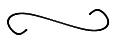
Petar Maymounkov
6.854J – Advanced Algorithms, Problem Set 2
With: Benjamin Rossman, Oren Weimann, and Pouya Kheradpour



Problem 1. Use a regular splay tree where each node keeps three extra variables:

1. left-count: The number of descendants of its left child (including the child itself),
2. right-count: The number of descendants of its right child (including the child itself), and
3. invert-bit: If this bit is set, the direction of each edge (left or right) in the subtree of this node is interpreted as switched. A descendant node (lower in that subtree) can switch the edge direction back by setting its invert-bit to 1, and so forth.

Before a splay operation on nodes x, y and z, and subtrees A, B, C, and D (using the standard notation) takes place, we normalize the invert-bits as follows. The invert-bits of x, y and z are set to 0, and their edges are switched accordingly to represent the corrrect direction. Additionally, the invert-bits of A, B, C and D are set accordingly to reflect the inherited (from their parents) edge direction inversion, combined with their own (via XOR). Then the splay operation can proceed as usual, without further regard of the invert-bits.

Splay operations also have to update the left/right-counts of x, y and z after splaying, in the straightforward way. Note that "right-count"s are not neces-

sary for this step as they can be solved for during a splay operation, however they will be necessary for reversal later on.

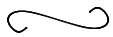
Split works as usual, except that the root of one of the subtrees has to inherit inversion from its past parent by XORing its parent's invert-bit into its own invert-bit. left-counts should also be updated accordingly. Join works similarly in the reverse way.

To **access**(k), start from the root t. If left-count$(t) \geqq k$, go to left child and recursively do **access**(k) there, else go to right child and do **access**$(k$ - left-count$(t))$. Finally, splay up as described above.

To **insert**(k, x), split the tree between k-th and $k+1$-st node, attach x as the root of the $1, \ldots, k$-tree via one left edge, set x's left-count to k, and set its invert-bit to 0. Then join the two trees.

To **reverse**(i, j), split the splay tree into three trees: $1, \ldots, i-1$, and $i, \ldots, j$, and $j+1, \ldots, n$. Flip the invert-bit of the middle tree, then join all three trees back together. (Note that this is where the right-counts are necessary, because they switch their meaning with the left-counts.)

All three update operations run in amortized $O(\log n)$ time, because we need no changes in the original splay analysis. We have only added a constant time of bookkeeping operations (for the left-count and invert-bit fields) per splay operation.



Problem 2. We are given a *work* and a *target* tree, both with identical sets of items and different shapes. The goal is to apply find/splay operations on the work tree until we bring it into a shape identical to the target tree.

Begin with the following:

Claim 1 *For any tree with $n \geqq 4$ nodes, we can turn any tree node into a leaf, by a sequence of find/splay operations on selected nodes.*

Proof: Prove by induction. Inductive step: assume claim is true for tree with $n-1$ nodes. Start with a tree of n nodes and pick an any element x. If x is leaf, we are done. Otherwise, pick any leaf y below x and delete it (mentally, for the moment). Then apply the inductive hypothesis. Now, if we hadn't deleted y, after the application of the inductive hypothesis x will either be a leaf (in which case done), or it will have one child: y.

Since x is now certainly not a root, there are two possible shapes (not counting their symmetric equivalents) for x, x's parent u, and y (edges listed in bottom-up order): $y \rightarrow x \rightarrow u$ or $y \leftarrow x \rightarrow u$.

The latter case is easy: splaying y will begin with a zig-zag step, which will leave x as a leaf.

In the former case, $y \rightarrow x \rightarrow u$, we start by making sure that u has a parent g (x's grandparent). If it doesn't, then u is root and it must have a child g different than x. We splay g, and it becomes a parent of u and a grandparent of x.

Next splay y. This begins with a zig-zig step, which produces $u \leftarrow x \leftarrow y$. Now, regardless of what is the next splay step for y (and there is one, because u has a parent g), it will leave x and u in the following configuration: $u \leftarrow x \rightarrow z$ (for some z coming from above). Which means that once y is fully splayed, and we apply the splay operation to u, the very first step will be zig-zag, and it will leave x as a leaf. This proves the inductive step.

Finally, we need to show that the base case $n = 4$ indeed holds. This is

demonstarted by an exhaustive list of all possibilities (28 in count: 7 possible tree shapes not couting symmetries, 4 nodes in each), which we omit for brevity. □

Call the operation described above *sink*(x). Let *deactivating* a node y mean that y and its descendants cannot be splayed during a *sink*(x) operation. Then the above claim still holds, but it ensures that x sinks to the bottom of the tree in the sense that there can only be deactivated nodes below it. The proof of the claim remains unchanged if we just pretend that the deactivated nodes are simply not present in the tree during the sink operation.

Our reshaping algorithm will go as follows. Introduce the operation *mark* on the data items, which marks (via a bit field) the tree nodes corresponding to that item both in the work and in the target trees. A an item is *markable* only if its corresponding node in the target tree: (a) is a leaf, or (b) all of its children are already marked.

By the definition of *mark*, marked nodes in the target tree form subtrees that are entirely marked. We will ensure that the same invariant holds for the working tree.

To bring the work tree into the shape of the target tree, recursively do: *Pick a markable item x, deactivate all marked items in the working tree, and do sink(x). Then mark x itself. Repeat until no unmarked items remain.*

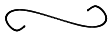
To show correctness, prove by induction that every marked substree in the target tree has an identical correspondent in the work tree. The base step is when we mark the first item. This case holds trivially. Now, assume the hypothesis holds true, and we are about to execute another iteration of the algorithm.

First note that all marked subtrees in the work and target trees can be treated

as single nodes, as no operations ever change their internal shapes. Therefore, we can assume we are in the case where both trees have an identical (item-wise) subset of leaves marked.

If the item x of the next iteration is also a leave (in the target tree) than it couldn't end up as a parent of marked (deactivated) nodes in the work tree, because that would violate the ordering of the marked items. For the same reasons, if it is a parent of two marked nodes, the corresponding marked nodes in the work tree must end up below it.

Each time a node is marked (and hence deactivated during iteration) the size of the tree on which the *sink* operation is performed shrinks. At some point it will shrink to 4 nodes. At this point, instead of applying our iteration procedure we use an explicit recipe for reshaping any tree of size 4 into any other such tree. We omit the recipe for the case 4 for the sake of brevity. It can easily be derived by playing with splay demo applications on the web.



Problem 3a. An item x of frequency p must be at most the i-th in order, where $i = 1 + (1-p)/p = 1/p$, because all items ahead of it must be of frequency at least p. Generally, an item of order i appears (for the first time) in S_k, where $k = \log\log i$. Also, the access time A_k for items in S_k is $\log|S_0| + \cdots + \log|S_k| = 2^0 + \cdots + 2^k = 2^{k+1} - 1$. Hence, the access time for an item of frequency p is $2^{\log\log i+1} - 1 = O(2^{\log\log i}) = O(\log 1/p)$. Therfore the total access time is $O(m \sum p_x \log 1/p_x)$, where m is the total number of accesses.

Problem 3b. Augment each S_k with a binary search tree B_k which contains all items in S_k that don't appear in S_{k-1}, keyed by their declared search count.

Begin with an empty S_0, and a counter m for the total number of searches to be performed. Maintain invariant that for every item x with access count a_x, x is to be found in at least $S_k, S_{k+1}, \ldots$ where $k = \log\log m/c_x$.

To maintain invariant, each time an item x is inserted, compute k and insert x in $S_k, S_{k+1}, \ldots$. If any S_i overflows, remove that item from S_i which has the smallest declared access count. (Find that item using B_i.) If S_i the last item in the tree collection, create the next tree S_{i+1} and insert all items from S_i into S_{i+1} as well the item from S_i that had the smallest access count and was removed from S_i. Update the B_i's accordingly: note that operations on B_i take the same time as operations on S_i.

Insert operations take $O(\log n)$ time except for when the last tree overflows. The cost for overflowing (and copying) a tree of size n is $O(n \log n)$. This cost is prepaied ahead of time (in the form of potential) by items inserted since the last overflow. The number of items since the last overflow is $n - \sqrt{n}$, hence each of them has to prepay $O(\log n)$ in potential. Therefore the amortized running time of insert is $O(\log n)$.

By construction, items appear in trees in their exact ascending order, therefore (using the result from part 3a), the access time is statically optimal.

Problem 3c. Given a data structure of the kind used in this problem, augment it by attaching a doubly linked list L_k to each S_k. L_k lists all items x in S_k for which S_k is the smallest tree they belong to. The items in L_k will be ordered chronologically, i.e. they are appended at the front every time they are inserted.

Given an arbitrarily, but fully (all subtrees are full except for maybe the last one), populated version of the data structure, define the *access* operation as follows.

To *access* item x, look for it in $S_0, S_1, \ldots$ until you find it in S_k, takes time $O(2^k)$. Insert x in each of $S_0, \ldots, S_{k-1}$ while removing the least recently added item from the corresponding trees (find that item using the chronological linked lists), in order to preserve the sizes of all trees, also done in $O(2^k)$. Update $L_0, L_1, \ldots, L_k$ accordingly in $O(\log \log k)$ time. Access and restructuring completes in total time $O(2^k)$.

Now consider the sequence A of m access operations and examine the average access time for some fixed element x with frequency p_x. The total number of accesses to x is $a_x = p_x m$. Name them $t_1, \ldots, t_{a_x}$. Let s_1 be the number of access operation in A before t_1, let s_1 be the number of access operations in A between t_1 and t_2, let s_2 be the number of access operations in A between t_2 and t_3, and so forth.

Due to our access algorithm, just before the i-th access, x will be located in $S_{\log \log s_i}$, and hence the access time of the i-th access will be $O(\log s_i)$ (see part 3a), and therefore the total access time for item x will be:

$$T_x = \sum_{i=1}^{a_x} \log s_i, \text{ subject to } \sum_{i=1}^{a_x} s_i \leqq m$$

Technically speaking, the access time for the first access should be counted as $O(\log n)$ because before the first access x might be in the largest subtree. However, static optimality analysis is only interesting in the case when n is fixed and m is asymptotic, therefore, we can safely regard the time for the first access to x as $\log s_i$ without influencing the order of the running time.

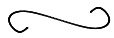
We want to pick the worst case configuration of $s_1, \ldots, s_{a_x}$ so we can compute the maximum possible T_x. Due to the concavity of the log-function, this is achieved when:

$$s_1 = \cdots = s_{a_x} = m/a_x = 1/p_x$$

And hence, the worst case total access time for x is:

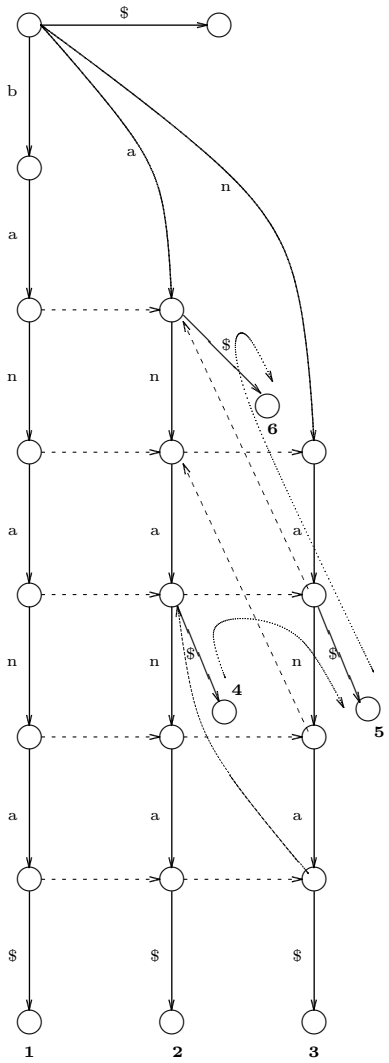
$$T_x = mp_x \log 1/p_x$$

Therefore the average running time for access to x is $O(p_x \log 1/p_x)$.



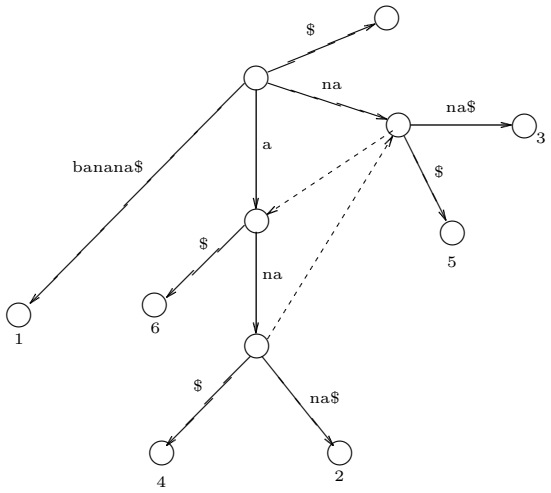
(page over)

Problem 4a. The insertion of the first 4 suffices is regular and it doesn't use any suffix links. The insertion of the 5-th and 6-th suffices is indicated with dotted lines below.



(page over)

Problem 4b. As noted in class, suffix links are not placed on leaf nodes:



Problem 5a. Denote by k_i the number of characters in T_i. We begin with a regular empty compressed suffix tree structure. And we first preprocess T_1 in the standard way.

Once we are done with T_1, we move the pointer back to the root of the suffix tree, and proceed to preprocess T_2 using the same suffix tree, but we force $s_{2,1}$, the first suffix of T_2 (i.e. the entire T_2 itself), to be inserted using a slowfind.

This takes $O(k_2)$ steps (in the worst case), but it is only done once so we are fine. This special step ensures that the insertion of $s_{2,1}$ doesn't violate the branching properties of the suffix tree. Once $s_{2,1}$ has been inserted, the rest of T_2's preprocessing can take advantage of the pre-exisiting suffix links. And the runtime analysis for one tree holds unchanged.

Repeat this procedure for the rest $T_3, \ldots, T_n$. By construction, the running time is then $O(k_1 + k_2 + \cdots + k_n)$.

Problem 5b. We perform a depth-first search on the subtree of N, and keep track of which kinds of $\$_i$ symbols we encounter on the way. If we encounter each kind at least once, this implies that the substring of N is to be found in all subtrees.

Problem 5c. Let each node in the common compressed suffix tree contain two mark bits m_1 and m_2, initially set to 0. Traverse the tree in post-order, and at each node flip the m_i bit to "1", if the node corresponds to a terminating character of type $\$_i$, or if either of its children has their m_i bit set.

By construction, a node's m_i bit is set if the substring (from root to node) of that node is present in T_i. Finally, we perform another depth-first traversal of the tree, which keeps track of the deepest node it encounters that has both of its mark bits set. That node corresponds to a maximal common substring. We are done in $O(|T_1| + |T_2|)$.