Section 04: Solutions

1. Hash table insertion

For each problem, insert the given elements into the described hash table. Do not worry about resizing the internal array.

(a) Suppose we have a hash table that uses separate chaining and has an internal capacity of 12. Assume that each bucket is a linked list where new elements are added to the front of the list.

Insert the following elements in the EXACT order given using the hash function $h(x) = 4x$:

0, 4, 7, 1, 2, 3, 6, 11, 16

Solution:

To make the problem easier for ourselves, we first start by computing the hash values and initial indexes:

key	hash	index (pre probing)
0	0	0
4	16	4
7	28	4
1	4	4
2	8	8
3	12	0
6	24	0
11	44	8
16	64	4

The state of the internal array will be

$6 \to 3 \to 0$	/	/	/	$16 \to 1 \to 7 \to 4$	/	/	/	$11 \to 2$	/	/	/

(b) Suppose we have a hash table that uses linear probing and has an internal capacity of 13.

Insert the following elements in the EXACT order given using the hash function $h(x) = 3x$:

2, 4, 6, 7, 15, 13, 19

Solution:

Again, we start by forming the table:

key	hash	index (before probing)
2	6	6
4	12	12
6	18	5
7	21	8
15	45	6
13	39	0
19	57	5

Next, we insert each element into the internal array, one-by-one using linear probing to resolve collisions. The state of the internal array will be:

13	/	/	/	/	6	2	15	7	19	/	/	4

(c) Suppose we have a hash table that uses quadratic probing and has an internal capacity of 10.

Insert the following elements in the EXACT order given using the hash function $h(x) = x$:

0, 1, 2, 5, 15, 25, 35

Solution:

Here, the keys, hash, and initial index (before probing) are all the same, so there's no need to construct the table.

The state of the internal array will be:

0	2	1	/	35	5	15	/	/	25

2. Analyzing dictionaries

(a) What are the constraints on the data types you can store in an AVL tree? When is an AVL tree preferred over another dictionary implementation, such as a HashMap?

Solution:

AVL trees are similar to TreeMaps. They require that keys be orderable, though not necessarily hashable. The value type can be anything, just like any other dictionary. A perk over HashMaps is that with AVL trees, you can iterate over the keys in sorted order.

(b) When is using a BST preferred over an AVL tree?

Solution:

One of AVL's advantages over BST is that it has an asymptotically efficient find() even in the worst-case.

However, if you know that find() won't be called frequently on the BST, or if you know the keys you receive are sufficiently random enough that the BST will stay balanced, you may prefer a BST since it would be easier to implement. Since we also don't need to worry about performing rotations, using a BST could be a constant factor faster compared to using an AVL tree.

(c) Consider an AVL tree with n nodes and a height of h. Now, consider a single call to `get(...)`. What's the maximum possible number of nodes `get(...)` ends up visiting? The minimum possible?

Solution:

The max number is h; the min number is 1 (when the element we're looking for is just the root).

3. True or false?

(a) An insertion in an AVL with n nodes requires $\Theta(n)$ rotations.

Solution:

False. Each insertion will require either no rotations, a single rotation, or a double rotation. So, the total number of rotations is in $\Theta(1)$.

(b) A set of numbers are inserted into an empty BST in sorted order and inserted into an empty AVL tree in random order. Listing all elements in sorted order from the BST is $\mathcal{O}(n)$, while listing them in sorted order from the AVL tree is $\mathcal{O}(\log(n))$.

Solution:

False. An AVL tree traversal is also $\mathcal{O}(n)$, since we still have to look at every node once to traverse the tree.

(c) If items are inserted into an empty BST in sorted order, then the BST's `get()` is just as asymptotically efficient as an AVL tree whose elements were inserted in unsorted order.

Solution:

False. If items are inserted into a BST in sorted order, it produces a linked list.

In that case, `get()` would take $\mathcal{O}(n)$ time.

(d) An AVL tree will always do a maximum of two rotations in an insert.

Solution:

True. For an intuition on why this is true, notice that an insertion causes an imbalance because the problem node has one subtree of height k, and the other subtree had height $k+1$, and the insertion occured on the subtree with height $k+1$ to make it height $k+2$.

Then, our rotation will rebalance the tree rooted from the problem node's position such that each subtree is height $k+1$. But since the tree prior to insertion was balanced when the problem node had height $k+2$, and the problem node after rotation still has height $k+2$, and the problem node is also now balanced, the tree must now be completely balanced.

(We assume that rotations do not introduce more imbalances below them, since they are a method of fixing imbalances).

4. Big-$\mathcal{O}$

Write down the big-$\mathcal{O}$ for each of the following:

(a) Insert and find in a BST.

Solution:

$\mathcal{O}(n)$ and $\mathcal{O}(n)$, respectively.

(b) Insert and find in an AVL tree.

Solution:

$\mathcal{O}(\log(n))$ and $\mathcal{O}(\log(n))$, respectively.

(c) Finding the minimum value in an AVL tree containing n elements.

Solution:

$\mathcal{O}(\log(n))$. We can find the minimum value by starting from the root and constantly traveling down the left-most branch.

(d) Finding the k-th largest item in an AVL tree containing n elements.

Solution:

With a standard AVL tree implementation, it would take $\mathcal{O}(n)$ time. If we're located at a node, we have no idea how many elements are located on the left vs right and need to do a traversal to find out. We end up potentially needing to visit every node.

If we modify the AVL tree implementation so every node stored the number of children it had at all times (and updated that field every time we insert or delete), we could do this in $\mathcal{O}(\log(n))$ time by performing a binary search style algorithm.

(e) Listing elements of an AVL tree in sorted order

Solution:

$\mathcal{O}(n)$.

5. Design

Imagine a database containing information about all trains leaving the Washington Union station on Monday. Each train is assigned a departure time, a destination, and a unique 8-digit train ID number.

What data structures you would use to solve each of the following scenarios. Depending on scenario, you may need to either (a) use multiple data structures or (b) modify the implementation of some data structure.

Justify your choice.

(a) Suppose the schedule contains 200 trains with 52 destinations. You want to easily list out the trains by destination.

Solution:

One solution would be to use a dictionary where the keys are the destination, and the value is a list of corresponding chains. Any dictionary implementation would work.

Alternatively, we could modify a separate chaining hash table so it has a capacity of exactly 52 and does not perform resizing. If we then make sure each destination maps to a unique integer from 0 to 51 and hash each train based on this number, we could fill the table and simply print out the contents of each bucket.

A third solution would be to use a BST or AVL tree and have each train be compared first based on destination then based on their other attributes. Once we insert the trains into a tree, we can print out the trains sorted by destination by doing an in-order traversal.

(b) In the question above, trains were listed by destination. Now, trains with the same destination should further be sorted by departure time.

Solution:

Regardless of which solution we modify, we would first need to ensure that the train objects are compared first by destination, and second by departure time.

We can modify our first solution by having the dictionary use a sorted set for the value, instead of a list. (The sorted set would be implemented using a BST or an AVL tree).

We can modify our second solution in a similar way by using specifically a BST or an AVL tree as the bucket type.

Our third solution actually requires no modification: if the trains are now compared first by destination and second by departure time, the AVL and BST tree's iterator will naturally print out the trains in the desired order.

(c) A train station wants to create a digital kiosk. The kiosk should be able to efficiently and frequently complete look-ups by train ID number so visitors can purchase tickets or track the location of a train. The kiosk should also be able to list out all the train IDs in ascending order, for visitors who do not know their train ID.

Note that the database of trains is not updated often, so the removal and additions of new trains happen infrequently (aside from when first populating your chosen DS with trains).

Solution:

Here, we would use a dictionary mapping the train ID to the train object.

We would want to use either an AVL tree or a BST tree, since we can list out the trains in sorted order based on the ID.

Note that while the AVL tree theoretically is the better solution according to asymptotic analysis, since it's guaranteed to have a worst-case runtime of $\mathcal{O}(\log(n))$, a BST would, pragmatically speaking, work just as well since we only have 200 trains.

Even if the tree ends up being degenerate, searching through a tiny linked list of only 200 element is realistically going to be a fast operation.

To put it another way, since $n \approx 200$ in this specific set of scenarios, it's the case that $\mathcal{O}(\log(n)) = \mathcal{O}(\log(200)) = \mathcal{O}(n) = \mathcal{O}(200) = \mathcal{O}(1)$.

6. Evaluating hash functions

Consider the following scenarios.

(a) Suppose we have a hash table with an initial capacity of 12. We resize the hash table by doubling the capacity. Suppose we insert integer keys into this table using the hash function $h(x) = 4x$.

Why is this choice of hash function and initial capacity suboptimal? How can we fix it?

Solution:

Notice that the hash function will initially always cause the keys to be hashed to at most one of three spots: 12 is evenly divided by 4.

This means that the likelyhood of a key colliding with another one dramatically increases, decreasing performance.

This situation does not improve as we resize, since the hash function will continue to map to only a fourth of the available indices.

We can fix this by either picking a new hash function that's relatively prime to 12 (e.g. $h(x) = 5x$), by picking a different initial table capacity, or by resizing the table using a strategy other then doubling (such as picking the next prime that's roughly double the initial size).

(b) Suppose we have a hash table with an initial capacity of 8 using quadratic probing. We resize the hash table by doubling the capacity.

Suppose we insert the integer keys $2^{20}, 2 \cdot 2^{20}, 3 \cdot 2^{20}, 4 \cdot 2^{20}, \ldots$ using the hash function $h(x) = x$.

Describe what the runtime of the dictionary operations will over time as you keep adding these keys to the table.

Solution:

Initially, for the first few keys, the performance of the table will be fairly reasonable.

However, as we insert each key, they will keep colliding with each other: the keys will all initially mod to index 0.

This means that as we keep inserting, each key ends up colliding with every other previously inserted key, causing all of our dictionary operations to take $\mathcal{O}(n)$ time.

However, once we resize enough times, the capacity of our table will be larger then 2^{20}, which means that our keys no longer necessarily map to the same array index. The performance will suddenly improve at that cutoff point then.

7. Code analysis

For this problem, we will consider a hypothetical hash table that uses linear probing and implements the `IDictionary` interface. Specifically, we will focus on analyzing and testing one potential implementation of the `remove` method.

(a) Come up with at least 4 different test cases to test this `remove(...)` method. For each test case, describe what the expected outcome is (assuming the method is implemented correctly).

Try and construct test cases that check if the `remove(...)` method is correctly using the key's hash code. (You may assume that you can construct custom key objects that let you customize the behavior of the `equals(...)` and `hashCode()` method.)

Solution:

Some examples of test cases:

- If the dictionary contains null keys, or if we pass in a null key, everything should still work correctly.
- If we try removing a key that doesn't exist, the method should throw an exception.
- If we pass in a key with a large hash value, it should mod and stay within the array.
- If we pass in two different keys that happen to have the same hash value, the method should remove the correct key.
- If we pass in a key where we need to probe in the middle of a cluster, removing that item shouldn't disrupt lookup of anything else in that cluster.

  For example, suppose the table's capacity is 10 and we pass in the integer keys 5, 15, 6, 25, 36 in that order. These keys all collide with each other, forming a primary cluster. If we delete the key 15, we should still successfully be able to look up the values corresponding to the other keys.

(b) Now, consider the following (buggy) implementation of the `remove(...)` method. List all the bugs you can find.

```java
public class LinearProbingDictionary<K, V> implements IDictionary<K, V> {
    // Field invariants:
    //
    // 1. Empty, unused slots are null
    // 2. Slots that are actually being used contain an instance of a Pair object

    private Pair<K, V>[] array;

    // ...snip...

    public V remove(K key) {
        int index = key.hashCode();

        while ((this.array[index] != null) && !this.array[index].key.equals(key)) {
            index = (index + 1) % this.array.length;
        }

        if (this.array[index] == null) {
            throw new NoSuchKeyException
        }
        V returnValue = this.array[index].value;
        this.array[index] = null;
        return returnValue;
    }
    }
```

Solution:

The bugs:

- We don't mod the key's hash code at the start
- This implementation doesn't correctly handle null keys
- This implementation does not correctly handle the ”clustering” test case described up above.

  If we insert 5, 15, 6, 25, and 36 then try deleting 15, future lookups to 6, 25, and 36 will all fail.

Note: The first two bugs are, relatively speaking, trivial ones with easy fixes. The last bug is the most critical one and will require some thought to detect and fix.

(c) Briefly describe how you would fix these bug(s).

Solution:

- Mod the key's hash code with the array length at the start.
- Handle null keys in basically the same way we handled them in `ArrayDictionary`
- Ultimately, the problem with the ”clustering” bug stems from the fact that breaking a primary cluster into two in any way will inevitably end up disrupting future lookups.

  This means that simply setting the element we want to remove to null is not a viable solution. Here are many different ways we can try and fix this issue, but here are just a few different ideas with some analysis:

  – One potential idea is to ”shift” over all the elements on the right over one space to close the gap to try and keep the cluster together. However, this solution also fails.

    Consider an internal array of capacity 10. Now, suppose we try inserting keys with the hashcodes 5, 15, 7. If we remove 15 and shift the ”7” over, any future lookups to 7 will end up landing on a null node and fail.

  – Rather then trying to ”shift” the entire cluster over, what if we could instead just try and find a single key that could fill the gap. We can search through the cluster and try and find the very last key that, if rehashed, would end up occupying this new slot.

    If no key in the cluster would rehash to the now open slot, we can conclude it's ok to leave it null.

    This would potentially be expensive if the cluster is large, but avoids the issue with the previous solution.

  – Another common solution would be to use lazy deletion. Rather then trying to “fill” the hole, we instead modify each `Pair` object so it contains a third field named `isDeleted`.

    Now, rather then nulling that array entry, we just set that field to true and modify all of our other methods to ignore pairs that have this special flag set. When rehashing, we don't copy over these “ghost” pairs.

    This helps us keep our delete method relatively efficient, since all we need to do is to toggle a flag.

    However, this approach also does complicate the runtime analysis of our other methods (the load factor is no longer as straightforward, for example).

8. Algorithm design: easier

When writing mathematical expression, we typically write expressions in *infix* notation: in the form `NUM OPERATOR NUM`. An example of an expression written in infix notation is $4 + 6 * 5$. This expression evaluates to 34.

An alternative way we can write this expression is using *post-fix* notation: in the form `NUM NUM OPERATOR`. For example, consider the following expression written in post-fix notation:

$$4, 6, 5, *, +$$

This expression is interpreted in the following way:

- Read and store 4
- Read and store 6
- Read and store 5
- Multiply the last two stored values (and remove them from storage), then store the result
- Add the last two stored values (and remove them from storage), then store the result

The last result stored is the final "output". In this case, the expression above also evaluates to 34.

Explain how you might apply or adapt the ADTs and data structures you've studied so far to evaluate an expression written in post-fix notation. Assume you accept the expression you need to evaluate as a string.

Solution:

Split the string (e.g. by doing `input.split(", ")` in Java) to get an array containing each number or operator.

Next, create a stack of ints, and iterate through the array.

Every time we encounter a number (or rather, anything we don't recognize as being an operator), convert that string to a number and push it onto the stack.

Every time we encounter an operator, pop the last two values on the stack and perform that operation. (We would likely need to hard-code how to handle each operator in a large if/else if/else branch or something). Push the result back on to the stack.

Once we finish handling every element in the array, pop whatever value we have left and return that.

9. Algorithm design: harder

(a) Given a BST, describe how you could convert it into an AVL tree. What is the runtime of your algorithm?

Solution:

One solution would be to just traverse through the BST and insert each element into a different AVL tree. The traversal takes n time, each insert take $\log(n)$ time. So, the total runtime would be $\mathcal{O}(n \log(n))$.

If we want to be clever, we can do this in $\mathcal{O}(n)$ time by first traversing through the BST and storing each element into an array. Note that the elements are now stored in the array in sorted order.

We can then perform an algorithm roughly similar to the binary search algorithm, except that instead of picking whether we want to traverse down to the left half of the array or the right half, we traverse down both.

Once this algorithm hits the base case, it constructs and returns a new node object. We then recursively go back up, constructing a (now balanced) tree as we go. We end up with a balanced tree which, by definition, must be an AVL tree.

Building the array takes $\mathcal{O}(n)$ time; building the tree will also take $\mathcal{O}(n)$ time (since we visit each element only once, and do a constant amount of work per each array element).

(b) Give pseudocode for an algorithm that verifies that a tree satisfies all of the AVL invariants in $\mathcal{O}(n)$ time.

Assume every node object has five fields: `key`, `value`, `height`, `left`, and `right`.

Be sure to verify that:

- The tree is actually binary search tree
- The height information of every node is correct
- Every node is balanced

Hint: rather then trying to check all three of these things in a single pass, try writing three separate methods: one per each invariant. While it's possible to check everything in one pass, doing so will be more challenging to implement.

(Note: If each method takes $\mathcal{O}(n)$ time, running all three of them will still take $\mathcal{O}(n)$ time.)

Solution:

```
boolean isAvl(Node n)
    return isBst(n) && heightMatches(n) && isBalanced(n)

boolean isBst(Node n)
    return isBstHelper(n, null, null)

boolean isBstHelper(Node n, K minPossibleKey, K maxPossibleKey)
    if n == null:
        return true

    // If we know what the min or max possible keys are, check them

    if minKey != null && n.key < minPossibleKey
        return false

    if maxKey != null && maxPossibleKey < n.key
        return false
    boolean isLeftOk = isBstHelper(n.left, minPossibleKey, n.key);
    boolean isRightOk = isBstHelper(n.right, n.key, maxPossibleKey)
    return isLeftOk && isRightOk

boolean heightMatches(Node n)
    // Note: we use '-99' to mean that the heights don't match
    return heightMatchesHelper(n) != -99

boolean heightMatchesHelper(Node n)
    if n == null
        return -1

    int leftHeight = heightMatchesHelper(n.left)
    int rightHeight = heightMatchesHelper(n.right)

    if leftHeight == -99 || rightHeight == -99
        return -99

    int expectedHeight = max(leftHeight, rightHeight) + 1

    if n.height == expectedHeight
        return expectedHeight
    else
        return -99

boolean isBalanced(Node n)
    if n == null
        return true

    int leftHeight = if n.left == null then -1 else n.left.height
    int rightHeight = if n.right == null then -1 else n.right.height

    if abs(leftHeight - rightHeight) > 1
        return false

    return isBalanced(n.left) && isBalanced(n.right)
```