Problem Set 2 Solutions

Problem 1.  **this needs to be fleshed out** We will use 3 arrays $A$, $B$, and $C$. $A$ will represent that actual array and hold the stored items in their stored positions. Arrays $B$ and $C$ will act as “checks” for which entries of $A$ are truly initialized. $B$ will contain a list, built up sequentially from position 0, of all entries of $A$ that have been filled. We use a “length” variable $i$ to keep track of the number of entries in $B$. Each time we fill a new position $k$ in $A$, we store the value $k$ at $B[i]$, then increment $i$. Array $C$ will “index” Array 2. When we store $k$ at $B[i]$, we store $i$ at $C[k]$. Now, when we want to check whether position $k$ is full, we look at $C[k]$ to find out where in $B$ the value $k$ should be stored. If $C[k] ≥ i$ then it must be garbage meaning $k$ is empty. If $C[k] < i$ then we check if $B[C[k]] = k$, in which case we have proof that position $k$ is full; if $B[C[k]]$ has any other value then $C[k]$ is garbage meaning position $k$ is empty.

Problem 2.  (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 $≥ n/2$ 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]  /
 .    \ \
```

```
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.

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 \( \geq 4 \) 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

---

1Zeyuan: this picture is actually not fully solving this problem, as one needs to prove that it is strongly-connected.
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 3.**

(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:

\[
\begin{array}{c}
z \\
/ \\
y & D & A & y \\
/ \\
x & C & B & z \\
\end{array} \\
\rightarrow \\
\begin{array}{c}
z \\
/
\end{array}
\]
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

\[
\Delta(x) = \log \frac{a + b + c + d + 3}{a + b + 1} \leq \log 1/0.9
\]

\[
\Delta(y) = \log \frac{b + c + d + 2}{a + b + c + 2}
\]

\[
\Delta(z) = \log \frac{c + d + 1}{a + b + c + d + 3} < \log 0.1.
\]

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:

\[
/ \ \ / \ \ / \ \ / \ \\
A \ B \ C \ D
\]

Using the same notation, we have

\[
\Delta(x) = \log \frac{a + b + c + d + 3}{b + c + 1}
\]

\[
\Delta(y) = \log \frac{a + b + 1}{a + b + c + 2}
\]

\[
\Delta(z) = \log \frac{c + d + 1}{a + b + c + d + 3}
\]

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

\[
\Delta(x) + \Delta(y) + \Delta(z) = \log \frac{(b + c + d + 2)(c + d + 1)}{(a + b + 1)(a + b + c + 2)} \\
\leq \log \frac{(a + b + c + d + 3)^2}{(a + b + 1)^2} = 2(r(z) - r(x)).
\]

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

\[
\Delta(x) + \Delta(y) + \Delta(z) = \log \frac{(a + b + 1)(c + d + 1)}{(b + c + 1)(a + b + c + 2)} \\
\leq \log \frac{(a + b + c + d + 3)^2}{(b + c + 1)^2} = 2(r(z) - r(x)).
\]

(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 \(2(r(root) - r(x))\), where \(x\) is the result of the search. We know that \(r(root) = \log n\), so the amortized cost cost of the entire operation is \(O(\log n)\).

Problem 4. 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 \(k, \Delta\) 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 5. We augment the vEB queue to also hold a maximum element. We implement the desired operations as follows:

- **find**(x,Q): We check if x is either the minimum or maximum of the current queue. If so, we return it. Otherwise, make a recursive call and find low(x) in the subqueue Q[high(x)].

- **predecessor**(x,Q): If x is less than the minimum of Q, return null. If x is greater than the maximum of Q, return the maximum of Q. Otherwise, we make a recursive call to find the predecessor of low(x) in the subqueue Q[high(x)]. If the result of this recursive call is non-null, then we return the result. Otherwise, we make a call to find the predecessor of high(x) in Q.summary. The result of this call tells us the subqueue that is non-empty among the subqueues. In particular, if it is non-null, then we return the maximum element from that subqueue. However, if the result of the call was null, then we can return the minimum of Q.

- **successor**(x,Q): The algorithm is very similar to predecessor.

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

Problem 7. There is no available solution for this problem at this time.

Problem 8. Forever.