6.854 Advanced Algorithms
Problem Set 2

John Wang
Collaborators:

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 $n, \ldots, n/2$. 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: THIS IS INCORRECT!!!! We will start by showing how we can use splay operations to make a specified node into a leaf. First, let us take an arbitrary node x which is in some tree T. To make x descend down the tree, we will splay the left and right child of x. Doing these splays will decrease the depth of the children, but increase the depth of x, sending it closer towards being a leaf. We perform this recursively until x has no more children. At this point, x has become a leaf.

Now, assume we have tree T which we want to restructure into T'. We first will find all the leaves of T' using BFS. Then, we will perform a series of recursive splay operations on the nodes in T which correspond to the leaves of T' to make them leaves. Notice that once a node y has become a leaf, no splay operation will be able to change y into an internal node unless we specifically splay y. Therefore, it is possible to reorganize T such that T has the same leaves as T'.

Once T has the same leaves as T', we must reorganize the internal nodes of T.

$\square$

6.854 Advanced Algorithms
Problem Set 2

John Wang
Collaborators:

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.

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:

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

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:

$$(2) \qquad \sum_i p_i m \sum_{j=1}^{\log \log l} O(2^j) \quad = \quad \sum_i p_i m O(2^{\log \log l})$$

$$(3) \qquad \qquad\qquad\qquad = \quad m \sum_i p_i O(\log l)$$

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

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

Which is what we wanted to show. $\square$

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. $\square$

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 $O(\sum_x m p_x \log 1/p_x)$. 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:

$$\text{(5)} \qquad \sum_{t=1}^{m} \log \frac{t}{\lambda(x,t)} \;=\; \log \frac{t}{\prod_{t=1}^{m} \lambda(x,t)}$$

$$\text{(6)} \qquad =\; \log \frac{m!}{\prod_x (p_x m)!}$$

$$\text{(7)} \qquad =\; \log \frac{m^m}{\prod_x (p_x m)^{p_x m}}$$

$$\text{(8)} \qquad =\; m \log m + \sum_x m p_x \log \frac{1}{p_x m}$$

$$\text{(9)} \qquad =\; m \log m + \sum_x m p_x \log \frac{1}{p_x} - \sum_x m p_x \log m$$

$$\text{(10)} \qquad =\; \sum_x m p_x \log \frac{1}{p_x}$$

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 $\sum_x m p_x \log m = m \log m$ since $\sum_x p_x = 1$. Therefore, we see that our data structure matches the static optimality theorem. $\square$

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:

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

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

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

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. $\square$

6.854 Advanced Algorithms
Problem Set 2

John Wang
Collaborators:

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:
□