find the node in the tree containing x, and then do an inorder traversal of the tree starting from x (skipping the initial call to the left subtree of x)

until we reach the node containing y. We will simply keep track of the number of keys that we see in this traversal (including x and y), and output this number.

To see that this is correct, note that by the definition of an inorder traversal we see precisely the nodes with keys at least x and at most y. Hence by returning the number of nodes we see, we will be returning the correct answer to the range query.

For the running time, clearly the lookup takes $O(h)$ time, and the inorder traversal takes $O(1)$ time per node in the traversal. Since we will see precisely k nodes in the traversal, the running time of the traversal will be $O(k)$. Hence the total running time is at most $O(h + k)$.

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.

Solution: We prove this by induction. For the base case, it is easy to see that it is initially true: after the first call to Make-Set we have a single node with rank 0 and height 0. For the inductive step, assume that this is true at some point in time and then we execute another operation. If this operation is a Find it has no effect on the ranks or the trees (since we are not using path compression), so all nodes still have rank = height. Similarly, if the operation is a Make-Set then it is true of the new tree (for the same reason as in the base case), and nothing has changed about the other nodes so it is still true for all nodes.

Now suppose that the operation is a Union. In particular, suppose that it is Union(x, y), and let w be the root of the tree containing x and let z be the root of the tree containing y. Note that the only node whose height might have changed is the whichever of z and w becomes the root of the new tree, so we just need to prove that the height of this node is equal to its rank. Let $r_w = w \to rank$ and let $r_z = z \to rank$. If $r_w > r_z$ then w becomes the new root with rank r_w. Its height will be $\max\{r_w, r_z + 1\}$, since it will have a path of length r_w to one of its descendants (by the inductive hypothesis) and a path of length $r_z + 1$ to one of z's descendants (by the inductive hypothesis). Since $r_w > r_z$, this implies that the height is equal to the rank, as they are both r_w. Similarly, if $r_z > r_w$ then z becomes the new root with rank r_z and height $\max\{r_z, r_w + 1\} = r_z$.

So all that remains is to analyze the case of $r_w = r_z$. In this case, w becomes the new root with rank $r_w + 1$. By the inductive hypothesis, its new height is equal to $\max\{r_w, r_z + 1\}$. Since $r_w = r_z$, this is equal to $r_w + 1$ and thus equal to the rank, as claimed.

(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).

Solution: This is just a simpler version of the same analysis that we did in class. We prove this by induction. Initially, after the first Make-Set, there is one element with rank 0 and its subtree has exactly $1 = 2^0$ elements in it. Now suppose it is true at some point in time, and consider doing another operation. A Find does not change the trees or the ranks, so it would still be true. A Make-Set would maintain the property for the same reason as in the base case.

Now suppose we do Union(x, y). The only node whose rank or subtree changes is the new root (either z or w depending on the ranks). Let $r_z = z \to rank$ and let $r_w = w \to rank$. If $r_w > r_z$ or $r_z > r_w$ then no node changes ranks and the size of each subtree does not decrease, so by the inductive hypothesis it is still true. If $r_w = r_z$, then the new rank of w is $r_w + 1$ and by the inductive hypothesis the size of its subtree is at least $2^{r_w} + 2^{r_z} = 2^{r_w} + 2^{r_w} = 2^{r_w+1}$, as claimed.

(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).

Solution: Make-Set clearly takes $O(1)$ time. The running time of each Union is $O(1)$ plus the time to do two Find operations. Hence we need only prove that the running time of a Find is $O(\log n)$. Part b clearly implies that every node has rank at most $\log n$ (since there are only n elements total), so this together with part a implies that every node has height at most $\log n$. Since the running time of a Find operation is at most the height of a tree (since it is just a walk up to the root), we get that the running time of a Find is $O(\log n)$.

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.

Solution: Let H be a 2-universal hash family. Let $x, y \in U$ be two arbitrary elements. For each outcome $a \in \{0, 1, \ldots, M-1\}$, let Z_a be the event that $h(x) = h(y) = a$, and let Z be the event that $h(x) = h(y)$. Note that by the definition of a universal hash family, $\mathbf{Pr}_{h\sim H}[Z_a] \leq 1/M^2$ for all a. Clearly the events $\{Z_a\}$ are disjoint, and their union is exactly Z. Hence

$$\mathbf{Pr}_{h\sim H}[h(x) = h(y)] = \mathbf{Pr}_{h\sim H}[Z] = \sum_{a=0}^{M-1} \mathbf{Pr}_{h\sim H}[Z_a] \leq M \cdot \frac{1}{M^2} = \frac{1}{M}.$$

Since x and y were arbitrary elements, we get that H is 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 $\{0, 1, \ldots, M-1\}$ (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.

Solution: Let H be the universal hash family we constructed in class, i.e., the hash functions in H are random 0/1 matrices with the appropriate dimensions. Let x be the all 0's vector, and let y be any other element. Note that $h(x) = \vec{0}$ for all $h \in H$. Moreover, it is easy to see that $\mathbf{Pr}_{h\sim H}[h(y) = \vec{0}]$ is exactly $1/M$ (as we discussed in class, for this hash family we know that $\mathbf{Pr}_{h\sim H}[h(y) = a] = 1/M$ for all $a \in U$, as long as $y \neq \vec{0}$). Thus by setting $a = b = \vec{0}$ we get that

$$\mathbf{Pr}_{h\sim H}[h(x) = h(y)] = \mathbf{Pr}_{h\sim H}[h(y) = \vec{0}] = 1/M.$$

Since $1/M > 1/M^2$, this implies that H is not a 2-universal hash family.

(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?

Solution. We use the following four hash functions:

	0	1	2	3	4	5	6	7
h_1	0	0	0	0	1	1	1	1
h_2	0	1	0	1	0	1	0	1
h_3	0	0	1	1	0	0	1	1
h_4	0	1	1	0	1	0	0	1

It is easy to verify that any two columns differ in at least two entries. Thus the probability over the choice of a random hash function from the above family that two fixed elements will collide is at most $1/2 = 1/m$, and thus the family is universal. It is not 2-universal, as $\Pr_{h\sim H}[h(0) = 0 \wedge h(7) = 1] = 1$, which is not at most $1/M^2 = 1/4$.