Problem Set 4.5 Solutions

Problem 4.5-1. Assessing the damage

The idea here is that we can augment a balanced binary search tree on the distances in order to keep track of the damage.

The augmentation is as follows: in each node we’ll add a field \( \Delta damage \) that will store the difference of a node’s total damage from that of its parent. For the root, \( \Delta damage \) will store the total damage to that node. The point is that we can update entire subtrees only by updating \( \Delta damage \) of the root, and can easily find the damage of a specific property by adding up the \( \Delta damage \) values on the search path. I’ll specify the operations with an extra argument, the current node; the operations the user actually calls should of course implicitly start at the root.

\[
\text{DECREASE-VALUE}(node, distance, amount) \\
\quad \text{if (node.distance} \leq \text{distance)} \\
\quad \quad \text{add amount to node.}\Delta damage \\
\quad \text{if a right subtree exists} \\
\quad \quad \text{subtract amount from node.right.}\Delta damage \\
\quad \text{DECREASE-VALUE(node.right, distance, amount)} \\
\quad \text{else DECREASE-VALUE(node.left, distance, amount)}
\]

\[
\text{ASSESS-DAMAGE}(node, distance) \\
\quad \text{if (node.distance} = \text{distance)} \text{ return node.}\Delta damage \\
\quad \text{else if (node.distance} < \text{distance)} \\
\quad \quad \text{return node.}\Delta damage + \text{ASSESS-DAMAGE(node.right, distance)} \\
\quad \text{else} \\
\quad \quad \text{return node.}\Delta damage + \text{ASSESS-DAMAGE(node.left, distance)}
\]

It is difficult to give a concise proof of correctness, and in any case the proof would only formalize the intuition given above for why the method should work, so I’ll skip it.

It is clear that each operation at most traverses a path from the root to a leaf, doing \( O(1) \) work at each node, so the running time of each operation is \( O(\log n) \).

Problem 4.5-2. Random TREAPS

(a) TREAP-insert can be accomplished as follows:

1. do a normal tree-insert of the new item
2. while the new item has higher score than its parent
   • rotate the new item and its parent
To show that this works, we can show that the following invariant is maintained in
the loop: the only possible violation of the rules of a TREA P is that the new item
has a bigger score than some of its ancestors.
It is clear that we don’t need to worry about the binary search tree order on the keys,
because we put the item in the right place in step 1, and then only use rotations,
which preserve search tree order. So we’ll just concern ourselves with the heap order.
At the beginning of the loop, the heap order can only be violated as the invariant
says because a normal tree-insert puts the new item in as a leaf.
Assuming that the invariant holds after $i$ iterations, let’s consider what happens
after the $i+1$st. The new item displaces its parent, and some of its children become
children of the (now former) parent. But since there were no heap violations before,
except for the new item being bigger than some ancestors, there are no new heap
violations after the rotation. So the invariant holds by induction.
Now when the loop terminates, the new item is less than its parent, and since the
parent must be less than its ancestors, the new item must be less than all of its
ancestors. Thus when the loop terminates we have a proper TREA P.

(b) We can show uniqueness by induction on the number of nodes.
Inductive hypothesis: a TREA P on $\leq n$ distinct keys is unique
Base case: $n = 0$, hypothesis is trivially true
Inductive step: Assume it is true for $n$. Consider $n + 1$ keys. In order to maintain
the heap property, the key with the largest score must be the root. And to be a
TREA P, the left subtree must be a TREA P on all keys less than the root, and the
right subtree must be a TREA P on all keys greater than the root. But each of
the left and right subtrees have $\leq n$ keys, so by induction, the TREA P on them is
unique. Thus the TREA P on all $n + 1$ keys is unique.

(c) By (b), we get the same tree structure, regardless of insertion order. So insertion
order is entirely irrelevant. Now to argue the expected height, we’d like to relate
the structure of a TREA P to that of a randomly built binary search tree. Suppose
we sorted the keys in decreasing order by score, and inserted them in order into
a normal binary search tree with the standard insert procedure. It is not hard to
see that we’d get the exact same structure as the TREA P. But since the scores are
random, sorting by score gives a random permutation. We know from recitation and
the book that a normal binary search tree built by inserting the keys in random order
has expected height $O(\log n)$. Therefore a TREA P has expected height $O(\log n)$.

Problem 4.5-3. Stacking the Deque

(a) To implement a min-stack, we use a singly-linked list, and keep for every items on
the stack the minimum of the items below it.
The stack will just be a head pointer, top. Each item is assumed to have fields min
and next. (If they didn’t, we could just put each item in a structure that did.) The
operations are as follows:
push(x): If the stack is empty, set x.min=x and top=x. Otherwise, set x.min =
min(x, top.min) and add x to the top of the stack (x.next = top; top = x).

pop(): Remove the top item (x = top; top = top.next) and return it.

find-min() Return top.min.

This is correct because of the LIFO nature of a stack. It is easy to argue by induction
that at all times each item’s min field correctly points to the minimum item below
it. Base case: the first item points to itself. Inductive step: on a push operation, we
set the new item’s min field to the min of it and the previous top. By the inductive
hypothesis the min of the previous top is set correctly, so the new min will be set
correctly. It is put above all other items, so it can have no effect on their min fields.
Likewise, pop removes an item that is above all other items in the stack, so every
remaining item is unaffected. Find-min does not change the stack at all.

All of the operations only look at/modify $O(1)$ pointers, so each operation takes
$O(1)$ time in the worst case.

(b) We implement a min-deque by using two min-stacks, A and B. The idea is that A
will be used for push and pop, and B will be used for inject and eject. This is fine,
as long as we don’t have A empty when we do a pop, or B empty when we do an
eject. If this bad situation arises, we will take the nonempty side and split it evenly
between A and B. We won’t have to split very often, so we’ll be able to show that
each operation takes $O(1)$ amortized time.

The operations are as follows:

push(x) push(x, A)
pop() If A is empty, pop half of the items off of B and push them onto a temporary
stack. Pop the remaining items off of B and push them onto A. Pop the items
off of the temporary stack and push them onto B. Now pop(A).

inject(x) push(x, B)
eject() If B is empty, pop half of the items off of A and push them onto a temporary
stack. Pop the remaining items off of A and push them onto B. Pop the items
off of the temporary stack and push them onto A. Now pop(B).

find-min() return min(find-min(A), find-min(B))

Correctness is even easier this time. Nothing can go wrong with a push or inject,
and we have handled the bad cases of pop and eject. By the correctness of the
min-stacks, find-min will do the right thing.

It remains to analyze the running time. Push, inject and find-min take constant
time, but pop may cost $3\text{size}(B)/2$ and eject may cost $3\text{size}(A)/2$. Luckily, this
doesn’t happen very often, so we can amortize the expensive operations against the
cheap ones.

Let $\Phi = 3|\text{size}(A) - \text{size}(B)|/2$. We define the amortized cost of each operation to
be the actual cost plus $\Delta \Phi$. Assuming we start with an empty deque, $\Phi$ starts at
zero, and it is clear by definition that it is always positive, so we need only check what happens for each operation:

<table>
<thead>
<tr>
<th>operation</th>
<th>actual cost</th>
<th>ΔΦ</th>
<th>amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>1</td>
<td>±1</td>
<td>0 or 2 = O(1)</td>
</tr>
<tr>
<td>inject</td>
<td>1</td>
<td>±1</td>
<td>0 or 2 = O(1)</td>
</tr>
<tr>
<td>find-min</td>
<td>2</td>
<td>0</td>
<td>2 = O(1)</td>
</tr>
<tr>
<td>pop (A empty)</td>
<td>1 + 3size(B)/2</td>
<td>≤ 1 - 3size(B)/2</td>
<td>≤ 2 = O(1)</td>
</tr>
<tr>
<td>pop (A not empty)</td>
<td>1</td>
<td>±1</td>
<td>0 or 2 = O(1)</td>
</tr>
<tr>
<td>eject (B empty)</td>
<td>1 + 3size(A)/2</td>
<td>≤ 1 - 3size(A)/2</td>
<td>≤ 2 = O(1)</td>
</tr>
<tr>
<td>eject (B not empty)</td>
<td>1</td>
<td>±1</td>
<td>0 or 2 = O(1)</td>
</tr>
</tbody>
</table>

**Problem 4.5-4. Quick Array Inserts**

(a) Each of the first log n − 1 calls to INSERT take O(1) time because they are just adding an element to the front of a list. The next call takes O(1) time to add an element to the list, O(m log m) = O(log n log log n) time to sort the list, and then O(n) time to merge the two lists, for a total time of O(n). (Note that it takes time O(n) to merge two arrays, one of length n and one of length m, n > m. This is because appending the end of one array to another requires copying the contents, unlike appending linked lists.) Thus the amortized running time of these calls to INSERT is \( \frac{(\log n - 1)O(1) + O(n)}{\log n} = O(\frac{n}{\log n}) \).

(b) The binary search takes time O(log n), and in the worst case of searching the entire almost full unsorted list afterwards, another O(m) = O(log n) time is spent, for a total of O(log n).

**Problem 4.5-5. Deletion in B-trees**

Deleting a key from a B-tree can be implemented as follows.

(a) If the key k is found in an internal node, we modify the B-tree so that k is transferred to a leaf. We do that by finding the immediate predecessor p of k and exchanging k with p. The immediate predecessor of k can be found by taking the largest key (rightmost leaf) in the subtree of keys smaller than k. Alternatively, we can exchange k with its immediate successor.

(b) If k is in the root T of the B-tree, there are three cases.

- If the root has some additional keys besides k, then k can simply be removed from the root, leaving the root with at least one other key.
- If k is the only key in the root and the root has a child, then the root is removed and its child becomes the new root.
- If k is the only key in the root and the root does not have children, then k is deleted from the root, and the B-tree’s root becomes empty.
(c) If $k$ is in a node $x$ with at least $t$ keys, we can remove $k$ from $x$ without violating the B-tree definition. Removing key $k$ leaves node $x$ with $t - 1$ keys, which is the minimal number of keys allowable in a B-tree node.

(d) If $k$ is in a node $x$ that has exactly $t - 1$ keys and $x$'s left sibling $y$ has $t$ or more keys, then we perform a rotation and remove $k$. Let $z$ be the common parent of $x$ and $y$, and let $p$ be the key in $z$ between $x$ and $y$. All the keys in $y$ are smaller than $p$, and all the keys in $x$ are larger than $p$. We perform the following right rotation around $p$; the largest key in node $y$ replaces $p$ in node $z$, and $p$ becomes the smallest key in node $x$. Now, $y$ has at least $t - 1$ keys, the number of keys in $z$ is unchanged, and $x$ has exactly $t$ keys. This configuration now enables us to remove $k$ from $x$. A similar left rotation can be performed if $x$ has a right sibling with $t$ keys or more.

(e) If $k$ is in a node $x$ that has exactly $t - 1$ keys, and $x$'s left and right siblings have no more than $t - 1$ keys each, then a merge operation and a recursive deletion is required. Let $y$ be the left sibling of $x$, let $z$ be their common parent, and let $p$ be the key in $z$ between $x$ and $y$. We remove $k$ from $x$ and merge the keys of $y$, the key $p$, and the rest of the keys of $x$, in that order, into one node with $2t - 2$ keys. We then recursively delete key $p$ from $z$. A similar operation can be performed with $x$'s right sibling.