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: We will use two arrays of size n. The first, S will be a summary array, and the second F, will physically store the items in the array. When init is called, we will find two blocks in memory which are both n units in length. These blocks of memory need not be cleared, but they should not overlap with memory which is currently being used. We shall assume this requires $O(1)$ time.

We shall have a counter of the number of items inserted into the queue, which will be initialized at 0. Whenever someone calls $set(i, x)$, we shall first set $S[i] = counter$, then set $F[counter] = StorageObj(S[i], x)$, and finally increment the counter. Here, $StorageObj(k, x)$ will be an object with a pointer to k and will store x in its memory. On the object, we can check $StorageObj.pointer()$ which will return a pointer to k, or we can check $StorageObj.data()$ which will return x.

To perform $get(i)$, we first look at $S[i]$ and see if anything is in this bin. If there is not, then we know we have not set i, and so we return null (regardless of what junk was in there before). If there is an item in the bin, it could either have been junk or it could actually have been set. To check, we first see if $S[i] < counter$. If not, then we know $S[i]$ is junk, so we can return null. Otherwise, we have to check and go to $F[S[i]]$ and see if there is a storage object initialized in it with the property that $StorageObj.pointer() == S[i]$. If this is true, then we can return $StorageObj.data()$.

It is clear our data structure uses $O(n)$ space since there are two arrays of n size, and also that it takes $O(1)$ time for initialization. Moreover, each operation requires $O(1)$ time. We only need to show that regardless of what garbage was in the memory slots before, we will still be able to achieve the correct performance.

Notice that if we ever set an item, we will always return it correctly. Thus, the only way anything can go wrong is if we have not yet set an item, and we return some junk. This can only possibly happen if somehow, $S[i]$ was already set so that $F[S[i]]$ had some $StorageObj$ which a poitner back to $S[i]$. This is unlikely to happen, but if it does (possibly because an array occupied this location previously), we can be sure that we will return the correct answer. This is because, up to the current counter, all items in $F[0], F[1], \ldots, F[counter]$ will have been set correctly. Thus, if we always check that $S[i] < counter$, then we can be sure that $F[S[i]]$ has already been set correctly, and that there is no way we will make a mistake. □

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. □

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 4-a: Suppose a random function (i.e., all function values are uniformly random and mutually independent) is used to map each item to buckets. Give a good upper bound on the expected number of collisions (i.e., the number of pairs of items that are placed in the same bucket).

Solution: For the kth inserted item, we know that the number of items inserted into slots is $k-1$. Therefore, when inserting the kth item, the probability that both slots that it should be inserted into are occupied is $\left(\frac{k-1}{n^{1.5}}\right)^2$, since the probability that just one of them is filled is $\frac{k-1}{n^{1.5}}$.

Now, when we insert n items, the expected number of collisions will be given by

$$\sum_{i=1}^{n-1}\left(\frac{i}{n^{1.5}}\right)^2 = \frac{1}{n^3}\sum_{i=1}^{n-1} i^2 \tag{1}$$

$$= \frac{(n-1)(2n-1)}{6n^2} \tag{2}$$

$$= \frac{2n^2-3n+1}{6n^2} \tag{3}$$

Since we know that $1-3n<0$ for all $n\geq 1$, we can say that the expected number of collisions is less than $1/3$. Therefore, We have $O(1)$ expected number of collisions. □

Problem 4-b: Argue that bashing can be implemented efficiently, with the same expected outcome, using the ideas from 2-universal hashing.

Solution: Let H be a 2-universal hash family. We will use two hash functions for the two arrays, where each array is of size $kn^{1.5}$ (k will be determined later). The probability that item i collides with item j in one of the arrays will be $\frac{1}{kn^{1.5}}$, since there are $kn^{1.5}$ slots available. Now, the probability that item i collides with any other item in one of the arrays will be $\sum_{j=1}^{n}\frac{1}{kn^{1.5}} = \frac{n}{kn^{1.5}} = \frac{1}{k\sqrt{n}}$.

Therefore, the probability that item i has a collision in both of the arrays will be $\left(\frac{1}{k\sqrt{n}}\right)^2$. The expected number of collisions is therefore given by

$$\sum_{i=1}^{n}\left(\frac{1}{k\sqrt{n}}\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{k^2} \tag{4}$$

$$= \frac{1}{n}\frac{n}{k^2} \tag{5}$$

$$= \frac{1}{k^2} \tag{6}$$

If we set $k=2$, then we can make the expected number of collisions to be $1/4$. Therefore, there must exist a set of 2 hash functions in H for which every item is placed in an empty bucket. Since every set of items i and j will get a set of functions which perfectly place the items into empty buckets, we have discovered a family of hash functions which will allow us to perform perfect hashing on the bi-bucket hashing. Moreover, since H has size which is $O(n^2)$, we know that we can implement bashing efficiently. □

Problem 4-c: Conclude an algorithm with linear expected time (ignoring array initialization) for identifying a perfect bash function for a set of n items. How large is the description of the resulting hash function?

Solution: We know that bashing can be implemented efficiently using a 2-universal hash. Thus, we can take the algorithm for finding a 2-universal hash function and adapt it here. We shall pick p where

$n < p \leq 2n$, then choose a, b uniformly at random such that $a < p$ and $b < p$. We use these three parameters to create the hash function $f(x) = (ax + b \mod p) \mod s$.

Using the analysis from above (and from the analysis in class that $f(x)$ is pairwise independent), we know that randomly selecting a and b will give us an expected value of $1/4$ collisions. If we select a, b t times, the probability that none of them are perfect bash functions is $(1/4)^t$. This means that if we keep selecting two sets of a and b until we get $f_1(x)$ and $f_2(x)$ which form perfect bash functions, we will expect to have $\frac{1}{3/4} = 4/3$ tries. Thus, we only need to perform the operation for selecting a, b an expected constant number of times. Moreover, we can check if $f_1(x)$ and $f_2(x)$ are perfect by performing $f_1(i)$ and $f_2(i)$ for all i in the set, to get a check in time $O(n)$. Therefore, we have an expected number of $4n/3$ operations, or $O(n)$ time for identifying a perfect bash function.

This shows that we can identify a perfect bash function with linear expected time. The size of the description of the resulting hash function will be $O(1)$ in size. However, the hash family is of size $O(n^2)$. $\square$