Dan Ports
6.046 PS 3-1

22/25

Yoav 2PM
Collab: Ø

a. unsorted array:
search(x): Θ(n) ✓
add: Θ(1) ✓
delete: Θ(1) ✗ replace the element to be deleted w/ the last element in the array, and shorten the array by 1. ✓
find most important: Θ(n) ✓

4

b. sorted array
search: Θ(lg n) ✓
add: Θ(n) ✓
delete: Θ(n) ✓
find most important: Θ(1) ✓

4

c. multiple arrays
search: $\Theta(\lg^2 n)$ Why?
add: The worst case is when the structure contains $2^k - 1$ elements for some k; at this point the binary representation is all ones, and so each of the $\lceil \lg(n+1) \rceil$ arrays is filled. In general, adding

8

new element and all previous arrays into the next array; all preceding arrays can then be emptied. In the worst case, this means each of the $\lg(n+1)$ arrays must be merged into one large array. This takes $\Theta(n)$ time. ok if $O(n)$ or n? we first merge the new element with the length-1 array, then merge the resulting 2-element array with the length-2 array, and so on. Since merging takes linear time, the total time required is $2 \cdot \sum_{i=0}^{\lg n} 2^i = 2 \cdot \frac{2^{\lg n+1}-1}{2-1}$ $= 2n - 2 = \Theta(n)$.

In the amortized case, we insert into the arrays in the following pattern:

1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 16... — what does it mean?

Assume the worst case that n is a power of Two. Note that half of the merges are into the first array, and half of the others into the second, and so on. So, noting that merges are performed in linear time, performing n additions requires time $\sum_{i=0}^{(\lg n+1)} \frac{n}{2^{i+1}} \cdot 2^i = \sum_{i=0}^{\lg(n+1)} \frac{n}{2}$

$= \frac{n}{2} \lg(n+1) = \Theta(n \lg n)$.

Delete: One way to perform a deletion is to merge every element into one large array. We showed above that it could be done in $\Theta(n)$ time if every array was full,

compute the binary representation of n and determine which arrays need to be filled. For each of these arrays, we can fill them with sorted elements by moving the first k elements (let k be the size of the array that needs to be filled) from the large array. Every element needs to be moved once, so this step is also $\Theta(n)$. ✓

✓

Find most important task: $\Theta(\lg n)$. Need to find the maximum of the final elements of each array: getting the final element needs $\Theta(1)$ time per array, and finding the minimum is linear in the number of arrays, so $\Theta(\lg n)$.

d. Balanced BST

Search: $\Theta(\lg n)$ ✓

Add a task: $\Theta(\lg n)$. ✓

Begin by adding the new task to the tree using whatever algorithm is required (red-black-insert, e.g.). This takes $\Theta(\lg n)$. Then we need to compare the priorities of the new element and the old maximum element; if the new element has higher priority,

Delete task: $\Theta(\lg n)$. ✓

We remove the element from the tree in $\Theta(\lg n)$ time using red-black-delete or whatever algorithm is appropriate. If the element we just deleted was the maximum element (the pointer pointed to it), then we find the new maximum by using the maximum procedure, which runs in $\Theta(\text{height}) = \Theta(\lg n)$ time (since the tree is balanced), and set the pointer to that maximum

6

Find most important task: $\Theta(1)$ ✓

10/10

Dan Ports
6.046 PS3-2

Yoav 2PM
Collab: Ø

We will store the weights in a tree data structure where each node represents a number i, its weight $W[i]$, a pointer to a left subtree containing only numbers less than i, a right subtree containing only numbers greater than i, and the total weights of the left and right subtrees. Since we never add or remove nodes, we can initialize the tree to a balanced state by using red-black, AVL, or similar tree algorithms.

To modify the weight $W[i]$, we search the tree for the node representing i, note the old value of $W[i]$ stored in the node, and replace it with the new value $W[i]'$. Then we follow the chain of parent pointers until we reach the root, and at each stage, compare i to that node's value

to x's left-subtree-weight. If $i > x$, we do the same to the right subtree. This requires $\Theta(\text{height}) = \Theta(\lg n)$ time.

To generate a random number, we take the root of the tree. Call its value i, its left-subtree-weight x, its weight y, and its right-subtree-weight z. Let $t = x+y+z$, and $r = \text{rand}(t)$. If $r \le x$, then recursively apply this procedure to the left subtree; if $r > x+y$, recurse on the right subtree. Otherwise return i. This requires at most $\Theta(\text{height}) = \Theta(\lg n)$ recursions, and constant time per recursion, so $\Theta(\lg n)$.

Justification of correctness:

First, we prove the invariant that each node's left and right subtree weights are the sum of the weights of the nodes in the appropriate subtree. We show that the invariant is maintained by a Modify operation: assume it holds beforehand. Then we need to modify the weight of some node i from $W[i]$ to $W[i]'$. Every node is either an ancestor of node i - that is, i is in one of its subtrees -

by following the chain of parent pointers from node i to the root. For the non-ancestor nodes, i is not in a subtree, so the subtree weights do not need to be changed. These nodes are not reachable by parent pointers, so the algorithm doesn't touch them, and the invariant still holds for them. The nodes that are ancestors of i by definition contain i in either the left or right subtree, and we can check which by comparing the node's value to i as previously described. The algorithm will reach every ancestor node by following the parent pointer chain, and at each step it finds out which subtree i was in. This subtree weight needs to be decreased by W[i] and increased by W[i]+1, which is what the algorithm does, and it does it for every ancestor node, so the invariant holds for the entire tree.

When we initialize the tree, the invariant holds. We can initialize the structure by starting by generating the tree with every weight and subtree weight set to zero. In this case the invariant trivially holds. We then use the Modify algorithm to set W[1] to 1; the others are still zero, so we have the desired initial state, and the invariant holds.

We next use this invariant to show that the generate algorithm gives the desired behavior: it returns i with probability $W[i] / \sum_{j=1}^{n} W[j]$. Note that in order for i to be returned, the algorithm must follow a path from the root to the i node and then choose to stop at i. For a tree starting at node k, write k's left-subtree-weight $L[k]$, k's right subtree weight as $R[k]$, and k's weight $W[k]$. The algorithm is defined such that the probability of returning k is $\frac{W[k]}{T[k]}$; the probability of descending the left subtree is $\frac{L[k]}{T[k]}$ and $\frac{R[k]}{T[k]}$ for the right, where $T = L[k] + R[k] + W[k]$. The path that leads to i returning i can be considered a series of choices at each ancestor node of i. Each of these choices has probability $\frac{L[k]}{T[k]}$, $\frac{R[k]}{T[k]}$, or $\frac{W[k]}{T[k]}$ depending on whether the needed choice was to traverse one of the subtrees or stop and return. For all decisions except the final one, the choice will be to follow a subtree; assume w/o loss of generality that it is the left. So at some stage, we move from node k to its left child k'; The probability of this decision being made is $\frac{L[k]}{T[k]}$. The next decision that needs to be made will have probability of form $\frac{x}{T[k']}$. But $T[k'] = L[k]$ by the invariant since k' is

the probabilities of each decision that needs to be made for i to be returned, we have a telescoping product. The only factors that do not cancel will be the denominator of the first, $T[root]$, and the numerator of the final decision, which will be $W[i]$ since i was returned. So the probability of returning i will be $W[i]/T[root]$, $T[root]$ is the sum of the weight of the root node and the nodes in its left and right subtrees, which is the sum of all weights $\sum_{j=1}^{n} W[j]$. So the probability of returning i is $\frac{W[i]}{\sum_{j=1}^{n} W[j]}$, which is the desired probability.

Dan Ports
6.096 PS 3-3

4 hours 2PM
Collab: Ø

First, copy the input array A[1..n] to a new array and augment it with the original array index as in PS1-3: B[i] = (A[i], i).
Then sort the array B using radix sort on the numbers; we can do this because the elements are integers between 1 and n. This operation runs in time $\Theta(n)$.
Then we iterate through the array, comparing the first and last elements; if their sum is too small, the first element cannot be in a pair; if their sum is too large the last element cannot be. In pseudocode:

```
i = 1;   j = n;
while (i ≤ j)
{
    p = B[i].value + B[j].value
    if (p > x)
      j = j - 1
    if (p < x)
      i = i + 1
    if (p = x)
    { output (B[i].position, B[j].position)
```

Each iteration of the array will either increment i or decrement j, so j - i will decrease by 1. Within n iterations, j - i will reach 0; then i = j and the algorithm is complete. Each iteration of the loop requires constant time and runs $\Theta(n)$ times, so the algorithm is $\Theta(n)$.

(correctness proof omitted) → your description already gives and indication of correctness.

Dan Ports
6.046 PS 3-4

4/10

Hour 2PM
Collab: ∅

First, we will show that $E[n] = O(\sqrt{n})$

Let A be the event that no matches are found in the first $\sqrt{n}$ pulls, and note that

$$E[n] = \Pr\{A\} E[n|A] + \Pr\{\bar{A}\} E[n|\bar{A}]$$

Clearly the probabilities are bounded above by 1, and $E[n|\bar{A}] \leq \sqrt{n}$ because if A does not happen, by definition a matching pair was found in the first n pulls.

So $E[n] \leq E[n|A] + \sqrt{n}$.

Consider $E[n|A]$. This is the expected # pulls required until a matching pair is found.

The probability of finding a matching pair on any try is at least $\frac{\sqrt{n}}{2n-\sqrt{n}}$ because there are at least $\sqrt{n}$ socks that will match one already drawn and at most $2n-\sqrt{n}$ socks remaining. So the expected time until a match is found is less than $\left(\frac{\sqrt{n}}{2n-\sqrt{n}}\right)^{-1}$

$= 2\sqrt{n} - 1$, plus the $\sqrt{n}$ already drawn.

So $E[n|A] \leq 3\sqrt{n} - 1$, and

$E[n] \leq 3\sqrt{n} - 1 + \sqrt{n} = \Theta(\sqrt{n})$ ✓

(c)

Next we note that this problem can be described as sampling without replacement until a match is found. The expected number of samples until success is greater than it would be if we were sampling with replacement, because it would be possible to draw the same sock more than once with replacement. From 6.042 and the birthday problem, the expected tries until a match is known to be $\Theta(\sqrt{n})$ in sampling with replacement. So the expected number of socks pulled is bounded below by this and is $\Omega(\sqrt{n})$. Since we already showed it was $O(\sqrt{n})$, it is thus $\Theta(\sqrt{n})$

Show Math for B-Day