6.854 Advanced Algorithms

Problem Set 2

John Wang

Collaborators: Ryan Liu, Varun Ganesan, Jason Hoch

Problem 1-a: In class, I stated that single rotations don't work for splay trees. To demonstrate this, consider a degenerate n-node linked list shaped binary tree where each node's right child is empty. Suppose the only leaf is splayed to the root by single rotations: show the structure of the tree after this splay. Generalizing, argue that there is a sequence of $n/2$ splays that each take at least $n/2$ work.

Solution: If we start with the given degenerate n-node tree and splay the bottom-left node (the leaf) x, then we will slowly move the node up to the root through a series of "kink" rotations. These kink rotations will move x (which is the global minimum of the tree) and decrease its depth by one each time. Visually, all the nodes above and below x will be left children of their parents, and x will be the only node which is a right child of its parent. The process of moving x up the tree will continue until x becomes the root. At this point, x will only have a right child, which will be the maximum element of the data structure, and everything else will continue down in a series of left children.

Let us call this resulting data structure T'. When we splay the new leaf of the tree x' (node which can be found by walking down the left half of the tree), we can see that it will take the same path up to the root through a series of "kink" rotations. Once x' becomes a right child of the root x, a rotation will occur which puts x' at the root of the tree, and x as the left child (since x is the absolute minimum and is the only node smaller than x'). The right child of x' will be the global maximum. The tree now has depth $n-1$.

Continuing in the same pattern and taking the deepest leaf x'' of the tree, we will go through a series of $n-2$ rotations until x'' reaches the root. At this point x' will be the left child of x'', with x as the left child of x'. The maximum node will be the right child of the new root, so the tree will now have depth $n-2$.

It is clear that the deepest leaf a will always be promoted up to the leaf in time equal to the depth of the tree. Once it has reached the root, the previous root b will become its left child, and the maximum node which was previously b's right child, will become a's right child. Therefore, the tree will decrease in depth by 1.

A series of $n/2$ operations that splay the deepest leaf will therefore cost $O(d)$ each, where d ranges from $n, \dots, n/2$. Therefore, we have shown that there is a sequence of $n/2$ splays that each take at least $n/2$ work. $\square$

Problem 1-b: Now from the same starting tree, show the final structure after splaying the leaf with (zig-zig) double rotations. Explain how this splay has made much more progress than single rotations in improving the tree.

Solution: Starting the the only leaf of the tree x, we will perform zig-zigs on the way up to the root. For notation, we will say x is the leaf (minimum element), x' is the parent of the leaf (second smallest element), and x'' is the parent of x', etc. After the first zig-zig, we see that everything except the last three nodes have been untouched, but x is now the child of x'''. Moreover, x has a right child of x' and x' has a right child of x''.

Performing another zig-zig operation, the x' to x'' chain will become the left subtree to x'''. Moreover, x''' will have a right child of x''''. In order to see the generalized behavior, we will call A the right subtree of x. Note that x has no left child at all. Now, x is the left child of its parent y, and y is also a left child of its parent, z. Thus, performing a zig-zig operation will bring x to the top of y and z. Thus, x will have a right child y and no left child, while y has a right child of z and a left child of A.

It is now clear that zig-zig operation will bring x up higher in the tree while adding the chain of y and z to the right subtree of x. Additionally, y's left child will now be x's previous subtree.

Once the splay of x is finished, each node on the left-most path from the root will have a right child. Thus, we see that using the zig-zig operation, the splay will decrease the depth of the tree by a multiplicative factor of 2. In the previous case, the depth after a single splay was $n-1$, while now the depth is $n/2$, which is clearly much improved. $\square$

Problem 1-c: Given the theorem about access time in splay trees, it is tempting to conjecture that splaying does not create trees in which it would take a long time to find an item. Show that this conjecture is false by showing that for large enough n, it is possible to restructure any binary tree on n nodes into any other binary tree on n nodes by a sequence of splay operations. Conclude that it is possible to make a sequence of requests that cause the splay tree to achieve any desired shape.

Solution: We will start by showing how we can use splay operations to make a specified node into a leaf. First, we make the tree T into a left chain (i.e. a tree with only left children from the root). This can be done by splaying the nodes in T in descending order. Now, to make x a leaf, we take x's grandchild and splay it. Since a zig-zig operation will be invoked, x will be moved to a leaf position with no child, and will be the child of x's previous child. Thus, x is now a leaf. If x does not have a grandchild, we make the tree T into a right chain (i.e. a tree with only right children from the root) by splaying the nodes in ascending order. Once this is done, we splay x's grandchild and the same analysis can be done to show that x will become a leaf. This works for trees of size $n \geq 4$. If x is already a leaf when T is in a chain, then there is no need to splay anything.

We will now use induction to show that a tree of any desired shape can be made. We shall start with $n = 4$. From the diagrams below, it is clear that a set of splays can achieve any tree configuration with $n = 4$ nodes.

Now, we will use a couple of observations. If a node x is a leaf, then any splays not on x will not affect the leaf status of x. This is clear because both zig-zig and zig-zag rotations only change the position of the subtrees A, B, C, and D. Therefore, if there is a rotation done in the splay, x will remain a leaf since the subtrees will be unchanged. Next, if x is a leaf, then splays on any other parts of the tree will not be affected by the location of x. Effectively, we can remove x from the tree and perform the same splay operations as when x was in the tree. This is true because x will always be in a subtree in any splay operation, and the subtree will be moved according to the rotations in the splay, but the subtree will not affect which nodes are actually being moved.

Also, we note that if two binary trees differ by a single node, that node cannot be a leaf node. Suppose by contradiction that this is the case and that in tree 1, y has a parent p_{y1} whereas in tree 2, y has parent p_{y2}. Without loss of generality, we can assume $p_{y1} < p_{y2}$ since they must be distinct parents in order for the leaf nodes to differ. Find $l = lca(p_{y1}, p_{y2})$ which is the least common ancestor of these two nodes. We must have $y < l$ in tree 1 whereas $y > l$ in tree 2. However, we know that y and l have fixed values, which is a contradiction. Therefore, two binary trees cannot differ by a single node which is a leaf.

Now we will invoke the induction hypothesis that we can restructure a tree of size n into any other tree of size n by a series of splay operations. We must show that is the case for $n + 1$. So suppose we have a tree T which we wish to restructure into tree A where both trees are of size $n + 1$. We shall take a leaf $l_1(A)$ in tree A and find the corresponding leaf $l_1(T)$ in tree T. We shall use our splay operations to make $l_1(T)$ into a leaf in tree T. We have shown, however, that splays are not affected by leaves and that $l_1(T)$ will remain a leaf through all splay operations expect for those on $l_1(T)$ itself. This means that we have reduced the problem to reorganizing $T - l_1(T)$ into $A - l_1(A)$. However, both of these trees are of size n. By the induction hypothesis, there is a series of splay operations to convert $T - l_1(T)$ into $A - l_1(A)$. We know, however, that if two binary trees differ by a single node, that node cannot be a leaf node. This means that T and A must actually be the same tree. This completes the inductive step. $\square$

6.854 Advanced Algorithms

Problem Set 2

John Wang

Collaborators: Ryan Liu, Varun Ganesan, Jason Hoch

Problem 2-a: Show that the above tree structure is asymptotically comparable to the optimal static tree in terms of the total time to process the access sequence.

uc**Solution:** First, we note that searching for node v in tree S_k requires $O(\log 2^{2^k})$ time since the S_k has 2^{2^k} nodes. Now, if v is the lth most frequent element, then it will be first found in tree S_k where $2^{2^{k-1}} < l < 2^{2^k}$. Thus, we see that $2^k < \log l$ so that $k < \log\log l$. The total search time is therefore:

$$\sum_{i=1}^{\log\log l} O(\log 2^{2^i}) = \sum_{i=1}^{\log\log i} O(2^i) \tag{1}$$

Which follows since we have to search in trees $S_1, S_2, \ldots, S_k$ before we can find v. Now, each node v_i will be searched for $p_i m$ times. This means that the total cost, given in terms of all nodes i which are in the access sequence, is given by:

$$\begin{aligned} \sum_i p_i m \sum_{j=1}^{\log\log l} O(2^j) &= \sum_i p_i m O(2^{\log\log l}) \qquad (2) \\ &= m \sum_i p_i O(\log l) \qquad (3) \end{aligned}$$

These steps follow because $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$. Now, we can get a bound on l. we know that the ith most probable element can have at most $1/p_i$ elements with access frequencies larger than it (otherwise $\sum_{j=1}^{n} p_j > \sum_{j=1}^{i} p_j > \sum_{j=1}^{i} p_i = 1$ which is impossible). Thus, we can say that $l < 1/p_i$. This shows that our total cost of processing the access sequence is

$$O\left(m \sum_i p_i \log \frac{1}{p_i}\right) \tag{4}$$

Which is what we wanted to show. □

Problem 2-b: Make the data structure capable of insert operations. Assume that the number of searches to be done on v is provided when v is inserted. The cost of insert should be $O(\log n)$ amortized time, and the total cost of searches should still be optimal (non-amortized).

Solution: The new data structure will have two sets of trees for each S_k in the previous data structure. The first tree will be a regular search tree keyed on the keys of each node. The second tree will be keyed on each node's access frequencies. Adding the second tree will not change the runtime of any of the previous operations, since each operation will only be performed twice.

Now, in order to insert a node x into the structure, we will start with S_1 and check if the access frequency of x is greater than the minimum access frequency in S_i. If it is, we insert x into S_i and check the size of S_i (assume we keep a counter on each tree so that this takes constant time). If $S_i > 2^{2^i}$, then we delete the minimum element of S_i.

If S_l is the final tree so that $l = \log\log n$, and we delete the minimum element of S_l, a new search tree S_{l+1} needs to be created with all the previous elements. S_{l+1} will be of maximum size $2^{2^{l+1}}$ and have all the keys in the data structure. Now we must show that insert has an amortized cost of $O(\log n)$.

Let use define a potential function $\Phi = 2^{l+1}(R - 2^{2^{l-1}})$, where R is the number of elements in S_l. Inserting a node, and possibly deleting a node, at each level costs $\sum_{i=1}^{l} O(\log 2^{2^i}) = \sum_{i=1}^{l} O(2^i) = O(2^{\log\log n + 1}) = O(\log n)$ with a change in potential of $2^{2^l} = \log n$. However, creating a new level has real cost of $O(n \log n)$, since there are n elements which need to be inserted. This occurs only when there are 2^{2^l} nodes in S_l, however, so the change in is $2^{l+1}(2^{2^l} - 2^{2^{l-1}}) = n \log n$, which pays for the cost of creating a new level.

Thus, we see that insert can be accomplished in $O(\log n)$ time, and time for a search is unaffected. □

Problem 2-c: Improve your solution to work even if the frequency of access is not given during the insert. Your data structure now matches the static optimality theorem of splay trees.

Solution: From problem 2-b, recall that we were keeping track of the access frequencies at each level S_i. Now, whenever an item x is queried or inserted, we will update the access frequencies at a cost of $O(\log S^i)$ for each level i. Also, we will check whether the new access frequencies (of the inserted or queried item) is greater than the minimum access frequency in S_{i-1}. If it is, we shall delete the minimum access frequency node in S_{i-1} and insert x into S_{i-1}. Notice that the cost of performing these operations is the same as an insert, which was analyzed in the previous section. Thus, queries and inserts will still require $O(\log n)$ amortized time.

To show that this data structure now satisfies the static optimality theorem, we need to show that if an item x is accessed $p_x m$ times, then the total access time is $O(\sum_x mp_x \log 1/p_x)$. After t total accesses, the cost of a query on item x, using the logic used in problem 2-a, will be at most the inverse of the current probability of access (or $\lambda(x,t)/t$ where $\lambda(x,t)$ is the total number of accesses for item x after t total acesses). This means the cost of a query is $O(t/\lambda(x,t))$. For m total items, we have a cost of $\sum_{t=1}^{m} O(t/\lambda(x,t))$. Simplifying, we obtain:

$$\begin{aligned}
&(5) & \sum_{t=1}^{m} \log \frac{t}{\lambda(x,t)} &= \log \frac{t}{\prod_{t=1}^{m} \lambda(x,t)} \\
&(6) & &= \log \frac{m!}{\prod_x (p_x m)!} \\
&(7) & &= \log \frac{m^m}{\prod_x (p_x m)^{p_x m}} \\
&(8) & &= m \log m + \sum_x m p_x \log \frac{1}{p_x m} \\
&(9) & &= m \log m + \sum_x m p_x \log \frac{1}{p_x} - \sum_x m p_x \log m \\
&(10) & &= \sum_x m p_x \log \frac{1}{p_x}
\end{aligned}$$

Where we have used sterling's approximation for factorials that $n! \approx c\sqrt{n}(n/e)^n$ so that $n! = O(n^n)$, and we have also used the fact that $\sum_x mp_x \log m = m \log m$ since $\sum_x p_x = 1$. Therefore, we see that our data structure matches the static optimality theorem. □

Problem 2-d: Make your data structure satisfy the working set theorem on splay trees. Ignore the static optimality condition.

Solution: We need to show that the cost of performing m accesses on a set of n items is given by $O(n \log n + \sum_{j=1}^{m} \log t(j)$ where $t(j)$ is the number of distinct elements accessed between access j and the previous time the same element was accessed. In order to do this, we change the structure so that instead of holding the most frequently accessed items, we shall hold the most recently accessed items in the second set of trees at each level. Queries and inserts will work in the same manner as before, and will still have amortized cost of $O(\log n)$.

To show that the working set theorem is satisfied, we will look at the cost of a series of m accesses on n items. First, we know that to build the structure requires $n \log n$ time. Next, we know that as soon as a query is performed on node x, it will be placed into S_1 and will slowly move into $S_2, S_3, \ldots$ as other nodes are queried. After $t(j)$ accesses, node x will be in level $S_{\log\log t(j)}$. Thus, a series of m accesses after the structure is built will cost:

$$\sum_{j=1}^{m}\left(\sum_{i=1}^{\log\log t(j)} O(\log 2^{2^i})\right) = \sum_{j=1}^{m}\left(\sum_{i=1}^{\log\log t(j)} O2^i\right) \tag{11}$$

$$= \sum_{j=1}^{m}\left(O(2^{\log\log t(j)})\right) \tag{12}$$

$$= \sum_{j=1}^{m}\log t(j) \tag{13}$$

Thus, the total cost of m accesses (including the cost to create the data structure), is $O(n\log n + \sum_{j=1}^{m}\log t(j))$ which is what we wanted to show. □

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators: Ryan Liu, Varun Ganesan, Jason Hoch

Problem 3: Describe a data structure that represents an ordered list of elements under the following three types of operations: 1) access(k): return the kth element of the list (in its current order) 2) insert(k,x): insert x (a new element) after the kth element in the current version of the list 3) reverse(i,j): reverse the order of the ith throgh jth elements. Each operation should run in $O(\log n)$ amortized time, where n is the current number of elements in the list. The list starts out empty.

Solution: We will create a splay tree and augment its nodes. For each node x, we will have $x.left$ as the left child, $x.right$ as the right child, $x.reverse$ as a reversal bit, $x.pseudoleft$ and $x.pseudoright$ as reversal children, and $x.size$ as a count of the number of nodes in a subtree rooted at x.

We will use $x.size$ as a proxy for the order of element x inside of the list. We will update $x.size$ during all splay tree operations without any extra asymptotic time. This is because each rotation will only affect the size of 3 nodes. Each of these three nodes can have their size recomputed by using $x.left.size + x.right.size + 1$. Since the subtrees of each of these nodes does not change during a rotation, we can be sure that the new size for each of the nodes will be correct. Since we perform a constant number of operations for a constant number of nodes on each rotation, $splay(x)$ will still cost $O(\log n)$ amortized time.

We shall also update $x.reverse$ during rotations. The reverse bit dictates which child is $x.pseudoleft$ and which is $x.pseudoright$. If the reverse bit is set, then $x.pseudoright = x.left$ and vice versa. To update $x.reverse$ during rotations, we can use the fact that $x.reverse = \sum_k k.reverse \pmod 2$ for all k which are the descendants of x which have been modified. Using this formula for rotations in the reversal of bits means that the traversal of the tree will be preserved, as long as we enter the tree through $x.pseudoleft$ then x itself before we go through $x.pseudoright$. Since $x.reverse$ can be calculated in a constant number of steps, due to the fact that subtrees of x, y, and z do not change during rotations, the splay operation still costs $O(\log n)$ amortized time.

To perform $access(k)$, we start at the root r and look at $r.left.size$. If $k = r.left.size + 1$ then we return the root. If $k > r.left.size + 1$, then we set the right child of the root as the new r and set k equal to $k - r.left.size - 1$. If $k < r.left.size$, then we set $r = r.left$ and look for k. We perform this recursively until we find the kth element. This requires $O(\log n)$ amortized time.

To perform $insert(k, x)$, we perform an insert into the splay tree, which should have runtime of $O(\log n)$, and we make sure to update all our augmented data. Since we first split the tree on x, we can update the augmented data in $O(1)$ time, then merge it back and update the new root in $O(1)$ time. This means that insert should still have a runtime of $O(\log n)$.

Finally, to perform $reverse(i, j)$, we must split at element i to form two trees T', T. Then we split T again at element j to form T'', T'''. We know that T'' will be composed of only elements which are less than j and greater than i. Thus, we can update the root of T'' and change its reverse bit in order to reverse all elements between i and j. Then, we merge T'' with T''' and then T' with the new structure to obtain T with elements i through j reverse. This also costs $O(\log n)$ amortized time because each split costs $O(\log n)$. $\square$

6.854 Advanced Algorithms

Problem Set 2

John Wang

Collaborators: Ryan Liu, Varun Ganesan, Jason Hoch

Problem 4-a: As in class, a double rotation will involve 3 nodes on the search path: x, its parent y, and its grandparent z. Call this triple biased if over 9/10 of z's descendants are below x, and balanced otherwise. Argue that along the given search path there can be at most $O(\log n)$ balanced triples.

Solution: Let us suppose we achieve a path with the maximum number of balanced triples possible. This occurs when we go on a path where each x node has $9/10 - \epsilon$ of z's descendants, where $\epsilon > 0$ is a small constant. The length of this path is then $O(\log_{9/10-\epsilon} n)$ since there are at most $O(\log_{9/10-\epsilon} n)$ times that one can move down towards the x in the recursion. We know that this is equivalent to $O(\log n)$, which shows that even in the worst possible case, there cannot be more than $O(\log n)$ balanced triples in a search path. $\square$

Problem 4-b: Argue that when a biased triple is rotated, the potential decreases by a constant, paying for the rotation. Do so by observing that rank of x only increases by a small constant, while the ranks of y or z decrease by a significantly larger constant. Do this for both the zig-zig and zig-zag rotations.

Solution: We shall start off with the zig-zig rotation for a biased triple. In this rotation, x becomes the new root of the subtree, exchanging positions with z. Here x retains its left subtree A, and y obtains x's right subtree B as its new left subtree. Moreover, z gains the right subtree C of y as its left subtree. If the triple is biased, then 9/10 of the nodes are concentrated in subtrees A and B. Thus, the rank of x will only change by less than $n/10$. However, z will lose both A and B and its rank will fall by at least $9n/10$ and y will lose some potential as well, since it no longer has A. Thus, we see that the change in potential is something along the order of $\log n/10 - \log 9n/10 = \log 1/9$. Thus, we see the potential decreases by a negative constant which is enough to pay for the biased rotation of cost $1 < |\log 1/9|$.

For the zig-zag rotation, we have x with left and right children B and C respectively, y with left and right child A and x respectively, and z with left and right children y and D. The rotation sends x to the root, with y and z as its left and right children. The children for y are A and B and the children of z are C and D. Now we see that x only gains a constant amount of weight, since it already held more than $9n/10$ of the nodes. Thus, its weight can only increase by at most $n/10$. As for z and y, the increase in size is distributed across B and C. However it is distributed, the nodes y and z lose at least $9n/20$ in weight in total. This means the change in potential is $\log n/10 - \log 9n/20 = \log 2/9$. We still see that the $1 < |\log 2/9|$ which allows us to pay for the zig-zag rotation with the change in potential. $\square$

Problem 4-c: Argue that when a balanced triple is rotated, the potential increases by at most $2(r(z) - r(x))$.

Solution: We shall start with a zig-zig rotation and label the nodes and subtrees as in lecture. The change in potential is given by $\Delta\Phi = r'(x) + r'(y) + r'(z) - (r(x) + r(y) + r(z))$. First, we know that $r'(x) = r(z)$ because they change places at the root, so their ranks must be the same. Moreover, we know that $r(y) > r(x)$ since x is a child of y. Therefore we have the simplification:

$$\begin{aligned}
\Delta\Phi &\leq r'(y) + r'(z) - r(x) - r(x) && (14)\\
&\leq r(z) + r(z) - r(x) - r(x) && (15)\\
&= 2(r(z) - r(x)) && (16)
\end{aligned}$$

Where the second step came from the fact that z was the root so that $r(z) > r'(y)$ and $r(z) > r'(z)$. This shows what we wanted for the zig-zig operation.

Now, for the zig-zag operation, we again assume the notation is the same as from class. Since we know that $r(z) = r'(x)$ as they are both roots, we can obtain $\Delta\Phi = r'(y) + r'(z) - r(x) - r(y)$. Moreover, since y is the parent of x, we know that $r(y) > r(x)$ so that $\Delta\Phi \leq r'(y) + r'(z) - r(x) - r(x)$. Finally, we know that $r(z) \geq r'(y)$ and $r(z) \geq r'(z)$ since z was the root in the original tree. This gives us $\Delta\Phi \leq 2(r(z) - r(x))$. Thus, whenever a balanced triple is rotated, the potential increases by at most $2(r(z) - r(x))$. $\square$

Problem 4-d: Conclude that enough potential falls out of the system to pay for all the biased rotations, while the real work and amount of potential introduced by the balanced rotation is $O(\log n)$, which thus bounds the amortized cost.

Solution: First, we know from 4-b that if there is a biased rotation, the cost of the rotation will be paid for by the decrease in potential. Now, if there is an unbiased rotation, then we know that the real cost is 1 and the potential increases by at most $2(r(z) - r(x))$. The total cost of a series of balanced rotations will be:

$$\sum_{i=1}^{\log n} 2(r(z_i) - r(x_i)) \tag{17}$$

Where x_i and z_i correspond to the x and z of a rotation during the ith rotation. Notice that since these are balanced rotations, there cannot be more than $\log n$ of them in a single search path. Moreover, notice that $r(z_i) = r(x_{i+1})$ because x replaces z as the root of the subtree. This implies that the above sum telescopes so that the cost of a series of unbiased rotations in a search path is at most

$$2(r(z_{\log n}) - r(x_1)) \quad = \quad 2(r(Z) - r(x_1)) \tag{18}$$

Where Z is the root of the entire tree. Using a weight of $w = 1$ for each node, we see that $r(Z) = \log n$ so that the total amortized cost of the unbiased rotations is $O(\log n)$. $\square$

6.854 Advanced Algorithms

Problem Set 2

John Wang

Collaborators: Ryan Liu, Varun Ganesan, Jason Hoch

Problem 5: The slowest part of multi-level-bucket heaps was the scan for the next nonempty bucket in a block. Suppose that instead of maintaining the set of a block's nonempty buckets in an array, you kept it in a standard binary heap (note this adds no data structural complexity since we already have the necessary array handy). Discuss how the runtimes of operations would change. By balancing parameters, argue that you can implement a heap with amortized runtimes of $O(\sqrt{\log C})$ for insert, decrease key, and delete-min, yielding a shortest paths algorthim with runtime $O((m+n)\sqrt{\log C})$.

Solution: We will use a multi-level-bucket heap as before, but each of the arrays will also have a copy of a standard binary heap. Note that maintaining the original array will not cost extra since all operations can be performed in $O(1)$ time once the element is found in the binary heap. Each heap A will be keyed on the the following function for each node x:

$$f(A,x) = \left\{ \begin{array}{ll} x.value & \text{if } x \text{ is non-empty} \\ (C+1)(x.value+1) & \text{else} \end{array} \right\} \tag{19}$$

Where $x.value$ is the index of bucket x. This function means that the minimum element x_m will be the element with the lowest key such that x_m is non-empty. In other words, it is the first bucket with an entry in it. Since $x.value < C+1$, we know that $(C+1)(x.value+1)$ will always be greater than anything less than C, so that $x.value < (C+1)(y.value+1)$ for all x and y. This means that if a node x is empty, it will have a higher key than a non-empty node.

This means that delete-min can be performed much faster than before. First, the node is removed from the second level of the bucket heap. This takes $O(\log b)$ time since the second level is now a binary heap containing b elements. We decrement the counter of items in the second level, and if it falls to zero, we must perform a decrease key on the binary heap (setting $f(A,x)$ to the correct value) in the first level which takes $O(\log C/b)$ since there are C/b buckets in the first level of the heap. Also, we must find a new minimum element. This will take $O(1)$ time by simply performing a find-min query on the first-level, then another find-min in the second level. Since $f(A,x)$ was defined so that the minimum non-empty bucket is produced by a find-min on the min-heap, we can do these operations in constant time.

If the delete-min did not remove all the items from the bucket, we can perform a find-min inside of that bucket. The keys are arranged according to $f(A,x)$ so that the minimum element that is non-empty is returned. This gives us the min of the entire data structure. Taking these two cases into account, we see that delete-min costs $O(\log b) + O(\log C/b)$ time.

Insertion is done by first finding the correct top-level bucket for the inserted node x. If x's top level bucket is empty, we must perform a decrease-key on the binary heap and reset the $f(A,x)$ value, costing $O(\log C/b)$ time. We do the same for the second level bucket, and perform a decrease-key on the second level binary heap and an insert into the original array. This costs an additional $O(\log b)$. Thus, insert requires $O(\log b + \log C/b)$ time.

Decrease-key is performed by by deleting an element from the data structure, then reinserting the element with an updated key value. The delete operation will cost $O(\log b)$ to delete from the second-level heap and $O(\log C/b)$ to update the top level heap (if deletion causes the second level bucket to become empty). Insertion is the same as above, so the total cost is $O(\log b + \log C/b)$. Now, if we set $\log b = \log C/b$, we find that $b = \sqrt{C}$ will minimize the cost of each operation. Each operation will therefore require $O(\log \sqrt{C})$ running time.

For shortest paths, we perform n inserts, m decrease-keys, and n delete-mins. Since we only need to use C buckets at any given time, we can perform shortest paths in $(n+m+n)(O(\log \sqrt{C}) = O((n+m)\log \sqrt{C})$.
□