Massachusetts Institute of Technology
6.854/18.415: Advanced Algorithms
David Karger

Handout 7
September 21, 1999

Problem Set 2 Solutions

Problem 1. The init operation declares an array A of size n and an array B of size n. Each element $A[i]$ is a (x, j) tuple, where j an index to array B. Each element in B holds an index to A.

Definition 1 *An item with index j* belongs *to array B iff $0 \leq j \leq B_{max}$.*

Definition 2 *An item with index i* belongs *to array A holding (x, j) iff*

(a) j belongs to array B.

(b) $B[j] = i$

Init sets B_{max} to 0. Let (x, j) be the tuple held in $A[i]$. Get(i) returns x if $A[i]$ belongs to array A. Set(i, y) stores (y, j) in $A[i]$, if $A[i]$ belongs to array A. Otherwise, set(i, y) increments B_{max}, stores (y, B_{max}) in $A[i]$ and stores i in $B[B_{max}]$,

Lemma 1 *An item with index i belongs to A iff it has been set by some operation after which no init happened.*

Proof. Once init sets B_{max} to 0, all items do not belong to array A anymore. If set(i, x) happens on index i such that i does not belong to array A, a new entry is created in B and index i is made valid. If such a set operation does not happen, there is no entry belonging to B that holds index i. According to above definitions, the item does not belong to array A.

Lemma 2 *The above data structure is correct and performs all operations in constant time.*

Proof. Get returns empty if an item does not belong to array A. From lemma 1, an item belongs to array A only if it has been set after the previous init operation. Thus the data structure is correct. It is evident that each operation takes constant time

Problem 2. Let m be the number of accesses made, and let $p(x) \cdot m$ be the number of accesses made to item x. The access time has a information theoretic lower bound of $\Omega(m \sum_x -p(x) \log p(x))$. It takes $\Omega(m)$ to process the sequence. Therefore the optimal access time is $\Omega(m + m \sum_x -p(x) \log p(x))$.

(a) Search data structure S_k holds 2^{2^k} most frequently accessed items.

Lemma 3 *The search data structure is statically optimal.*

Proof. There are at most $1/p(x)$ items with more access frequency than x. Therefore x must belong to an S_k such that

$$2^{2^{k-1}} < 1/p(x)$$

i.e., $2^k < 2(1 - \log p(x))$. Therefore the search time in S_k is $O(2^k) = O(1 - \log p(x))$. The search time in smaller S_i's is $O(2^0 + 2^1 + \cdots + 2^{k-1})$ which is $O(2^k)$. So the total access time is $O(m + m \sum_x -p(x) \log p(x))$ which matches the lower bound.

(b) We make the data structure dynamic. S_k now holds the 2^{2^k} most frequently accessed items that have been accessed at least once previously. The search data structure is still optimal in search time since S_k still holds at least 2^{2^k} most frequently accessed items that can be accessed by the subsequent search.

The items in S_k are also organized in a search tree in the increasing order of access frequencies. It can be seen that every insert or delete operation in S_k will still take $O(2^k)$ time.

Item x in inserted in S_i if $p(x)$ of x is more than the minimum access frequency in S_i. If the bucket S_i is full, the item with minimum access frequency is deleted. Notice that the deleted item will be present in a higher S_j data structure.

A new S_{l+1} needs to be created if S_l cannot hold all elements after an insert. The creation of this level costs $O(n \log n)$ time. We will now show that the cost of insert is $O(\log n)$ amortized.

Lemma 4 *The amortized cost of insert operation is $O(\log n)$.*

Proof. The cost of insertions in each level is

$$O(2^0 + 2^1 + \cdot + 2^l) = O(2^l) = O(\log n)$$

since $2^{2^l} \geq n$. The cost of creating a new level is $O(n \log n)$. But we have to create a new level only if $n = 2^{2^l}$. We define the potential function

$$\phi = 2^{l+1} \cdot \#\text{ elements in } S_l - 2^{2^{l-1}}$$

where S_l is the last search data structure. The change in potential if a new level is not created is only 2^{l+1}. The change is potential if a new level is created is

$$2^{l+1}(2^{2^l} - 2^{2^{l-1}}) \geq 2^l \cdot 2^{2^l} = n \lg n$$

which pays for the cost of creating a new level.

(c) Recall that in (b), the access frequencies were organized in a search tree for each S_k. The data structure now updates values in the search tree on accesses and maintains the current access frequency of every element in S_k.

Lemma 5 *The dynamic online data structure is statically optimal.*

Proof. The cost of the jth search is $O(\log(j/f(x,j)))$, where $f(x,j)$ is the current access frequency of item searched. Therefore the total time to process the access sequence is

$$\begin{aligned} T(m) &= \sum_x O(\log(j/f(x,j))) \\ &= O(\log(m!/\prod_x (mp(x))!)) \end{aligned}$$

Let us denote $mp(x)$ by m_x. Note that $\sum_x m_x = m$. By plugging in the Stirling approximation of factorials, we get

$$\begin{aligned} T(m) &= O\left(\log \frac{m^{m-1/2}e^{-m}}{\prod_x m_x^{m_x-1/2}e^{-m_x}}\right) \\ &= O\left(\log \frac{m^m}{\prod_x m_x^{m_x}} + \sum_x \log m_x\right) \\ &= O\left(\log \frac{m^m}{\prod_x m_x^{m_x}} + m\right) \end{aligned}$$

since $\sum_x \log m_x = O(m)$.

(d) Instead of holding the most frequently accessed items, we hold the most recently accessed item. We can replace the search tree on access frequencies by a doubly linked list holding the items in LRU order. The proof that working set theorem is satisfied is similar to lemma 3.

Problem 3. We augment every node x in the splay tree with the number $x.desc$ of descendants (including itself) and a reverse bit $x.reverse$. No key needs to be maintained.

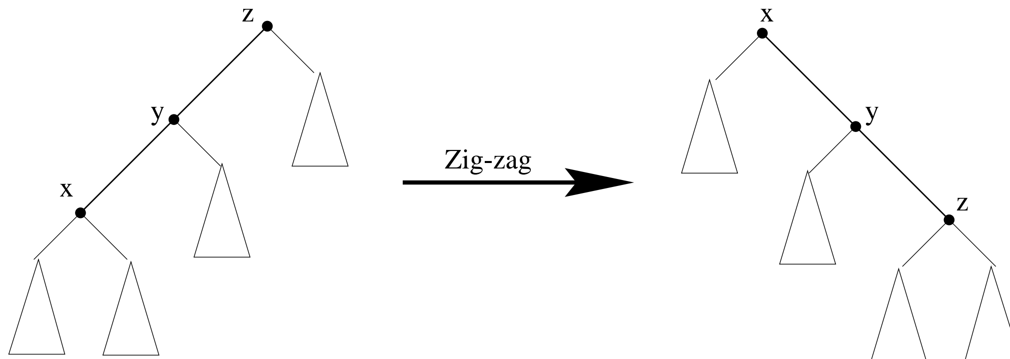
Each node x has a minor child $x.minor$ and a major child $x.major$. The left child $x.left$ is the minor child and the right child $x.right$ is the major child if *an even number of ancestors (including itself) have their reverse bit set.* Otherwise $x.right$ is the minor child and $x.left$ is the major child.

An in-order traversal $Trav(x)$ on node x is defined as $Trav(x.minor) + x + Trav(x.major)$. We ensure the invariant that $Trav(t)$, where t is the root, is the list of elements in order.

When splay tree operations are performed, the notion of left and right children is replaced with that of minor and major children. The minor and major children of a node x can be identified by looking at the reverse bits of its ancestors. This computation can be done when a search for x is performed.

It is evident that all splaying operations preserve $Trav(t)$ if we update the reverse bit appropriately. For example in Figure Problem 3, the reverse bit of z is modified $z.reverse \oplus x.reverse \oplus y.reverse$, where $\oplus$ denotes the exclusive-or operation. Similarly, the value of number descendants can be updated on rotations. For example in Figure Problem 3, the value of $z.desc$ is updated to $1 + y.major.desc + z.major.desc$.

The potential function argument works for the data structure as it does for splay trees except when a reverse bit is flipped. When a reverse bit $x.reverse$ is flipped, the major and minor children are flipped for all the descendants of x. However this does not change the potential $\sum_x r(x)$.



Therefore we can perform splay operation correctly in $O(\log n)$ amortized time. Split and join operations can be defined on our structure. The removal or addition of a root only causes changes to the new root.

We can perform $access(k)$ by a search based on $desc$ field. Operation $insert(k, x)$ is done like a splay tree insert, using split and join. The $reverse(i, j)$ involves flipping $x.reverse$ where x is the subtree containing the range $[i, j]$ as its descendants. To obtain an x of this form, we split at i and then at j. We now have x as the root of a splay tree. After flipping $x.reverse$, the three trees can be joined.

Problem 4. Given the value $t.key$ of root node t, the successor operation cost as much as a splay. Therefore, the operation has amortized cost $O(\log n)$.

We can improve the bound using the scanning theorem. First we will show that we can assume the splay tree to be have a single node as the left child of root

Lemma 6 *If no splay operations are done on the subtree S of descendants of node x, the subtree S can be replaced by a leaf node y.*

Proof. Follows from definition of splay operation.

So the complexity of operations are exactly the same as having a single node left child of root. Let m be the number of elements with key greater than the root. The total time to scan $m+2$ nodes of the tree takes $O(m)$ time. So successor takes $O(1)$ amortized time if it is applied repeatedly until the rightmost node is reached.

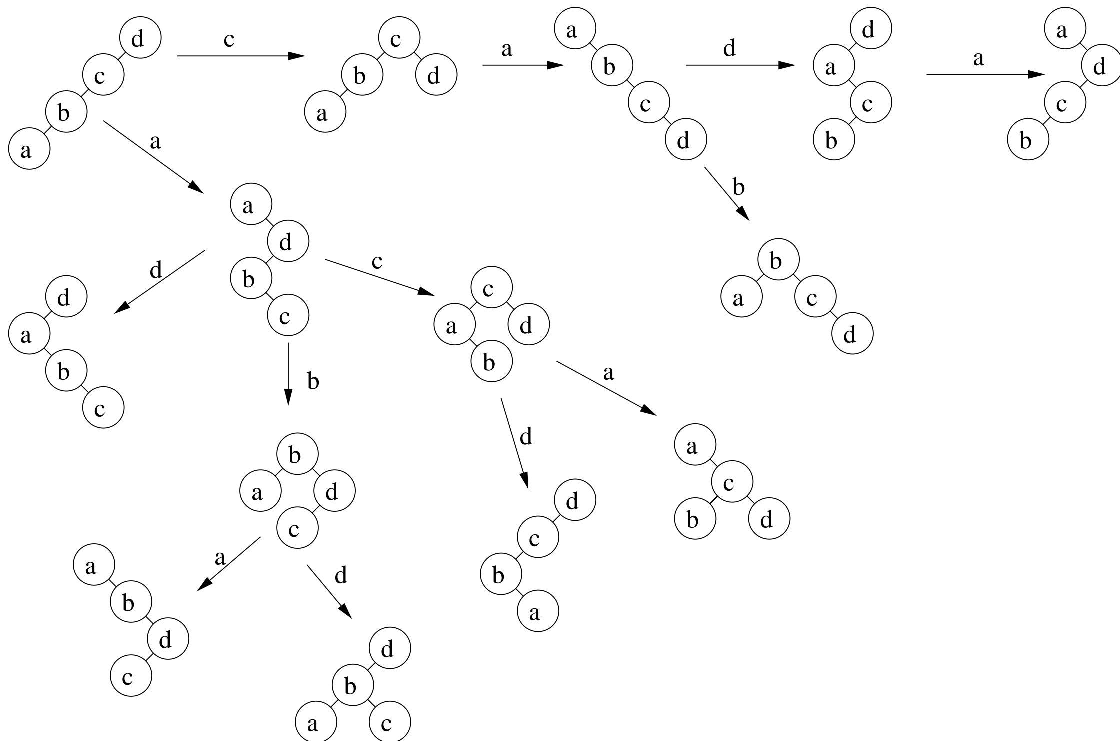
Problem 5. Observe first that the claim in the question is not true for $n = 3$; it is not possible to turn a zig-zig into a zig-zag by splaying (try it).

Claim: For $n \geq 4$, it is possible to turn any n node binary search tree into any other by a sequence of splay operations.

Proof:

We will prove this claim by induction on n.

Base case: $n = 4$. We can turn the tree into a left path by splaying on the items in order. (It is easy to show this for all n by induction. The key observation is that the last step of each successive splay must be a zig or zig-zag, which pushes the root onto the left path.) This is true for all n It remains to check that we can turn a left path into anything:



Inductive step: We need to show that if it is possible to restructure any $n-1$ node binary search tree into any other by a sequence of splay operations then the same is true for any n node binary search tree.

We will accomplish this goal via the following four lemmas:

Lemma 7 *Any node in a binary search tree with ≥ 4 nodes can be moved to a leaf position by an appropriate sequence of splay operations.*

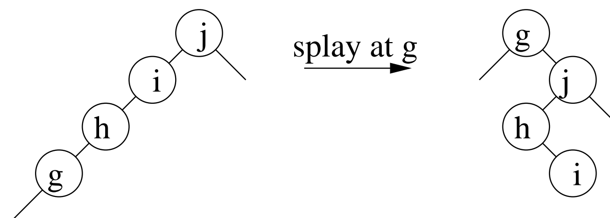
Lemma 8 *A leaf node will remain a leaf node under a sequence of splay operations if it is not splayed.*

Lemma 9 *The structure of the tree containing the descendants of a node that is splayed has no effect on the structure of the tree that results.*

Lemma 10 *No two binary search trees on n nodes differ only in the position of one leaf node.*

By Lemma 7 we can pick a node that is to become a leaf in the final tree and make it a leaf. Now Lemmas 8 and 9 say that this leaf will stay a leaf if we splay the other nodes, and will not affect the results of splaying on the other nodes. Thus by the inductive hypothesis we know that we can restructure the other $n-1$ nodes to match the desired tree. Finally, by Lemma 10 we know that we have gotten the desired tree.

Proof of 7. Let i denote the item we wish to turn into a leaf. If i is the minimum item we can turn it into a leaf by splaying on i and its successor. If i is the maximal element we can handle it symmetrically. If i is not the second element, splay i's predecessor's predecessor, i's, predecessor, i, and i's successor, giving the following situation:



If i is the second element we can handle it symmetrically. (Splay succ(succ(i)), succ(i), i, pred(i), and then succ(succ(i) again.)

Proof of 8. It is clear from the definition of splaying that no leaf node is ever given a descendant unless it is splayed.

Proof of 9. It is clear from the definition of splaying that descendants of a splayed node have no effect on the result of the operation.

Proof of 10. Suppose two binary search trees differed only in the position of one leaf node. Then the path from the root to the leaf differs in these two trees. Look at the place where it first differs. In order for the path to go left at this point the leaf must be less than this node; in order for the path to go right the leaf must be greater than this node. It is impossible for both of these to happen. Contradiction.

Several people misinterpreted this question by assuming that they could just apply the zig, zig-zig, and zig-zag cases at will. A splay operation applies the three cases as appropriate until the item is at the root. So splay(x) *always* brings x all the way to the root. Thus you cannot just splay in subtrees, and inversion of splays is difficult. (This theorem implies that you can invert splays, but you can't use this theorem to prove itself.)