CS561 HW9

Ryan Scherbarth

November 2024

1. A cat hops on posts arranged in a circle. There are $2n$ posts, with n rad and n black. The cat can start at any post, and always hops to the next post in the clockwise direction, until it visits all posts. It "wins" if, at every point during its trip, the number of red posts visited so far is always at least the number of black posts visited so far. In the figure below, the cat wins by starting at post 3, but loses if it starts at any other post.
   Prove that the cat can always win if it starts on the right post. Prove this by induction on n for $n \geq 1$. Let your IH be that the cat can always win in the case where there are between 2 and $2(n-1)$ posts.

   Base Case: $n = 1$
   When $n = 1$, we have a total of $2(1) = 2$ posts, one black and one red. The cat always goes in a clockwise motion, it can either start on a red or black post. If the cat chooses to start on the red post it will succeed the win condition, therefore, it is possible to win.

   Inductive Hypothesis:
   Assume that for any k s.t. $k \geq 1$, the cat can win by starting at the correct post in any arrangement of $2k$ posts, evenly split between red and black, where the number of red posts visited is always at least the number of black posts visited.

   Inductive Step:
   Show that $n = k + 1$ where there are $2(k + 1)$ posts.
   We have made the assumption it is possible to win with $2k$ total posts, and it is only possible to win when $n\%2 == 0$ since we always must have equally as many red and black posts, and cannot have a portion of a post.
   We can show the winning property holds true as we increase n by showing that $R(n) \geq B(n)$. After the first hop, the at has visited now red post and zero black, so $R(1) \geq B(1)$, therefore, the property holds as we increase n so long as we maintain the strict requirement that $n = R + B$ and $R == B$.

2. Design and analyze a data structure that maintains a bag of numbers and supports the following operations:

   - INSERT(x) inserts the number x into the bag
   - DELETE-LARGER-HALF() deletes the largest $\frac{n}{2}$ items from the bag where n is the bag size

   Show how you can implement this so that the amortized cost of both operations is $O(1)$ and so that you can output all n numbers in the bag in $O(n)$ time.

   We initialize an array of size n which will store all values in the order in which they are inserted. We will also create a max heap of size $\frac{n}{2}$ to store the largest half of the elements. Every time we insert an item, we first add it to the array and then check to see if we should add it to the max heap, which means our insert time will remain at $O(1)$.

   Inserting into the array takes $O(1)$ time. Adding to a max heap regularly takes $O(\log(n))$ time, however since we are only tracking the top $\frac{n}{2}$ elements, this amortizes itself to $O(1)$.

   Deleting the larger upper half is more difficult, we will need to utilize our max heap to loop through each $\frac{n}{2}$ elements and delete each one we see from the n elements in the array. This will take $O(n)$ time.

   Delete larger half trivially amortizes to $O(1)$ because for every operation that clears the max heap which costs $O(n)$ once per halving the bag size. Since each doubling of the size happens after exactly $\frac{n}{2}$ more inserts have taken place, so we offset the extra time it takes with the quick $O(1)$ operations at a consistent rate, bringing our actual amortized speed of the entire algorithm back to $O(1)$.

3. Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

   Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = [\lg(n+1)]$, and let the binary representation of n be $< n_{k-1}, n_{k-2}, \ldots, n_0 >$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

   (a) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

   SEARCH needs to search each array. Since each is sorted, we can do a binary search on A_i. When an element is found, return the element. If it is not found, exit.

   If there are at most $k \leq \log(n+1)$, and a single array can be no larger than the array A_{k-1} with size 2^{k-1}. In the worst case we need to search through every single array before determining that we do not have the element, which will take $\sum_{i=0}^{k-1} O(i) = O(k^2)$ time, where $k = O(\log(n))$, we have a final worst case running time of $O((\log(n))^2)$.

(b) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

We find the smallest index i in some A_i moving on through the different arrays A only once they're full, such that we can find a location for our inserting item. If A_0 is empty, insert directly into A_0. if A_0 is full, recursively move A_0 into a higher indexed arrays until we're able to form an empty one.

The worst case si when every single array $A_0 ... A_{k-1}$ is full, so we end up doing a full A_{k-1} merges. This results in a total of $O(2^k) = O(n)$ operations. Even with a linear worst case running time, our average case still overtakes the worst case and we have an amortized runtime of $O(1)$.

(c) Discuss how to implement DELETE We need to start with running the SEARCH function which will run in $O(\log(n)^2)$ time. After we've removed the item, we have broken our array structure since the current array (unless the element was in A_0, so we need to re-balance all of the arrays. Replicating the same process we need to do for every insert, this will end up taking $O(n)$ time.

We will then have a worst case running time of $O(n)$ bounding the entire delete operation. This operation is bounded by the same logic in parts a and b, so we still end up with an amortized runtime of $O(1)$ dispite the chance for a particularly inefficient worst case.

4. Professor Curly conjectures that if we do union by rank, without path compression, the amortized cost of all operations is $o(\log(n))$. Prove him wrong by showing that if we do union by rank without path compression, there an be m MAKESET, UNION, and FINDSET operations, n of which are MAKESET operatiosn, where the total cost of all operations is $\Theta(m \log(n))$

The union operation consists of attaching the tree with the smaller rank to teh tree with the larger. Functionally, we will only actually increase the height of the tree when the subtree is the exact same height as the tree it is being added to. The height of the tree is then $O(\log(n))$

If we were using path compression, the FINDSET operation would flatten the trees. Now that we're restricted to not do so, we need to traverse a tree that can have a worst case height of $O(\log(n))$.

We need to start with making n MAKESET operations. We next do $n-1$ UNION operations that gives a tree of height $O(\log(n))$, and then $m-n$ FINDSET operations.

Each MAKESET takes $O(1)$ time, across n operations costs $O(n)$.

UNION operations take $O(1)$ time for a total of $n-1$ operations, or $O(n)$ bounded.

We need to end with $m-n$ FINDSET operations, each will take $O(\log(n))$ based off of the height tree, for a total time of $O(m \log(n))$.

Overall, our total runtime becomes $O(m \log(n))$, which contradicts the assertion that it is bounded by $o \log(n)$.

5. Describe and analyze a data structure to support the following operations on an array $A[1..n]$ as quickly as possible. Initially, $A[i] = 0$ for all i.

   - **SetToOne(i)** Given index i such that $A[i] = 0$, set $A[i]$ to 1
   - **GetValue(i)** Given an index i, return $A[i]$
   - **GetClosestRightZero(i)** Given an index i, return the smallest index $j \geq i$ s.t. $A[j] = 0$, or report that no such index exists.

   The first two operations should run in worst-case constant time, and the amortized cost of the third operation should be as small as possible.

We start with an array $A[1..n]$ that will hold all of the values. We can also define a pointer array $P[1..n]$ where each index i points to a doubly linked list, $A[i]$, if $A[i] = 0$, and just null otherwise.

SetToOne(i)
start by setting $A[i] = 1$. If $P[i]$ is not null, remove node i from linked list. We can then update $P[i]$ to null. This ends up taking only $O(1)$ time.

GetValue(i)
just returns $A[i]$, so simply $O(1)$.

GetClosestRightZero(i)
With our structure we can just return i when $A[i] = 0$. If $A[i] = 1$, we need to follow through the linked list that $P[i]$ is pointing to and find the smallest index $j \geq i$ where $A[j] = 0$ and return this value. If you traverse the entire structure without finding one, you return no such index exists. This will take up to $O(n)$ runtime in the worst case, but the amortized case will work out to be $O(1)$ because of how we are first sorting into an array of linked lists.

With this structure we can get down to an overall worst case runtime bounded by $O(n)$, but with utilizing our second array storing pointers and the primary array essentially with boolean values corresponding to the existance of each linked list, we can optimize to have an amortized runtime of the overall data structure as $O(1)$.