6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators:

Problem 1: Devise a way to avoid initializing large arrays. More specifically, develop a data structure that holds n items according to an index $i \in \{1, \ldots, n\}$ and supports the following operations in $O(1)$ time per operation: $init, set(i, x)$, and $get(i)$. Your data structure should use $O(n)$ space and should work regardless of what garbage values are stored in that space at the beginning of the execution.

Solution: $\square$

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators:

Problem 2: Our Van Emde Boas construction gave a high-speed priority queue, but with a little more work it can turn into a high-speed "binary search tree". Augment the Van Emde Boas priority queue to track the maximum as well as the minimum of its elements, and use the augmentation to support the following operations on integers in the range $\{0, 1, 2, \ldots, u-1\}$ in $O(\log \log u)$ worst-case time each and $O(u)$ space total: $Find(x, Q)$, $Predecessor(x, Q)$, and $Successor(x, Q)$.

Solution: First let us define some notation. We will call the array of the high bits $Q.top$ and the array of arrays of the low bits $Q.bottom$. Additionally, we will assume that there are two functions $high(x)$ and $low(x)$ which will find the top and bottom bits respectively of an integer x. We also will have a function $combine(low(x), high(x)) = x$ which can combine the low and high bits back together into x.

To perform a find operation, we will use a find operation. If x is the minimum or maximum of the current queue, return it. Next, examine $Q.bottom[high(x)]$ and check if the bin at position $low(x)$ is nonempty. If it is, we recurse on $Find(x, Q.bottom[high(x)][low(x)])$. We stop until we have found the minimum or maximum of the current sub-queue, or if we have found an empty element in one of the bins we are checking (and return null). The runtime is given by $T(b) = 1 + T(b/2) = O(\log b)$, and since b is the size of the bucket, we can set $b = \log u$ so that the find costs $O(\log \log u)$.

To perform the predecessor operation, we will again use a recursive structure. To begin, $Predecessor(x, Q)$ will return $Q.maximum$ if $x > Q.maximum$ and will return null if $x < Q.mininum$. Otherwise, it will call $R = Predecessor(low(x), Q.bottom[high(x)])$, which will search to see if there is any smaller element inside of the $Q.bottom[high(x)]$ array than x. If there is, then we will return R as the result. If there is not, then we must go to the next higher level in $Q.top$ and perform $S = Predecessor(high(x), Q.top)$. If S is not null, then we can take the maximum element in S and return $combine(S.maximum, high(x))$. If S is null, then we must take the minimum element in Q since there is no smaller element in any of the bins. This operation requires $T(b) = 1 + T(b/2) = O(\log b)$ time. This is because we will only perform successive $Predecessor$ operations if the previous $Predecessor$ operation returned null, which implies that the operation stopped at the second level of the recursive call. This means that any $Predecessor$ operation that returns null can return in $O(1)$ time. Therefore, in the worst case, only one "deep" $Predecessor$ call is made which costs $T(b/2)$. Thus, the operation requires $O(\log \log u)$ time.

For the successor operation, we do the same thing as in predecessor, except we will switch $Q.maximum$ with $Q.minimum$ and vice versa, and also exchange $Successor(x, Q)$ whenever $Predecessor(x, Q)$ is called. Thus, we can use the same analysis and show that successor also runs in $O(\log \log u)$ time. $\square$

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators:

Problem 3: In class we saw how to use a van Emde Boas priority queue to get $O(\log \log u)$ time per queue operation (insert, delete-min, decrease-key), when the range of values is $\{1, 2, \ldots, u\}$. Show that for the single-source shortest paths problem on a graph with n nodes and range of edge lengths $\{1, 2, \ldots, C\}$, we can obtain $O(\log \log C)$ time per queue operation, even though the range of values is $\{1, 2, \ldots, nC\}$.

Solution: We will use Dijkstra's algorithm for shortest paths and a priority queue Q of size C. Initially, Q will holds items of weight in the set $\{1, 2, \ldots, C\}$. Dijkstra's will start at some source vertex s and will insert all the neighbors and their distance from s into Q. Note that the max size of any of these paths will be C because there is no edge of length greater than C and in the first iteration, there will only be paths of length 1 inserted.

Now, perform a delete-min operation on Q and relax the edges of each neighbor of the min just deleted. If a path has a relaxed weight in $1, \ldots, C$, put this new path back into Q. However, relaxing these edges means that there will now be some new edges x with weight $C < x \leq 2C$. We shall create a new van Emde Boas priority queue for these edges and call it Q'. The new structure will hold all paths with weight in $\{C + 1, \ldots, 2C\}$. However, note that the maximum path length is $2C$ because the can only be paths of length 2 being relaxed.

We next perform a delete-min on Q, and insert the new paths into Q and Q'. If the path weight is in $\{1, \ldots, C\}$, we insert the path into Q, otherwise, we insert it into Q'. We know, since we removed an edge from Q, that the maximum path weight must be $2C$ for any edge we relax, since we are only relaxing using a single edge of weight at most C.

We continue to perform delete-min and insert into Q and Q' until Q is empty. At this point, all of the available paths should now be in Q'. We shall set $Q = Q'$ and repeat the operation for $Q' = \{2C+1, \ldots, 3C\}$ until we finish the pass of Dijkstra's. Note that this still follows Dijkstra's algorithm because $\forall x \in Q, y \in Q'$ we must have $x < y$, so that removing the minimum element from Q will always give the minimum element in the visited set.

Moreover, we are only using two van Emde Boas structures at any given time, each of bucket size $b = \log u$. This means we can perform insert, delete-min, and decrease-key operations in $O(\log \log u)$ time. $\square$

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators:

Problem 3:

Solution: □