Sample Solutions

CSC 263H

June 9, 2016

This document is a guide for what I would consider a good solution to problems in this course. You may also use the TeX file to build your solutions.

1. Consider the following problem we have seen in class:
   We have a set of unique integers \( X \) where \( |X| = n \).
   We need to support the following operations:
   \( \text{Insert}(x, X) \): Add element \( x \) to \( X \).
   \( \text{Sum}(X) \): Return the sum of all elements in \( X \).
   Insert should run in \( \mathcal{O}(\log(n)) \) time while Sum should take \( \mathcal{O}(1) \) time.
   \textbf{Note} there is an obvious solution to this where you just keep track of the sum on the fly, but that’s not very interesting.
Solution:

We will augment an AVL tree to solve this problem. We will simply map each element of \( X \) to a node in an AVL tree using the value of \( x \) as the key in the tree. We will assume that we have a pointer to the root node of the tree, we call this \( X.root \).

As we saw in class an AVL tree is a BST that in addition to the key value stores the balance factor at each node. We further augment this by storing at each node \( i \) the sum of all values under it at that node (that is the sum of all values contained in the subtree rooted at node \( i \)).

It is easy to see if each node contains this augmented value we can efficiently calculate \( i.sum \) as \( i.key + i.LeftChild.sum + i.RightChild.sum \). For nodes without a left (right) child we simply assume \( i.LeftChild.sum \) (\( i.RightChild.sum \)) is zero. This means for leaf nodes \( i.sum = i.key \).

Now we describe how to implement each of the above operations.

**Algorithm 1** Sum(\( X \))

1: return \( X.root.sum \)

It is easy to see this takes constant time since we simply return a value stored at the root node of the tree. It is also easy to see this is correct since \( X.root.sum \) contains the sum of all elements in the subtree rooted at it, that is the entire set \( X \).

**Algorithm 2** Insert(\( x \))

1: Do an AVLInsert(\( x, X \)) with the modifications described below.

For the first state of the modified AVLInsert do exactly as a normal AVLInsert does, that is work your way to a free leaf node and insert \( x \). Afterwards when we set \( x.BalanceFactor = 0 \) we also set \( x.sum = x.key \), this is the correct thing to do since the subtree rooted at \( x \) contains only \( x \). Now as we work our way up the tree updating the Balance Factor of each node as AVLInsert does we also update the Sum values of each node. As stated we can easily calculate \( i.sum \) as \( i.key + i.LeftChild.sum + i.RightChild.sum \), thus we only add a constant amount of work to each step so far. When we do a rotation since some nodes have had the sub tree under them change (only a constant number as we learned in class / the notes) we again update the Sum value of those nodes along with their Balance Factors. Again this is only a constant amount of work extra.

The one major difference is we can’t stop the traversal towards the root after a rotation or a sub tree becomes balanced. While we know the Balance Factor values will not change (and thus no more rotations are necessary) the Sum values of ancestors of the inserted node must still be updated since the subtree rooted at them has a different sum after insertion. Thus we need to continue traversing up towards the root. But the tree is of height \( O(log(n)) \) since AVL trees are balanced. Thus since we only do a constant amount of extra work at each node over regular AVLInsert our running time is still \( O(log(n)) \).
2. Consider the following problem we have seen in class:

We have an array of $n$ integers $[x_1, x_2, ..., x_n]$.

We want to determine if any element is duplicated in this array.

Show how to do this in $O(n)$ expected time subject to certain probabilistic assumptions we have discussed in class.

**Note:** When I presented our initial solution (the first one seen in class) I assumed that no integer would appear more than two times, this is an assumption I forgot to state initially.
**Solution 1:**

For the first solution we will assume that no integer appears more than twice in the array.

We will solve this problem using hashing.

First to do hashing we need to define our set of keys \( K \). We will let \( K = \mathbb{Z} \) the set of all integers.

Next we need to determine how large our hash table \( T \) is. That is how many buckets \( m \) are in \( T \). As usual we will pick \( m \) to be a constant multiple of \( n \), say something like \( m = \frac{n^2}{25} \).

Finally we need to assume our hash function \( h() \) satisfies SUHA. That is \( h \) spreads the keys \( K \) uniformly about \( 0, ..., m \) in expectation.

Now we can present our solution:

---

**Algorithm 3 HashSolutionOne**

1: for \( x_i \in [x_1, x_2, ..., x_n] \) do
2: Insert element \( x_i \) into bucket \( T[h(x_i)] \) at the beginning of the chain.
3: end for
4: for bucket \( \in T \) do
5: Look at each pair \((a, b)\) of items in the chain at bucket, if \( a == b \) print \( a \)
6: end for

---

To see the algorithm is correct note that if we truly had a pair of matching integers \((a, b)\) then they must hash to the same bucket in the first for loop and we will end up comparing them in the second for loop.

To see our running time in the first for loop we simply iterate over each element and insert it into the hash table. Calculating the hash function can be done in \( \mathcal{O}(1) \) time and since we insert at the beginning of a chain we only require \( \mathcal{O}(1) \) work again. Thus in total the first for loop takes \( \mathcal{O}(n) \) work.

The second for loop simply iterates over each pair of elements in each bucket. Since we assumed SUHA we expect that \( \frac{n}{m} = 25 \) keys will be assigned to any given bucket under \( h() \). But since a key appears at most twice in \([x_1, x_2, ..., x_n]\) we get the length of a chain should be no longer than 50 elements in expectation. Thus we must make \( 50^2 \in \mathcal{O}(1) \) comparisons in expectation for the second for loop per bucket. Since there are \( \frac{n}{m} \) buckets we do \( \mathcal{O}(1) \cdot \frac{n}{m} \) work (in expectation) in the second for loop, this is just \( \mathcal{O}(n) \). Thus in expectation we only do \( \mathcal{O}(n) \) work for the whole procedure (in expectation).
**Solution 2:**

All is the same as solution one but now we don’t have the assumption of having an element duplicated at most once. We will skip right to the procedure. Also for the proof we will borrow many of the derived facts (like $\frac{n}{m} \in \mathcal{O}(1)$) from above so we don’t restate them.

<table>
<thead>
<tr>
<th>Algorithm 4 HashSolutionTwo</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: for $x_i \in [x_1, x_2, \ldots, x_n]$ do</td>
</tr>
<tr>
<td>2: Probe $T[h(x_i)]$ for element $x_i$</td>
</tr>
<tr>
<td>3: if The probe was successful (found $x_i$ in the chain) then</td>
</tr>
<tr>
<td>4: print $x_i$</td>
</tr>
<tr>
<td>5: else</td>
</tr>
<tr>
<td>6: Insert $x_i$ into the start of the chain found at $T[h(x_i)]$</td>
</tr>
<tr>
<td>7: end if</td>
</tr>
<tr>
<td>8: end for</td>
</tr>
</tbody>
</table>

To see this algorithm is correct again we note if we truly had a pair of matching integers $(a, b)$ then they must hash to the same bucket. If this were the case we would insert the first we saw of the pair into the table, and when we saw the second (or the third / fourth ...) we would find it during the probe of the chain.

To see the running time is correct we again note that calculating the hash function can be done in $\mathcal{O}(1)$ time. To probe the chain requires work linear in the length of the chain. Since we assumed SUHA we get that the expected number of keys that hash to a given bucket under $h()$ is $\mathcal{O}(1)$. Since we only insert a key into a chain one time we get that each chain should be no longer than $\mathcal{O}(1)$ in expectation. Thus probing a chain should only take $\mathcal{O}(1)$ time in expectation, and insertion can still be done in $\mathcal{O}(1)$ since we are inserting into the beginning of a list. Thus each element requires a total of $\mathcal{O}(1)$ work in expectation so our expected running time is now $\mathcal{O}(n)$. 

5