Massachusetts Institute of Technology — Handout 6
6.854J/18.415J: Advanced Algorithms — Wednesday, September 19, 2012
David Karger

Problem Set 2 Solutions

Problem 1. **(a)** Here's the tree after splaying the leaf:

```
            x[n]
              \
               x[1]
              /
             .
            .
           .
          /
     x[n - 3]
        /
   x[n - 2]
    /
x[n - 1]
```

Repeatedly splaying the leaf $n/2$ times will cost $\geq n/2$ units of work each time.
If we use zig-zig double rotations instead, the final structure is (when n is even):

```
     x[n]
     /
  x[2]
  /  \
x[1] x[4]
     /  \
   x[3] x[6]
        /  \
      x[5]  .
             .
              .
               \
             x[n - 2]
             /     \
        x[n - 3] x[n - 1]
```

(b) This splay has improved the tree by reducing the heights of $n - o(1)$ nodes by a factor of about 2. Subsequent splays will take much less time than the first one because of this height reduction.

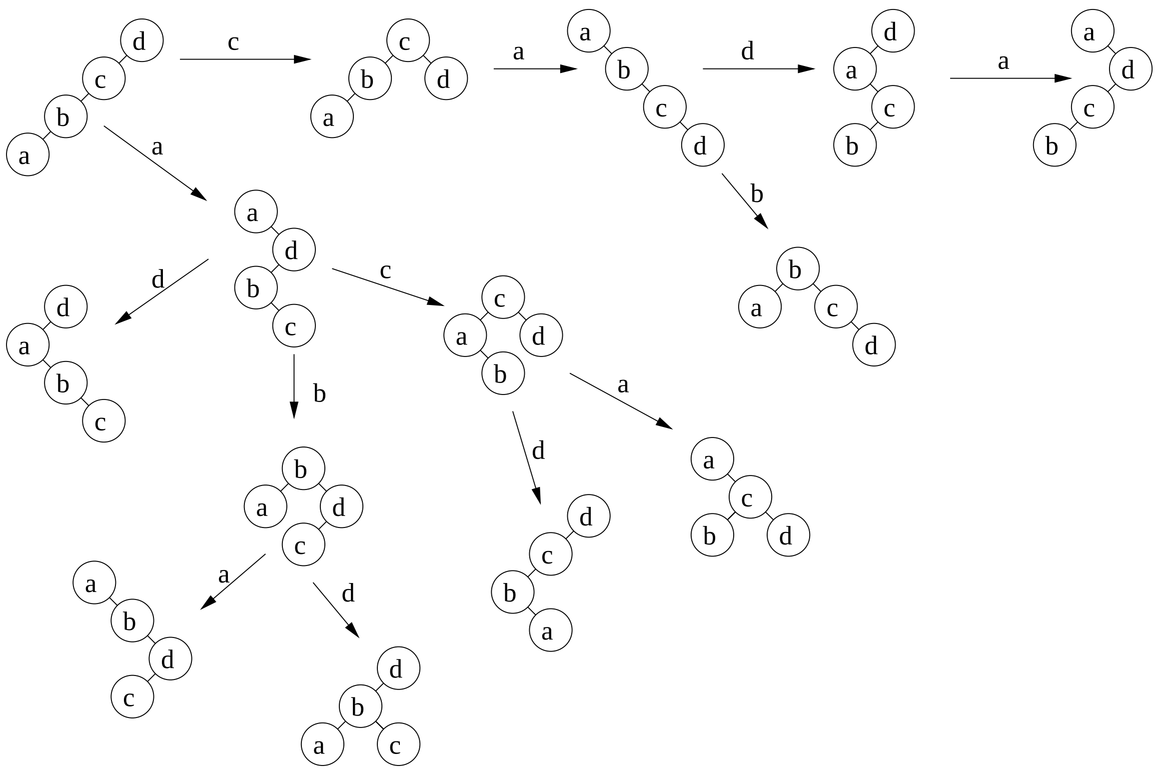
(c) Observe first that the claim in the question is not true for $n = 3$; it is not possible to turn a zig-zig into a zig-zag by splaying (try it).

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

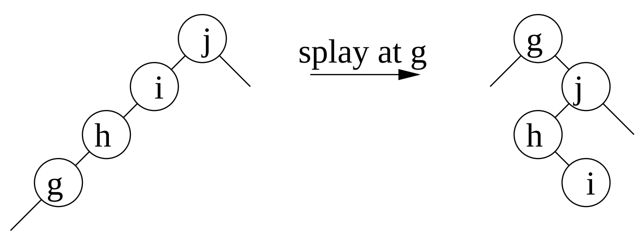
Lemma 2 *A leaf node will remain a leaf node under a sequence of splay operations if it is not splayed.*

Lemma 3 *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 4 *No two binary search trees on n nodes differ only in the position of one leaf node.*

By Lemma 1 we can pick a node that is to become a leaf in the final tree and make it a leaf. Now Lemmas 2 and 3 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 4 we know that we have gotten the desired tree.

Proof of 1. 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 2. It is clear from the definition of splaying that no leaf node is ever given a descendant unless it is splayed. ■

Proof of 3. 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 4. 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. ■

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 5 *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 6 *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 7 *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 $\sum_x \log m_x = O(m)$. ■

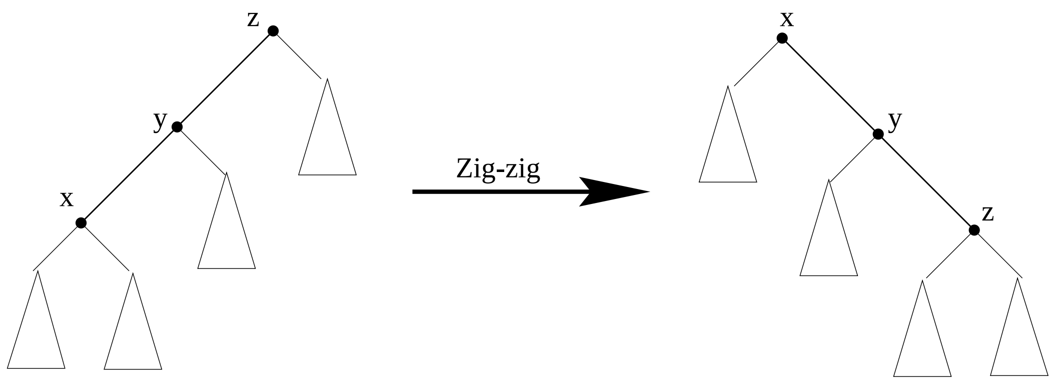
(d) Instead of holding the most frequently accessed items, we hold the most recently accessed item. We can replace the search tree on access frequencies by a doubly linked list holding the items in LRU order. The proof that working set theorem is satisfied is similar to lemma 5.

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. **(a)** For each balanced triple, the number of descendants of x is less than the number of descendants of z by at least a constant (0.9). We know that the root has n descendants, so if a path has k balanced nodes, the final node in the path has at most $(0.9)^k n \geq 1$ descendants, so that $k \leq \log_{1/0.9} n = O(\log n)$.

(b) Consider the biased triple below, and its form after a ZIG-ZIG rotation:

```
          z                x
         / \              / \
        y   D            A   y
       / \        =>        / \
      x   C                B   z
     / \                      / \
    A   B                    C   D
```

Let a, b, c, d denote the sizes of subtrees A, B, C, D respectively and write $\Delta(u)$ for the change in the rank of a node u. Then

$$\begin{aligned}
\Delta(x) &= \log \frac{a+b+c+d+3}{a+b+1} \leq \log 1/0.9 \\
\Delta(y) &= \log \frac{b+c+d+2}{a+b+c+2} \\
\Delta(z) &= \log \frac{c+d+1}{a+b+c+d+3} < \log 0.1.
\end{aligned}$$

The bounds for $\Delta(x)$ and $\Delta(z)$ follow directly from the definition of a biased triple. In the worst case for $\Delta(y)$, $a = 0$ and $d < 0.1(a+b+c+d+3)$. Then $\Delta(y) < \log(1.1/0.9)$. The total change in potential is at most $\log[(1/0.9)(1.1/0.9)(0.1)] = \log(11/81)$, a negative constant.

For the ZIG-ZAG case, consider the triple below:

```
        z                  x
       / \                / \
      y   D              /   \
     / \        =>      y     z
    A   x              / \   / \
       / \            A   B C   D
      B   C
```

Using the same notation, we have

$$\begin{aligned}
\Delta(x) &= \log \frac{a+b+c+d+3}{b+c+1} \\
\Delta(y) &= \log \frac{a+b+1}{a+b+c+2} \\
\Delta(z) &= \log \frac{c+d+1}{a+b+c+d+3}
\end{aligned}$$

Therefore,

$$\Delta(x) + \Delta(y) + \Delta(z) = \log \frac{(a+b+1)(c+d+1)}{(b+c+1)(a+b+c+2)}$$

By the inequality of the means, $(a+b+1)(c+d+1) \leq (a+b+c+d+2)^2/4$. Using this, we can bound the change in potential:

$$\Delta(x) + \Delta(y) + \Delta(z) \leq \log \frac{(a+b+c+d+2)^2}{4(b+c+1)(a+b+c+2)} < \log(1.1^2/4).$$

Once again, this is a negative constant bound.

(c) Using the notation from the previous part, now consider a balanced triple undergoing a ZIG-ZIG rotation. The total change in potential is

$$\begin{aligned}\Delta(x) + \Delta(y) + \Delta(z) &= \log \frac{(b+c+d+2)(c+d+1)}{(a+b+1)(a+b+c+2)} \\ &\leq \log \frac{(a+b+c+d+3)^2}{(a+b+1)^2} = 2(r(z) - r(x)).\end{aligned}$$

Similarly, for a ZIG-ZAG rotation, the change is

$$\begin{aligned}\Delta(x) + \Delta(y) + \Delta(z) &= \log \frac{(a+b+1)(c+d+1)}{(b+c+1)(a+b+c+2)} \\ &\leq \log \frac{(a+b+c+d+3)^2}{(b+c+1)^2} = 2(r(z) - r(x)).\end{aligned}$$

(d) Adding up the costs of rotations along the search path, we find that each of the biased rotations pays for itself by causing a constant reduction in potential. For the balanced ones, we get a sum bounded by $\sum_{i=1}^{k} 2(r(z_i) - r(x_i))$, where x_{i+1} is an ancestor of z_i. But this implies that $r(x_{i+1}) \geq r(z_i)$, so that we can telescope the sum and get a looser bound of $2(r(\text{root}) - r(x))$, where x is the result of the search. We know that $r(\text{root}) = \log n$, so the amortized cost cost of the entire operation is $O(\log n)$.

Problem 5. We will modify the lazy multi-level bucketing scheme described in lecture. At each level of the structure, there used to be an array of all the non-empty buckets, and a summary structure. We replace the array by a binary heap, which takes up the same amount of space.

- **insert** formerly took $O(k)$ time because it involved searching for the right array and then inserting into it. Now, we take $O(k)$ time to find the right heap to insert into, and then spend an additional $O(\log \Delta)$ time inserting the item into the heap for a total of $O(k + \log \Delta)$ time.
- **delete-min** formerly took $O(\Delta)$ time, but we can now reduce this to $O(\log \Delta)$ because we can avoid scanning the list by using the heap's cheap find-min and delete-min operations. However, if the item that is deleted is the last one in the bucket, this might cause a cascading delete-min. However, each of those deletes until we reach the first non-singleton heap will only cost $O(1)$, so the total cost is $O(k + \log \Delta)$.

- **decrease-key** formerly cost $O(1)$; in this version, the cost will be $O(\log \Delta)$ (for the same reason as **insert**).

We still need to pick values of k, Δ satisfying $\Delta^k = C$. Set $k = \log \Delta = (\log C)/k$, so that $k = \log \Delta = \sqrt{\log C}$ to get the desired time bounds for all the operations.

Problem 6. This was an open problem. If you solved it, you should go publish a paper.

Problem 7. Forever.