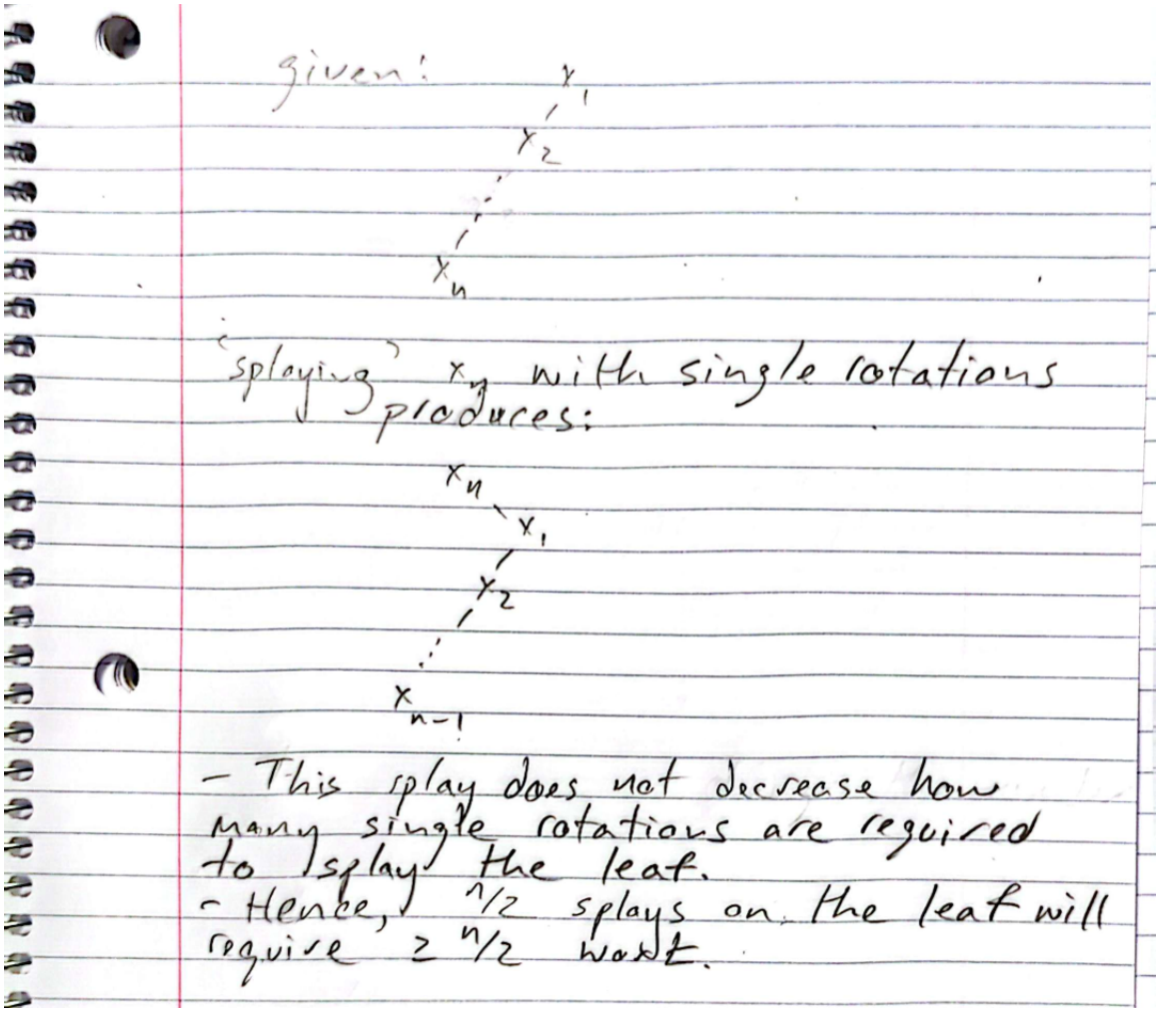
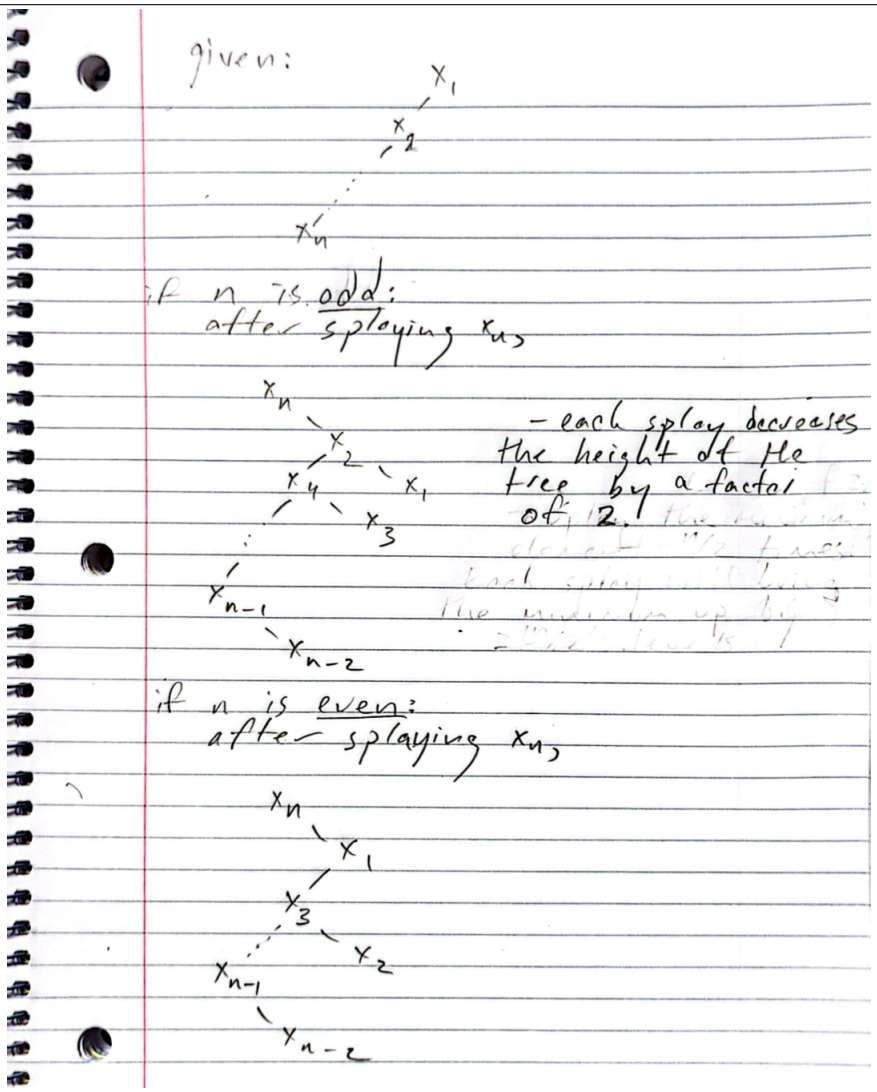
Contact: David Zhang – dzhang [at] cs [dot] toronto [dot] edu
All questions are taken from Karger's 6.854 Advanced Algorithms Homework (link):

1. Some Splaying Counterexamples

   (a) In class, we 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. To generalize this example, argue that there is a sequence of $n/2$ splays that each take at least $n/2$ work.



(b) Now, for the same starting tree, show the 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.



It is more accurate to say that the specific splay decreases the height of the tree rather than 'each' splay. However, any subsequent splay will be cheaper than if we had used single rotations.

(c) To quantify this, compute the splay tree potential change for each single and double rotation of the linked list described above.

Let $s(x)$ denote the size of the subtree rooted at node x. Let $r'(x)$ denote the rank of node x following a **single rotation** and let $r(x)$ denote its original rank. Suppose that x_i is the parent of x_n. Then, the difference in potential is

$$\begin{aligned} r'(x_n) - r(x_n) + r'(x_i) - r(x_i) &= \log(s(x_n)+1) - \log s(x_n) + \log(s(x_i)-1) - \log s(x_i) \\ &= \log(s(x_n)+1) - \log s(x_n) + \log s(x_n) - \log(s(x_n)+1) \\ &= 0 \end{aligned}$$

Let $r''(x)$ denote the rank of node x following a **double rotation** and suppose that x_{i-1} is its grandparent. Note that only the ranks of x_n and x_{i-1} change. Thus, the difference in potential in this case is

$$\begin{aligned} r''(x_n) - r(x_n) + r''(x_{i-1}) - r(x_{i-1}) &= \log(s(x_n)+2) - \log s(x_n) + 0 - (\log s(x_n) + 2) \\ &= -\log s(x_n) \end{aligned}$$

Hence, double rotations will simplify the data structure while single rotations will not.

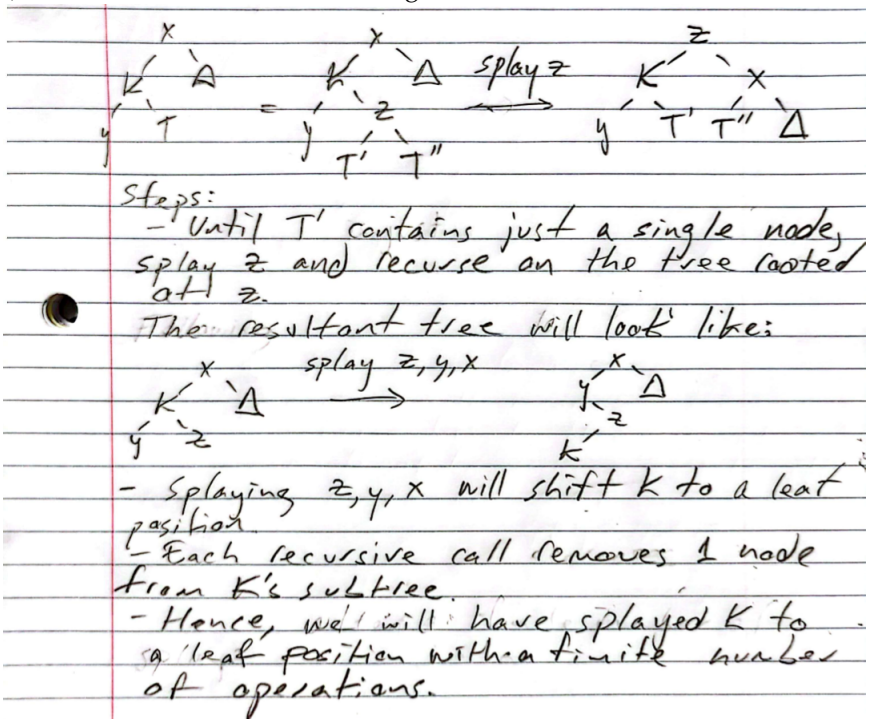
(d) Given the theorem about access times in splay trees, it is tempting to conjecture that splaying does not create trees in which it would take a long time to find an item. Show that this conjecture is false by showing that for large enough n, it is possible to restructure any binary tree on n nodes 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.

Hint: Start by showing how you can use splay operations to make a specified node into a leaf; then recurse.

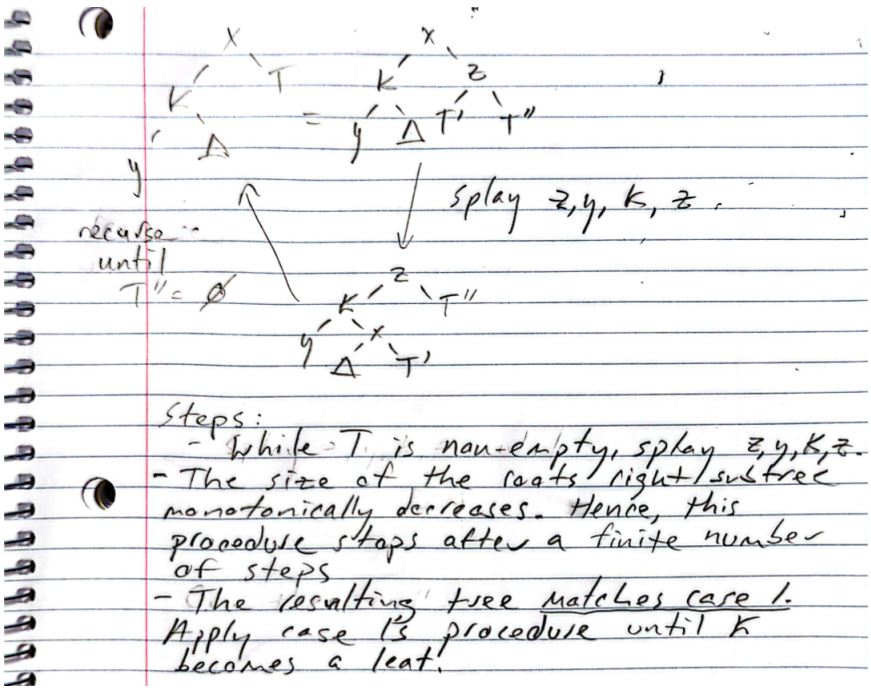
The first n for which the claim is true is $n = 4$. One can verify that given any binary search tree on 4 nodes, some sequence of spays will transform it into one of the other 13 possibilities.

Let the claim hold for $n \geq 4$. Suppose that we have an arbitrary binary search tree on $n + 1$ nodes, and let our desired tree (also on $n + 1$ nodes) be arbitrary. Pick a leaf K in our desired binary search tree and consider the non-trivial case where K is *not* a leaf in our given binary search tree. We will demonstrate a procedure to force K to a leaf position. The special cases of the procedure are as follows:

- If K is the minimum or maximum key in the tree, then splaying its successor (or predecessor) will push it to a leaf position.
- If K is the second smallest key in the tree, then its left node must exist, and it must be the minimum. In particular, splaying the minimum element or any of its ancestors will induce a series of zig-zig right rotations and maybe a final single right rotation. First, splay K's parent, x. Every right rotation (single or double) leaves K as the left child of its parent. Let $\triangle$ denote unimportant but present subtrees (including empty subtrees). After splaying x to the root, we will be in one of 2 cases (or their flipped variant): Either K has an element in its right subtree,

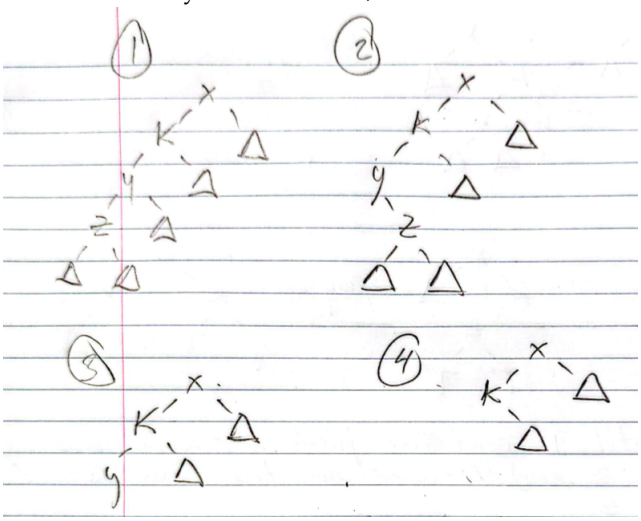


or x has an element in its right subtree. At least one of these cases must be true since $n + 1 \geq 5$.



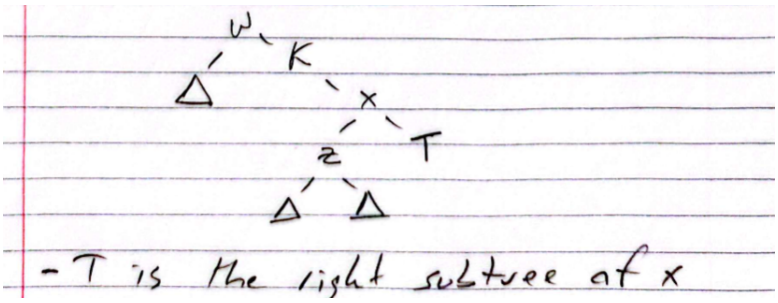
Both cases splay K to a leaf position.

If K is any other element, consider the case where it is the left child of its parent.



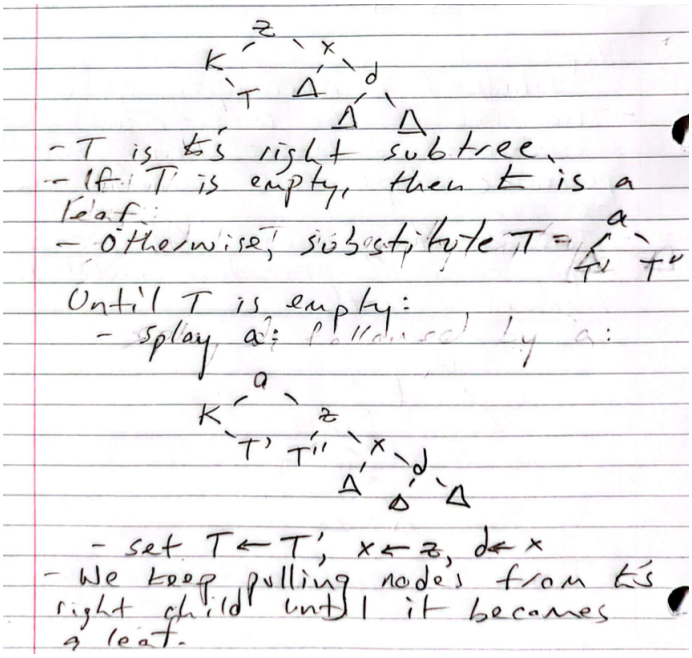
If K is the root, then splay its non-empty child to obtain (1)-(4) or their flipped variant. We can deal with the flipped versions of (1)-(4) symmetrically. Notice that

- (2) can be transformed into (1) by splaying z, K and then x. This sequence of splays works regardless of if x has a (left or right) parent or not.
- (3) can be transformed into (1) as well. Notice that y is not the minimum element (otherwise K would be the second-smallest element), so we cannot be in the leftmost path of the tree. Thus, x either has a left parent or a right parent. If x has a left parent, then splay x, K, y to obtain a flipped variant of (1). On the other hand, if it has a right parent then splay y, K, x to obtain an instance of (1). As it turns out, this sequence of splays works regardless of if x has a grandparent.
- (4) is slightly more interesting. If K's right subtree contains only a single node, then splay it to the root. Then, K will be a leaf. Otherwise, x must have a parent since K is not the minimum element. If x's parent is $w < x$, then splay K followed by w. If z is K's right child, then the resulting tree will be:



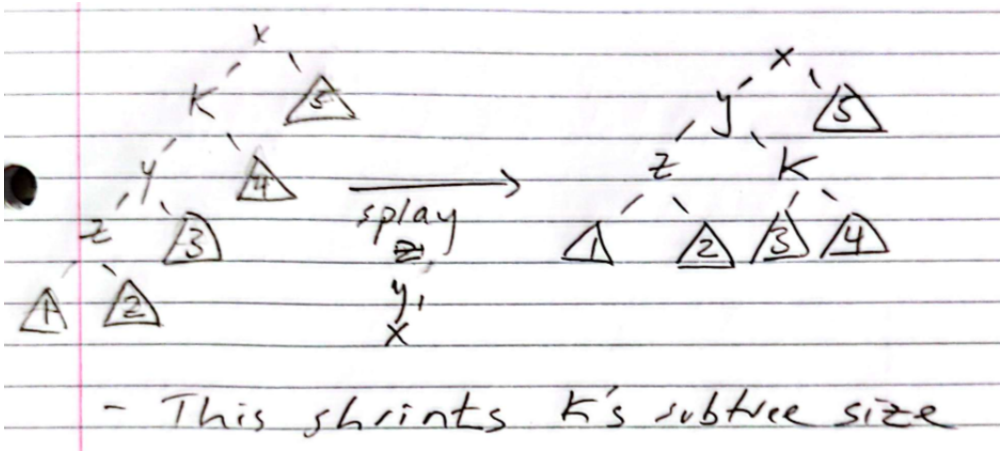
If T is empty, then we have (2). If T is not empty, then we have (1). Again, w's parent (or lack thereof) has no bearing on the result.

Conversely, if x's parent is $d > x$, then splay K followed by z. The resulting tree is:

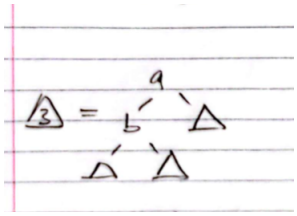


Therefore, we can either repeated splay K's child until K becomes a leaf node, or convert any tree into an instance of (1) or its inverse.

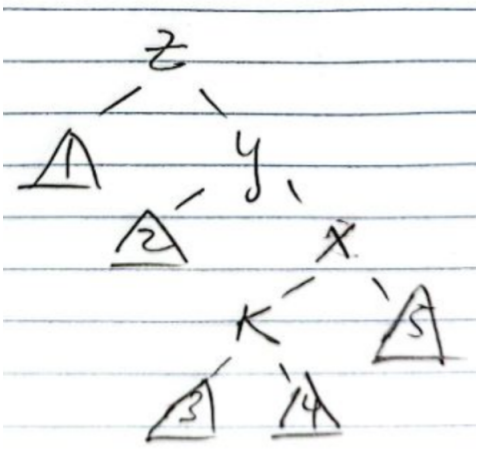
Given an instance of (1), we can strictly decrease K's subtree size until it only contains K, making K a leaf. We will do precisely this by splaying z, y and x.



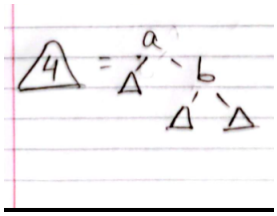
a) If K is a leaf following the procedure, then we are done.



b) If _____ then splay z:

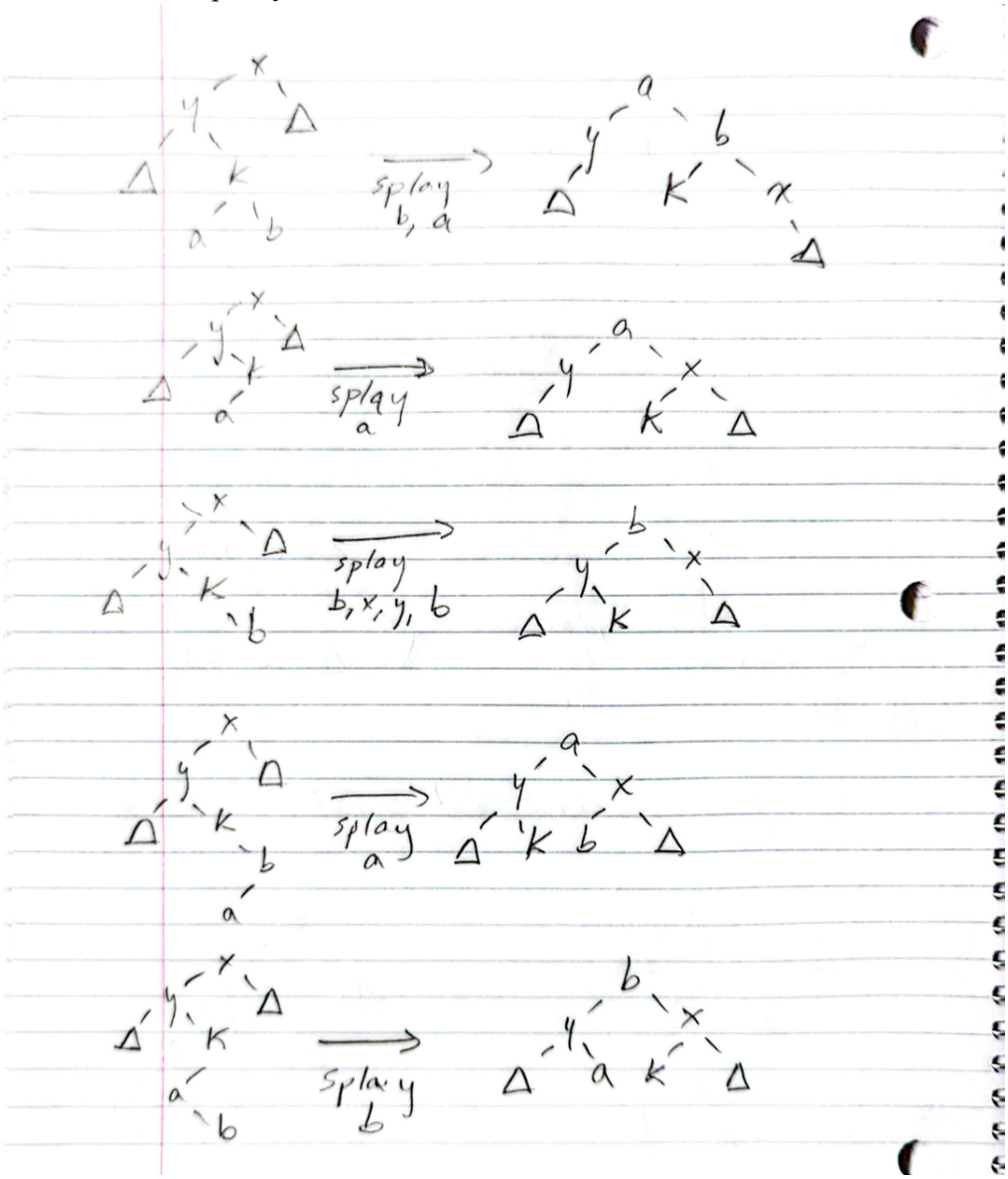


and recursively apply the procedure on x's subtree to further shrink K.



c) If _____ then keep the tree as-is and recursively apply the symmetric procedure on y's subtree.

All other cases require y's subtree to be as follows:



In each figure, we present a sequence of splay operations that sink K to a leaf position. Hence, any node can be made into a leaf by a sequence of splay operations.

Now that K is a leaf, recurse on the remaining n nodes to produce the desired tree. K is not in any path that subsequent splays take, so it remains a leaf. Further, K must be in the same position in our constructed tree and our desired tree; otherwise, there is some other node for which K is in one subtree in the constructed tree, and in the other subtree in the desired tree. This is a contradiction since K is either less than the node or greater than it. Hence, we have produced our desired tree using only splays.

2. In this problem, we're going to develop a less slick but hopefully more intuitive analysis of why splay trees have low amortized search cost. We need to argue that on a long search, all but $O(\log n)$ of the work from the descent and the follow-on splay is paid for by a potential decrease from the splay. We'll use the same potential function as before – the sum of node ranks (multiplied by some constant). Also as before, we'll analyze one double rotation at a time, and argue that the potential decrease for each double rotation cancels the constant work of doing that double rotation.

(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.

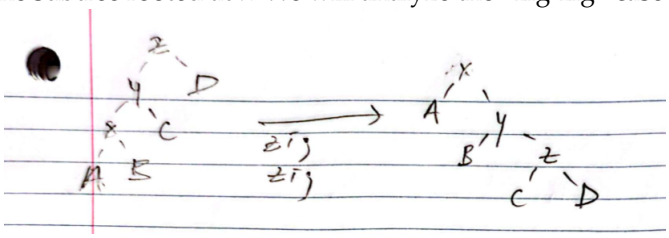
Suppose that there are k balanced triples in the search path. The root of the last balanced triple has at least 3 descendants; otherwise, it would not be a balanced *triple*. Further, k is maximized when all of the tree's n nodes form the search path (i.e. the tree is a chain), and each balanced triple decreases the number of descendants by a factor of 9/10.

$$\begin{aligned}(9/10)^k n &\geq 3\\ \log_{10/9}(9/10)^k + \log_{10/9} n &\geq \log_{10/9} 3\\ k &\leq \log_{10/9} n - \log_{10/9} 3\end{aligned}$$

Therefore, $k = O(\log n)$.

(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.

Let $r'(i)$ denote the rank of node i following the double rotation and let $r(i)$ denote its prior rank. Let $s(i)$ denote the size of the subtree rooted at i. We will analyze the "zig-zig" case first:



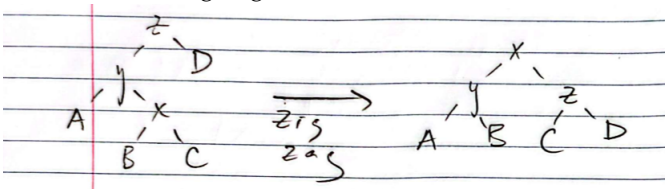
- $r'(x) - r(x) = \log(s(A) + s(B) + s(C) + s(D) + 3) - \log(s(A) + s(B) + 1) < \log \frac{s(A)+s(B)+s(C)+s(D)+3}{9/10(s(A)+s(B)+s(C)+s(D)+3)} = \log 10/9$
- $r'(y) - r(y) = \log \frac{s(B)+s(C)+s(D)+2}{s(A)+s(B)+s(C)+2} < \log \frac{s(B)+s(C)+s(D)+2}{9/10(s(A)+s(B)+s(C)+s(D)+3)+s(C)+1}$. We are interested in the worst-case change in potential, i.e. we want to maximize this difference in ranks. Observe that we maximize $r'(y)$ when A is sparse and D is dense. We will achieve this by setting $s(A) = 0$ and $s(C), s(D) < 1/10(s(A) + s(B) + s(C) + s(D) + 3)$. Then,

$$\begin{aligned}r'(y) - r(y) &< \log \frac{9/10(s(A) + s(B) + s(C) + s(D) + 3) + 2/10(s(A) + s(B) + s(C) + s(D) + 3)}{9/10(s(A) + s(B) + s(C) + s(D) + 3)}\\ &= \log 11/9\end{aligned}$$

- $r'(z) - r(z) = \log \frac{s(C)+s(D)+1}{s(A)+s(B)+s(C)+s(D)+3} < \log \frac{1/10(s(A)+s(B)+s(C)+s(D)+3)}{s(A)+s(B)+s(C)+s(D)+3} = \log 1/10$

Thus, the total change in potential in the "zig-zig" case is $\Delta\Phi < \log 10/9 + \log 11/9 + \log 1/10 = \log 11/81 < 0$.

Next, we will consider the "zig-zag" case:



- $r'(x) - r(x) = \log \frac{s(A)+s(B)+s(C)+s(D)+3}{s(B)+s(C)+1}$
- $r'(y) - r(y) = \log \frac{s(A)+s(B)+1}{s(A)+s(B)+s(C)+2}$
- $r'(z) - r(z) = \log \frac{s(C)+s(D)+1}{s(A)+s(B)+s(C)+s(D)+3}$

Therefore, the total change in potential in the "zig-zag" case is

$$\Delta\Phi = \log \frac{(s(A)+s(B)+1)(s(C)+s(D)+1)}{(s(B)+s(C)+1)(s(A)+s(B)+s(C)+2)}$$

We can show that $(s(A)+s(B)+1)(s(C)+s(D)+1) < \frac{(s(A)+s(B)+s(C)+s(D)+3)^2}{2}$ by expanding. Hence,

$$\begin{aligned}
\Delta\Phi &< \log \frac{(s(A)+s(B)+s(C)+s(D)+3)^2}{2(s(B)+s(C)+1)(s(A)+s(B)+s(C)+2)} \\
&< \log \frac{(s(A)+s(B)+s(C)+s(D)+3)^2}{2 \cdot 9/10(s(A)+s(B)+s(C)+s(D)+3) \cdot 9/10(s(A)+s(B)+s(C)+s(D)+3)} \\
&= \log 50/81 \\
&< 0
\end{aligned}$$

(c) Argue that when a balanced triple is rotated, the potential increases by at most $2(r(z)-r(x))$ (again consider both rotation types).

We will analyze the "zig-zig" case first.

- $r'(x) - r(x) = \log \frac{s(A)+s(B)+s(C)+s(D)+3}{s(A)+s(B)+1}$
- $r'(y) - r(y) = \log \frac{s(B)+s(C)+s(D)+2}{s(A)+s(B)+s(C)+2}$
- $r'(z) - r(z) = \log \frac{s(C)+s(D)+1}{s(A)+s(B)+s(C)+s(D)+3}$

The total change in potential in this case is $\Delta\Phi = \log \frac{(s(B)+s(C)+s(D)+2)(s(C)+s(D)+1)}{(s(A)+s(B)+1)(s(A)+s(B)+s(C)+2)}$. We can show that $(s(B)+s(C)+s(D)+2)(s(C)+s(D)+1) \leq (s(A)+s(B)+s(C)+s(D)+3)^2$ by expanding. Hence,

$$\begin{aligned}
\Delta\Phi &\leq \log \frac{(s(A)+s(B)+s(C)+s(D)+3)^2}{(s(A)+s(B)+1)(s(A)+s(B)+s(C)+2)} \\
&< \log \frac{(s(A)+s(B)+s(C)+s(D)+3)^2}{(s(A)+s(B)+1)^2} \\
&= 2(\log(s(A)+s(B)+s(C)+s(D)+3) - \log(s(A)+s(B)+1)) \\
&= 2(r(z)-r(x))
\end{aligned}$$

Next, we will analyze the "zig-zag" case. From 2 (b),

$$\Delta\Phi = \log \frac{(s(A)+s(B)+1)(s(C)+s(D)+1)}{(s(B)+s(C)+1)(s(A)+s(B)+s(C)+2)} < \log \frac{(s(A)+s(B)+s(C)+s(D)+3)^2}{(s(B)+s(C)+1)^2} = 2(r(z)-r(x))$$

(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.

The $O(1)$ loss in potential pays for the real cost of rotating a biased triple. There are $O(\log n)$ balanced triples, so the actual cost of performing every balanced rotation is $O(\log n)$. The potential increase induced by all the balanced rotations is at most $2\sum_{\text{root } z\text{, grandchild } x \text{ of balanced triple } i} r(z) - r(x)$. Notice that $r(z) \leq \log((9/10)^i n)$. We simplify our sum as follows:

$$\begin{aligned} 2 \sum_{\text{root } z\text{, grandchild } x \text{ of balanced triple } i} r(z) - r(x) &\leq 2 \sum_{i}^{O(\log n)} \log((9/10)^i n) \\ &= 2\log\left(\frac{1-(9/10)^{O(\log n)}}{1/10} n\right) \\ &= 2\log(O(n)\cdot n) \\ &= O(\log n) \end{aligned}$$

Hence, performing $O(\log n)$ balanced rotations adds $O(\log n)$ potential to the system. Charging each splay operation $O(\log n)$ is sufficient to amortize its cost.

3. Let S be a search data structure (such as a red-black tree) that performs insert, delete and search in $O(\log n)$ time, where n is the number of elements stored. An empty data structure S can be created in $O(1)$ time.

   We would like to construct a static data structure with n elements that is statically optimal in total access time, given the number of times an element is accessed in an access sequence. Recall that if item i is accessed $p_i m$ times in a sequence of m operations then the static optimal access time is $O(m \sum p_i \log 1/p_i)$.

   The data structure is constructed as follows. Search data structure S_k contains the 2^{2^k} most frequently occurring items in the access sequence (note this means items may be in multiple trees). A search on v is done on $S_0, S_1, \ldots$ until an S_i holding v is encountered. Notice that all elements in S_i are held in S_{i+1}.

   (a) Show that the above data structure is asymptotically comparable to the optimal static tree in terms of the total time to process the access sequence.

Consider a search for item x. Suppose that the first bucket that x appears in is j, i.e. x is less frequent than $2^{2^{j-1}}$ items. By total probability, the least frequent item in bucket $j-1$ can appear at most $\frac{1}{2^{2^{j-1}}}$ of the time. In particular, we derive the following bound on p_x: $p_x < \frac{1}{2^{2^{j-1}}}$. By implication, $2^{j-1} < \log 1/p_x$ so $2^j = O(\log 1/p_x)$.

Since we do not create unused buckets and there must be sufficient space in the final bucket to store all n elements, the total number of buckets is $\Theta(\log\log n)$. The total time requirement to binary search $S_0, \ldots, S_j$ is $\sum_{i=0}^{j} O(\log 2^{2^i}) = \sum_{i=0}^{j} O(2^i) = O(2^j)$. We perform $p_x m$ searches on x, so searching for each element takes time $O(p_x m 2^j) = O(p_x m \log 1/p_i)$. Hence, the total time required to process the entire access sequence is $O(m \sum p_i \log 1/p_i)$.

(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).

Augment the search structure with min heaps keyed by item frequencies. On insert, if $n > 2^{2^k}$, then create a new heap (and search tree) on the n elements. Then, compare v with the minimums of heaps 0, 1, ... until the first heap j whose $min_j < p_v$. For buckets $j, \ldots, k$, if the heap is not full then insert v to it and its corresponding search tree. Otherwise, if the heap is full then evict its minimum frequency element and insert v to both the heap and the search tree. This procedure will preserve the property that S_i contains the 2^{2^i} most frequently occurring elements.

Notice that we only create a heap and search tree until after $2^{2^k} - 2^{2^{k-1}}$ elements are inserted to the k-th bucket. Heap creation and tree walks take time linear in the their size, so $O(2^{2^k})$ time is spread across $2^{2^k} - 2^{2^{k-1}} = 2^{2^{k-1}}(2^{2^{k-1}} - 1)$ inserts. By charging each insert an additional $O(1)$, we can fully pay for this data structure's initialization time. The actual time required for the procedure to traverse buckets $0, \ldots k$ is $\sum_i^k O(2^i) = O(2^k) = O(\log n)$ since once again, $k = \Theta(\log \log n)$. Searches retain the same time requirement as before and inserts are $O(\log n)$ amortized.

(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 (but of course, had to be built special case).

Use the same heap augmentation, still keyed by frequencies. Specifically, associate with each node, the number of occurrences that we have seen of that element thus far. On inserting x, start at the last bucket (the one containing the most elements) and query whether x is in the structure. If it is, then increase its key. Otherwise, insert x with an initial frequency of 1. Continue by visiting buckets in reverse order (by decreasing size). If x is in a bucket, then increase its key by 1. If x is not in a bucket, then check if x's frequency is greater than the bucket's minimum frequency. If x is less frequent, then terminate the operation; otherwise, evict the bucket's minimum and replace it with x. Perform a similar process on searches. All operations retain the same time complexities.

Consider the time required to perform access $l \leq m$ on item i. Let occ_i denote the number of times that i has been accessed thus far (on insert, occ_i is 1). The current access frequency of i is $\frac{occ_i}{\ell}$. Suppose that item i is in bucket j. This means that i's access frequency is less than the minimum frequency in bucket $j-1$. The minimum access frequency of bucket j is at most $1/2^{2^{j-1}}$ so $occ_i/\ell < 1/2^{2^{j-1}}$. Hence, $2^j = O(\log \ell/occ_i)$ and the total time to access item i on access ℓ is $O(\log \ell/occ_i)$.

The total access time is therefore

$$\sum_{\text{access } l \leq m \text{ on item } i} O(\log \ell/occ_i)$$

Notice that occ_i is the final number of access to item i after processing the whole sequence. On each access, we add a corresponding $O(\log \ell/occ_i)$ term and increment occ_i. Thus, this log sum is the factorial of total accesses and final item occurrences. Note that, after processing the whole sequence, $occ_i = mp_i$.

$$\begin{aligned}
\sum_{\text{access } l \leq m \text{ on item } i} O(\log \ell/occ_i) &= O\left(\log \frac{m!}{\prod_i^m occ_i!}\right) \\
&= O\left(\frac{\log m^{1/2+m}(1/e^m)}{\prod_i^m (mp_i)^{1/2+mp_i}(1/e^{mp_i})}\right) \quad (m! \approx \sqrt{2\pi m}(m/e)^m) \\
&= O\left(\log \frac{m^{1/2+m}}{\prod_i^m (mp_i)^{1/2+mp_i}} + \log \frac{1}{e^m \prod_i^m (1/e^{mp_i})}\right) \\
&= O\left(\log m^{1/2+m} - \log \prod_i^m (mp_i)^{1/2+mp_i} + \log \frac{1}{e^m (1/e^{m\sum_i^m p_i})}\right) \\
&= O\left(m \log m - \sum_i^m \log(mp_i)^{1/2+mp_i}\right) \\
&= O\left(m \log m - (\sum_i^m mp_i \log m - mp_i \log 1/p_i)\right) \\
&= O(m \sum_i^m p_i \log 1/p_i)
\end{aligned}$$

Thus, this data structure satisfies the static optimality theorem of splay trees.

(d) Make your data structure satisfy the working set theorem on splay trees. Ignore the static optimality condition.

Instead of a heap, maintain a doubly linked list in LRU order in each bucket so that bucket i has both a $O(\log n)$ search structure and list, each containing the 2^{2^i} least recently accessed items. Maintain pointers between corresponding nodes in the tree and in the linked list. Suppose that we want to access item x. If the first bucket that x appears in is j, and it was accessed t_x operations ago, then it must be more recent than any of the $2^{2^{j-1}}$ least recently accessed items; otherwise, it would have been in bucket $j-1$. Thus, $2^{2^{j-1}} < t_x + 1$ so $2^j = O(\log(t_x+1))$.

On an access to x, first search buckets $0, \ldots$ until j. Like before, this takes time $O(2^j) = O(\log(t_x+1))$. Then, insert a copy of x to bucket k (the last bucket). Then, for buckets $i = j$ to $k-1$,

- Suppose that y precedes and z succeeds x in bucket i's linked list (the logic is the same if y or z is Nil).
- Delete x from bucket i by pointing y and z to each other.
- Search for the tail of bucket i's linked list in bucket $i+1$ and obtain a pointer to its successor w.
- Insert a copy of w as bucket i's tail.

We have put x in the most recently used position and updated all the buckets that contained x. This loop over buckets j to $k-1$ takes time $\sum_i^k O(2^i) = O(2^k) = O(\log n)$. Hence, the total time required to process the whole access sequence is $O(\sum_x(\log(t_x+1)+\log n)) = O(n \log n + \sum_x \log(t_x+1))$, which matches the working set theorem on splay trees.

4. The following data structure is useful in efficiently implementing the Lin-Kernighan heuristic for the travelling salesman problem, which requires repeatedly reversing arbitrary sub-intervals of a sequence.

Describe a data structure that represents an ordered list of elements under the following three types of operations:

Access(k): Return the kth element of the list (in its current order).

Insert(k, x): Insert x (a new element) after the kth element in the current version of the list.

Reverse(i, j): Reverse the order of the ith through jth elements.

For example, if the initial list is $[a, b, c, d, e]$, then Access(2) returns b. After Reverse(2,4), the represented list becomes $[a, d, c, b, e]$ and then Access(2) returns d.
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.

Hint: First consider how to implement Access and Insert using splay trees. Then think about a special case of reverse in which the $[i, j]$ range is represented by a whole subtree. You can represent the reversal of that subtree by storing a "reverse" bit in the root node telling you to traverse that subtree "backwards." Generalize this idea to solve the real problem. Remember, extra information stored in the tree must be maintained under restructuring operations.

To index an element, assign each node a $.size$ attribute denoting the size of its tree. Assume that empty subtrees have $.size = 0$. Give each node a $.reverse$ bit denoting whether its range is reversed. Initially set this bit to 0. Also, give each node a $.cleft$ and $.cright$ attribute. If an even number of ancestors have their $.reverse$ bit set, then keep $.cleft = .left$ and $.cright = .right$. Otherwise set $.cleft = .right$ and $.cright = .left$.

Replace $.left$ and $.right$ with $.cleft$ and $.cright$ in all operations. We incur $O(1)$ cost at each level of the descent to sum $.reverse$ bits. We perform **Access(k)** by using the $.size$ attribute to index and **Insert(k, x)** is identical, except that we use $.cleft$ and $.cright$ instead of $.left$ and $.right$. To perform **Reverse(i, j)**:

- Split the tree at index j. Take the subtree containing indices $\leq j$ and split it at index $i-1$ if $i \geq 2$. If $i = 1$, then ignore this second split. If we performed the second split, then the right subtree of the node corresponding to index $i-1$ contains the range $[i, j]$ and if we only split once, then the tree rooted at index j contains the range $[1, j] = [i, j]$. Next, flip the $.reverse$ bit of the tree containing the range $[i, j]$.

- Reconstruct the tree by making the $[i, j]$ tree the right child of the node corresponding to index $i - 1$, if it exists. Splay the minimum element of the tree containing indices $> j$ (from the first split) and set its left child to be the tree containing $[1, j]$.

Note that all tree traversals are paid for by a subsequent splay. We trivially maintain the $.size$ attributes during splay operations. We also update $.reverse$ bits during splays by ensuring that each node involved retains its parity with respect to the number of ancestors with flipped bits. Splaying is also performed using $.cleft$ and $.cright$.