601.433/633 Introduction to Algorithms **Fall 2024**
Homework #4 **Due:** October 3, 2024, 9am

Bella Lu
Partners: Jecia Mao, Luna Liu

1 Amortized Analysis (34 points)

In this problem we have two stacks A and B (recall that a stack allows us to push elements onto it and to pop elements off of it in LIFO order). In what follows, we will use n to denote the number of elements in stack A and use m to denote the number of elements in stack B. Suppose that we use these stacks to implement the following operations:

- PushA(x): Push element x onto A.
- PushB(x): Push element x onto B.
- BigPopA(k): Pop $\min(n, k)$ elements from A.
- BigPopB(k): Pop $\min(m, k)$ elements from B.
- Move(k): Repeatedly pop one element from A and push it into B, until either k elements have been moved or A is empty.

We are using the stacks A and B as black boxes – you may assume that PushA, PushB, BigPopA(1), and BigPopB(1) each take one unit of time (i.e., it takes one time step to push or pop a single element).
Use a potential function argument to prove that the amortized running time of every operation is $O(1)$.

Let the potential function be $c_1 \cdot n + c_2 \cdot m$, for some constant c_1 and c_2.

From the 5 operations, we can see that A has more expensive operations. Specifically, Move(k) repeatedly pop one element from A and push it into B, until either k elements have been moved or A is empty. This has a true cost of $2k$ every time, and can only be done from A to B. Since A causes more expensive operations, it should have more weight in the potential.

Let $c_1 = 3$ and $c_2 = 1$

Amortized analysis:

- PushA(x): 1 + 4 = 5 = O(1)
- PushB(x): 1 + 1 = 2 = O(1)
- BigPopA(k): k + (-3k) = -2k = O(1) since it is negative
- BigPopB(k): k + (-k) = 0 = O(1)

- MOVE(k): 2k + (-2k) = 0 = O(1)

$\therefore$ The potential function $3n+m$ proves that the amortized running time of every operation is O(1). ■

2 Amortized analysis of 2-3-4 trees (33 points)

Recall that in a 2-3-4 tree, whenever we insert a new key we immediately split (on our way down the tree) any node we see that is full (has 3 keys in it).

(a) (11 points) Show that in the worst case, the number of splits that we do in a single operation can be $\Omega(\log n)$. In other words, show that for there is a series of n inserts so that the next insert causes $\Omega(\log n)$ splits. You can be a bit informal here — just explain at a high level what such a sequence would look like and why it would result in $\Omega(\log n)$ splits.

To show that, in the worst case, the number of splits that we do in a single operation can be $\Omega(\log n)$, we can construct a tree that has full nodes along the right-most path. In such a tree, if we insert an element that is bigger than any other element currently in the tree, we will need to traverse the right-most path, causing $height = \Omega(\log n)$ amount of splits.

To build such a tree, we can follow the algorithm below:

1. Fill the right-most leaf node by inserting larger numbers than any other number currently in the tree.

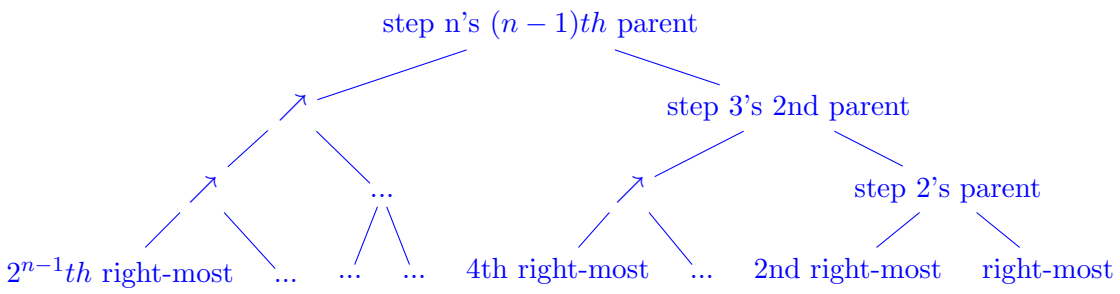
2. Keep filling the second right-most leaf node until its parent is a full node. We can do this by inserting numbers that are less than step 2's parent, but greater than step 3's 2nd parent (shown in graph below).

3. Keep filling the fourth right-most leaf node until its parent's parent is a full node. We can do this by inserting numbers that are less than step 3's 2nd parent, but greater than step 4's 3rd parent (shown in graph below).

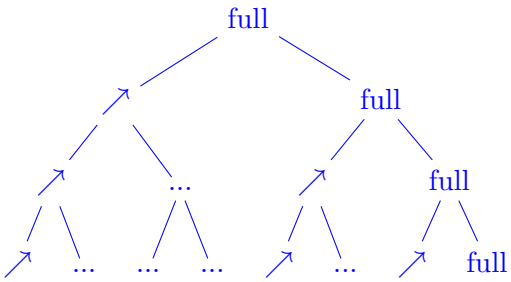
...

n. Keep filling the $2^{n-1}th$ node until its $(n-1)th$ parent is a full node. We can do this by inserting numbers that are less than all other numbers currently in the tree.

An illustration of the algorithm above:



The key here is that we start from the bottom of the path, and don't modify the node on the path after it becomes full. Since we can only insert at the leaf node, we can make the next node on the path full by starting at the leaf of the left sub-branch (which ensure that the already full node isn't modified), then keep on promoting until that top node is full. Here is a simpler diagram to illustrate:



Now, if we make an insert to the right-most bottom node (i.e. a number that is larger than all current numbers in the tree), we will do $\log n$ splits as we go down that path of all full nodes.

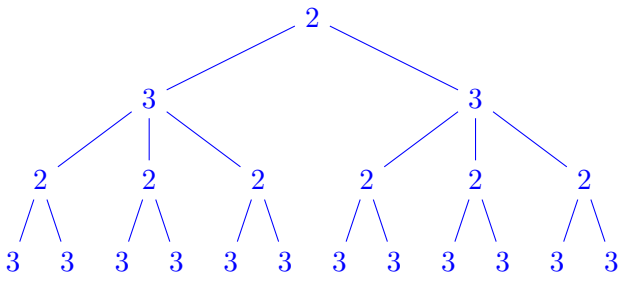
$\therefore$ In the worst case, the number of slits that we do in a single operation can be $\Omega(\log n)$. ■

Let's try to get around this worst-case bound by using amortized analysis. We will try to prove that the amortized number of splits is only $O(1)$. Note that this does not mean that the amortized running time of an insert is $O(1)$ (since this is not true); it just means that the amortized number of splits is $O(1)$. So think of "cost" not as "time", but as "number of splits". Since we didn't talk about them in class, feel free to assume there are no delete operations.

(b) (11 points) Since we only have to split a node when it's full, a natural approach is to have a bank on every node which works as follows. When a node is first created its bank is initialized to 0. When a node becomes full, we add 1 to its bank. Then when we split a node we use the 1 in its bank to pay for the split. In other words, a node's bank is 1 if the node is full and 0 otherwise.

This argument unfortunately does not work. Prove that this banking scheme *does not* imply that the amortized number of splits is $O(1)$.

Let's consider a tree where each level alternates having 2 keys and 3 keys. For simplicity, the number in each node of the graph below represents the number of keys in each node.



...

Actual work: Since every other level consist of full nodes, the number of splits we have to do after going down any path is $\frac{height}{2}$, which is the actual work.

Change in potential: Potential before splitting is $\frac{height}{2}$ because that is the number of full nodes. After all the splitting, each full node will become two nodes with 1 key each, and every node with two keys will become full nodes. The potential after splitting is still $\frac{height}{2}$, since there are an equal number of full nodes before and after splitting. Thus, the change in potential is $\frac{height}{2}$ - $\frac{height}{2}$ = 0.

Amortized split: (actual work) + (change in potential) = $\frac{height}{2}$ + 0 = $\frac{height}{2}$ = O($\log n$) $\neq$ O(1)

$\therefore$ We have proven that this banking scheme does not imply that the amortized number of splits is O(1). ■

(c) (11 points) Show how to modify this argument to work. That is, show how we can use a bank at each node to prove that the amortized number of splits is $O(1)$. Hint: the previous part shows that this bank cannot just equal 1 if the node is full and 0 otherwise.

We saw above that the problem with that potential function was that the potential of a node with 1 key was the same as a node with 2 keys. So, we can modify the potential function that distinguishes them.

Let a = number of full nodes
Let b = number of nodes with 2 keys
Let c = number of nodes with 1 key

Potential function = 2a + 1b + 0c

Types of split (numbers represent number of splits):

Case 1:

Actual work: 1
Change in potential: (2 + 0 + 0) - (1 + 2) = 2 - 3 = -1
1 + (-1) = 0 = O(1)

Case 2:

Actual work: 1
Change in potential: (1 + 0 + 0) - (0 + 2) = 1 - 2 = -1
1 + (-1) = 0 = O(1)

Case 3: (root)

Actual work: 1
Change in potential: (0 + 0 + 0) - (2) = 0 - 2 = -2
1 + (-2) = -1 = O(1)

Amortized number of splits in an insert operation:
Let number of splits = k. From the analysis of all different types of splits, we can see that the change in potential is $\leq -k$.

(number of splits) + (change in potential) + 1
$\leq$ k + (-k) + 1
$\leq$ 1
= O(1)

$\therefore$ We have proved that the potential function makes the amortized number of splits O(1). ■

3 Union-Find (33 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) (11 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.

We can induct on the operations. In other words, we can show that if it holds that the rank is equal to the height for x, then after any operation, the rank remains equal to the height.

Base case: If we make a new set, its rank will be 0 and its height will also be 0. The rank and height are equal, thus the base case holds.

For the sake of induction, let's assume that tree Y_1 has rank (r_1) = height (h_1).

Make-Set(x):

Follows exactly the base case.

Find(x):

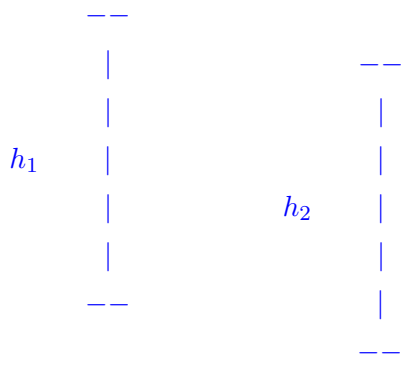
Find does not alter rank or height since there is no path compression. From our induction hypothesis, we assumed that they were the same, so not altering them makes them remain the same.

Union(x, y):

Case 1: Assume that another tree (Y_2, has rank r_2) we are unioning with Y_1 has the same height. (i.e. $r_1 = r_2$).

Let's look at the rank. By the definition of union, since $(Y_1 \to rank) == (Y_2 \to rank)$, set $(Y_1 \to rank) := (Y_2 \to rank) + 1$. In other words, r_1 increased by 1.

Now let's look at the height. When we union two trees, we put one's root as the child of the other's root.



From this diagram, we can see that h_2 is one lower than h_1, so the total height will be $h_2 + 1$. And since $h_1 = h_2$, h_1 will never exceed the bottom of h_2. Thus, the height of the new tree

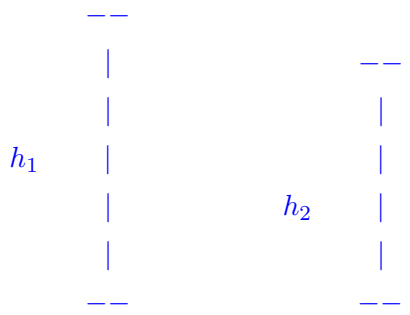
is $h_2 + 1$, or $h_1 + 1$.

Thus, both r_1 and h_1 increased by 1, so they stay the same, based on our initial induction hypothesis.

Case 2: Without loss of generality, assume that another tree (Y_2, has rank r_2) we are unioning with Y_1 has height less than Y_1. (i.e. $r_1 > r_2$).

Let's look at the rank. By the definition of union, since $(Y_1 \rightarrow rank) \geq (Y_2 \rightarrow rank)$, rank doesn't change.

Now let's look at the height. We can rewrite the expression as $h_1 \geq h_2 + 1$. We are putting the smaller tree's root as the child of the root of the larger tree. Since h_2 is at most the same as $h_1 - 1$, the height of the new tree will take on the height of the larger tree, which is h_1. Thus, height doesn't change as well.



Thus, both r_1 and h_1 stayed the same, so based on our initial induction hypothesis, they are still equal to each other.

$\therefore$ We have shown that after any operation, the initial inductive hypothesis still holds true. For every node x, the rank of x is always equal to the height of x. ■

(b) (11 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).

Again, we can induct on the operations. In other words, we can show that if it holds that there are at least 2^r elements in the subtree rooted at x that has rank r, then after any operations, there are still at least 2^r elements.

Base case: If we make a new set, its rank will be 0 and there are 1 element. Thus, $r = 0$, elements $= 2^0 = 1$. Base case holds.

For the sake of induction, let's assume that a tree Y_1 that has rank r_1, has at least 2^{r_1} elements.

Make-Set(x):

Follows exactly the base case.

Find(x):

Find does not alter rank (since there is no path compression) or the number of elements in the tree. From our induction hypothesis, we assumed that if Y_1 has rank r_1, it has at least 2^{r_1} elements. So, not altering them makes the statement still hold true.

Union(x, y):

Case 1: Assume that another tree (Y_2, has rank r_2) we are unioning with Y_1 has the same rank. (i.e. $r_1 = r_2$).

Let's look at the rank. By the definition of union, rank will increase by 1. New rank of Y_1 is $r_1 + 1$.

Let's look at the number of elements. Y_1 has at least 2^{r_1} elements, and Y_2 has at least 2^{r_2} elements. The new tree will have at least $2^{r_1} + 2^{r_2}$ elements. Since $r_1 = r_2$, the tree has at least $2^{r_1} + 2^{r_1} = 2 \cdot 2^{r_1} = 2^{r_1+1}$ elements.

After the union operation, the new rank is $r_1 + 1$ and has at least 2^{r_1+1} elements.

Case 2: Without loss of generality, assume that another tree (Y_2, has rank r_2) we are unioning with Y_1 has rank lower than Y_1. (i.e. $r_1 > r_2$).

Let's look at the rank. By the definition of union, rank will stay the same.

Let's look at the number of elements. Since the number of elements in a tree is always non-negative, adding the number of elements in Y_2 will obviously still keep the property that Y_1 has at least 2^{r_1} elements.

After the union operation, the new rank is still r_1 and still has at least 2^{r_1} elements.

$\therefore$ We have shown that after any operation, the initial inductive hypothesis still holds true. If x has rank r, then there are at least 2^r elements in the subtree rooted at x. ■

(c) (11 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).

Make-Set(x): Set rank to 0 and its parent to itself

These two steps make its running time O(1).

Find(x): Walk from x to the root, and return the root

The longest path we can take is when x is a leaf, is the height of the tree, which is O(height).

In part a, we found that rank = height, so O(height) = O(rank).

In part b, we found that the number of elements, $n \geq 2^{rank}$. After algebraic manipulation, that becomes $rank \leq \log n$, so O(rank) = O($\log n$).

Since O(height) = O(rank) = O($\log n$), Find(x) takes O($\log n$).

Union(x, y): Link(Find(x), Find(y))

The union function finds the root of x, then the root of y, then links the two. We found that the Find operation takes O($\log n$), so finding twice is also O($\log n$). Linking takes constant time.

Thus, two Finds plus constant linking is O($\log n$).

$\therefore$ We have proven that every operation takes only O($\log n$) time. ■