CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 9

For: March 13–March 17

Problem 1. Hashing Basics

Problem 1.a.

Try hashing these items $[42, 24, 18, 36, 52, 0, 47, 45, 60, 27, 32, 7]$ with the following hash function $h(x) = x \mod 7$. Each row in the table corresponds to the bucket of $h(x)$. Fill in the table below with your answer!

Assume that we are using **chaining** to handle hash collisions!

h(x)	x_1	x_2	x_3	x_4
0				
1				
2				
3				
4				
5				
6				

Solution: First, we would work out what are the hashed values of the above items. They are $[0, 3, 4, 1, 3, 0, 5, 3, 4, 6, 4, 0]$.

Next, we can iterate through these hashed values one-by-one and fill up our table!

h(x)	x_1	x_2	x_3	x_4
0	42	0	7	
1	36			
2				
3	24	52	45	
4	18	60	32	
5	47			
6	27			

Problem 1.b.

We typically use Linked Lists to store the items in a bucket! But... what if instead of a Linked List, we use an AVL Tree to store the items in a bucket?

What are the advantages or disadvantages of such a solution?

Solution:

- Time Complexity
  - Consider the theoretical worst case for searching a Hash Table of size n, where every element is hashed into the same bucket. This bucket's container would then be of size n. Then, the time complexity for searching for an element would be heavily dependent on the type of container chosen.
  - Searching through a Linked List of size n is $O(n)$.
  - Searching through an AVL Tree of size n is $O(\log n)$.
  - Thus, the time complexity when using an AVL Tree would be better in the worst case!
- Overhead
  - Each node in an AVL Tree has to store more information than a Linked List node.
  - An AVL Tree node has to store the balance factor and two child pointers.
  - Meanwhile, a Linked List node only has to store a single pointer to the next node.
- Complexity of Algorithm
  - Additionally, AVL Trees are more complex than Linked Lists!

In summary, using an AVL Tree would improve the **worst case** scenario, but at the cost of more overhead and complexity!

However, if the hash function chosen is good enough (under uniform hashing assumption) and the number of buckets is larger than the number of elements, the expected time would be identical in both cases. This worst case scenario only occurs when the hash function is not good.

Bonus information: Java's implementation of HashMap converts the buckets from a Linked List to Red Black Trees (a type of self-balancing Binary Search Tree) after a certain threshold of number of buckets and bucket size. This switching to Red Black Trees is a fallback that is expected to be rare if the hash function chosen is good enough.

Problem 1.c.

The goal of Hash Tables are to store (key, value) pairs. Here's a question, at each bucket, is storing just the **(value)** sufficient? Or do we need to store the entire **(key, value)** pair? Why do you think so?

For example, for a (key, value) pair of (17, 200). At the bucket $h(17)$, is storing (200) sufficient, or do you need to store (17, 200)?

Solution: Storing just the (value) is not sufficient! Instead, the entire (key, value) pair needs to be stored. Consider two pairs (x_1, y_1), (x_2, y_2) and let $h(x_1) = h(x_2)$. We would only store the values y_1 and y_2 at the bucket $h(x_1)$.

When we try to search for the value of x_1 in the hash table above, we would encounter the bucket $h(x_1)$, and at this bucket, there would be two values y_1 and y_2. However, there would be no way to tell if y_1 or y_2 belonged to x_1!

If we had stored the entire (key, value) pair, it would be simple to tell that y_1 was the correct value!

Problem 2. The Missing Element

Let's revisit the same old problem that we've discussed at the beginning of the semester, finding missing items in the array. Given n items in no particular order, but this time possibly with duplicates, find the first missing number (if we were to start counting from 1), or output "all present" if all values 1 to n were present in the input.

For example, given $[8, 5, 3, 3, 2, 1, 5, 4, 2, 3, 3, 2, 1, 9]$, the first missing number here is 6.

Bonus: (no need for hash functions): Can we do the same thing using $O(1)$ space? i.e. in-place.

Solution: The idea here is to use a hash set to store all the values that we've seen in the array and ignore duplicates. Then, we'll start going through the hash set to check if each number was present, starting from 1. This would take $O(n)$ time in expectation given the right assumption for hash functions.

Note that we could actually just use an array of size n instead to accomplish the same goal (by naively using element value as the array index)! Sometimes it's important to realise when a simple array would suffice and using something more complex such as a hash set can be unnecessary.

Solution to Bonus: The solution to the bonus question does not use hash sets, but it has a similar idea to cuckoo hashing. The idea is that we will consider every item once, and given its value, we will try to place it in its own index by swapping it in-place.

For example, based on the input given above, we would try to place item 8 at index 8, but there's an item of value 4 there, so then item 8 "kicks out" the 4 from that place. Now we try to place 4 in its index, but there's an item of value 3 there. So we repeat the same process with 3 before stopping because we realise that 3 is already sitting at index 3. If there is a value larger than n, we can simply ignore that value since we know that it will not be part of consideration.

After we do this for all elements, we run through the array one more time and check if at index i the value i is stored. Then, output the first index such that it fails.

The overall time complexity of this solution is $O(n)$. One way to reason about this is that an element will not be moved once it is in the correct position. So, when a swap happens, at least one element would be moved to the correct position, and not swapped thereafter. Hence, the total number of swaps would be at most n. Finally, we add in the cost of iterating through the array (which involves only constant time operations per element). Therefore, we get $O(n)$.

Problem 3. Data Structure 2.0

Implement a data structure RandomizedSet with the following operations:

1. RandomizedSet() which initializes the data structure.
2. Insert(val) which inserts an item *val* into the set if not present.
3. Remove(val) which removes the item *val* from the set if present.
4. GetRandom() which returns a random element from the current set of elements. Every element must have an **equal probability** of being returned.

All these operations must work in expected $O(1)$ time! Hint: a Hash Table might come in handy!

Assume that the maximum number of elements present in the RandomizedSet will never exceed a reasonable number n.

Solution: We will use a combination of an Array and a Hash Table. The Array is used to store all our items. The Hash Table is used to check if an item is present in the Array efficiently, **and** also to check the position of an item in the Array efficiently.

On initialization of RandomizedSet, we will initialize an Array A with a size of n and we will initialize an empty Hash Table T. Additionally, we will keep track of the current size of A with a variable $size$ initialized to 0.

On Insert(val), if *val* is already in T, we do nothing. Otherwise, we will insert *val* into the back of A, i.e. $A[size] = val$. Then, increment $size$ by one. We will also insert a (key, value) pair of (val, $size$) into T. This (key, value) pair is used to keep track of the location of *val* in A.

On Remove(val), if *val* is not in T, we do nothing. Otherwise, we will need to remove *val* from A and T. First, we check the location of *val* in A with $T[val]$. If *val* is not positioned at the *last* index of A, we swap it with the element at the last index of A, and update the corresponding positions in T. Finally, we decrement $size$ by one (effectively deleting the last element) and remove *val* from T.

On GetRandom(), we simply need to randomly choose an index from $[0, size - 1]$, both inclusive. Then, return the value at that index in A.

Problem 4. Data Structure 3.0

Let's try to improve upon the kind of data structures we've been using so far a little. Implement a data structure with the following operations:

1. Insert in $O(\log n)$ time
2. Delete in $O(\log n)$ time
3. Lookup in $O(1)$ time
4. Find successor and predecessor in $O(1)$ time

Solution: The gist of the solution is to do two things:

1. Use both a height balanced BST (such as an AVL tree) and a hash table. The hash table here will map each key to the node that it corresponds to.
2. Modify the BST: for each node of the BST, we also store a pointer to its predecessor and successor (let's call the pointers "previous" and "next").

- For insertions and deletions: perform the usual insertion and deletion procedure on the BST, and add/remove the corresponding entry in the hash table, which will take $O(\log n)$ time.

  However, there is also the added work of having to maintain the two additional pointers. After we insert, we run the usual successor and predecessor algorithm, which should take $O(\log n)$ time. After those are found, we need to: (1) set the predecessor's "next" to the newly inserted node, (2) set the newly inserted node's "previous" to the predecessor, and the "next" to the successor and (3) set the "previous" of the successor to the newly inserted node. Deletion works off the same idea.

- For lookup queries: use the hash table to find the corresponding node in O(1) time.
- For successor and predecessor queries: first, perform a lookup query to obtain the node in the BST, and then use either the "next" or the "previous" pointer to get the corresponding successor/predecessor node that we need.

Problem 5. Coupon Chaos!

Mr. Nodle has some coupons that he wishes to spend at his favourite cafe on campus, but there are different types of coupons. In particular, there are t distinct coupon types, and he can have any number of each type (including 0). He has n coupons in total.

He wishes to use one coupon a day, starting from day 1. He wishes to use his coupons in ascending order and will use up all his coupons that are of a lower type first before moving on to the next type. Nodle wishes to build a calendar that will state which coupon he will be using.

- The list of coupons will be given in an array. An example of a possible input is: $[5, 20, 5, 20, 3, 20, 3, 20]$. Here, $t = 3$, and $n = 8$. The output here would be $[3, 3, 5, 5, 20, 20, 20, 20]$.
- Since the menu at the cafe that he frequents is not very diverse, there aren't many different types of coupons. So we'll say that t is much smaller than n.

Give as efficient an algorithm as you can, to build his calendar for him.

Solution: There are a few possible solutions to this problem, but it boils down to sorting an array with many duplicates.

- One possible way is to modify a height balanced BST (such as an AVL tree) to also store the counts of each item. Using this, insert all the items in the input, then do an in-order traversal, outputting the value at the current node for as many times as it was inserted into the balanced binary search tree. This takes $O(n \log t)$ time, since the size of the tree is at most t and we need to do n insertion operations.
- Another solution that works better on average, given reasonable hashing assumptions is to hash the coupon types, and keep track of how much of each type we've seen. Additionally, for every new type of coupon that we hash into the table, we add it into a list. In expectation this should take at most $O(n)$ time.

  After processing the input, we'll sort the list of keys that we had obtained previously, which will take $O(t \log t)$ time.

  Then in the sorted order, we look up the counts for each key and add the key to the calendar as many times as it was counted based on the hash table. Since there were a total of n items, this should take at most $O(n)$ time.

  In total, this solution takes $O(n + t \log t)$ time, which is equivalent to $O(n)$ since $t << n$.
- Finally, we can also consider quicksort with 3 way partitioning! Among the other comparison based sorting algorithms, this is optimal since it is best able to handle the situation where we have many duplicates and $t << n$. After each partitioning step, if the pivot has multiple duplicates, all the duplicates would already be in the correct index and we are left with two significantly smaller subarrays to recurse on since the duplicates would not be included in the recursive calls to quicksort. The sort will use a maximum of t pivots, hence the expected runtime is given by $O(n \log t)$.