Section 04: Solutions

Section Problems

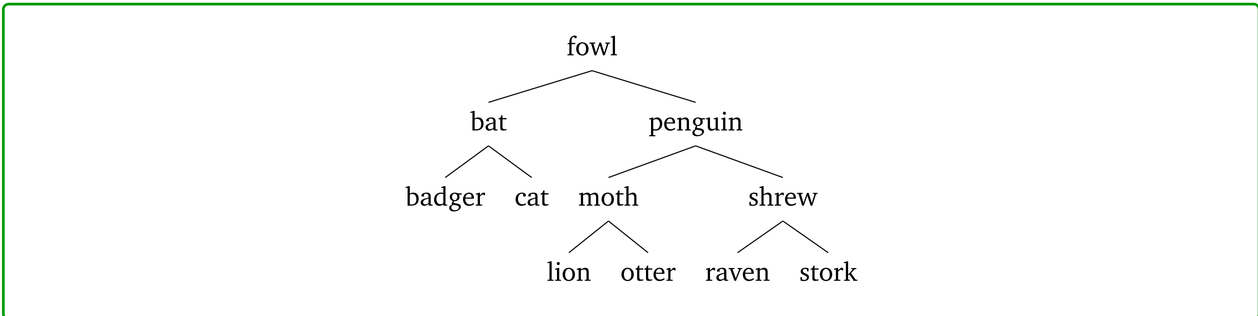
1. Constructing AVL trees

Draw an AVL Tree as each of the following keys are added in the order given. Show intermediate steps.

(a)

{“penguin”, “stork”, “cat”, “fowl”, “moth”, “badger”, “otter”, “shrew”, “lion”, “raven”, “bat”}

Solution:

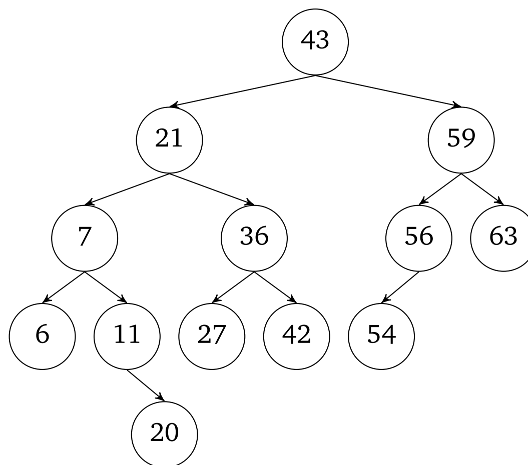


(b)

$$\{6, 43, 7, 42, 59, 63, 11, 21, 56, 54, 27, 20, 36\}$$

Solution:

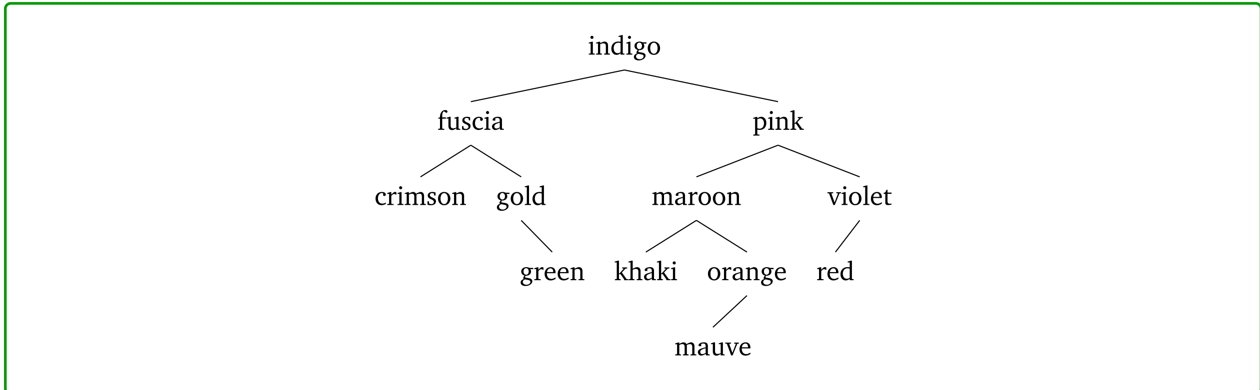
Note: Here's a step-by-step walkthough of this one!
https://docs.google.com/presentation/d/16JKXLmEYxkb7d7yLixsZMcqmaO2sBJgC9eDJuDLTMEs/edit?usp=sharing



(c)

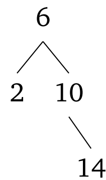
{“indigo”, “fuscia”, “pink”, “goldenrod”, “violet”, “khaki”, “red”, “orange”, “maroon”, “crimson”, “green”, “mauve”}

Solution:



2. AVL tree rotations

Consider this AVL tree:



Give an example of a value you could insert to cause:

(a) A single rotation

Solution:

Any value greater than 14 will cause a single rotation around 10 (since 10 will become unbalanced, but we’ll be in the line case).

(b) A double rotation

Solution:

Any value between 10 and 14 will cause a double rotation around 10 (since 10 will be unbalanced, and we’ll be in the kink case).

(c) No rotation

Solution:

Any value less than 10 will cause no rotation (since we can’t cause any node to become unbalanced with those values).

3. True or false?

(a) An insertion in an AVL tree with n nodes requires $\Theta(\log(n))$ 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. Although it is true that listing all elements in sorted order from a BST is $\mathcal{O}(n)$, the statement as a whole is false because 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 know that rotations do not introduce more imbalances below them, since they are a method of fixing imbalances.)

4. Big-$\mathcal{O}$

Write down a tight big-$\mathcal{O}$ for each of the following. Unless otherwise noted, give a bound in the worst case.

(a) Insert and find in a BST.

Solution:

$\mathcal{O}(n)$ and $\mathcal{O}(n)$, respectively. This is unintuitive, since we commonly say that `find()` in a BST is "`log(n)`", but we're asking you to think about *worst-case* situations. The worst-case situation for a BST is that the tree is a linked list, which causes `find()` to reach $\mathcal{O}(n)$.

(b) Insert and find in an AVL tree.

Solution:

$\mathcal{O}(\log(n))$ and $\mathcal{O}(\log(n))$, respectively. The worst case is we need to insert or find a node at height 0. However, an AVL tree is always a balanced BST tree, which means we can do that in $\mathcal{O}(\log(n))$.

(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:

$\mathcal{O}(n)$. An AVL tree is always a balanced BST tree, which means we only need to traverse the tree in in-order once.

5. 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 indices:

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

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

(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:

The state of the internal array will be:

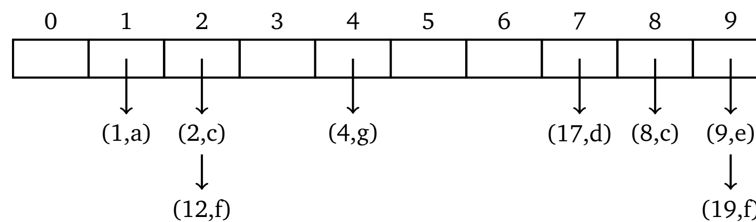
0	1	2	35	/	5	15	/	/	25

(d) Suppose we have a hash table implemented using separate chaining. This hash table has an internal capacity of 10. Its buckets are implemented using a linked list where new elements are appended to the end. Do not worry about resizing.

Show what this hash table internally looks like after inserting the following key-value pairs in the order given using the hash function $h(x) = x$:

$$(1, a), (4, b), (2, c), (17, d), (12, e), (9, e), (19, f), (4, g), (8, c), (12, f)$$

Solution:



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 likelihood 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. Analyzing dictionaries

(a) What are the constraints on the data types you can store in an AVL tree?

Solution:

The keys need to be orderable because AVL trees (and BSTs too) need to compare keys with each other to decide whether to go left or right at each node. (In Java, this means they need to implement `Comparable`). Unlike a hash table, the keys do *not* need to be hashable. (Note that in Java, every object is technically hashable, but it may not hash to something based on the object's value. The default hash function is based on reference equality.)

The values can be any type because AVL trees are only ordered by keys, not values.

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

Solution:

(i) You can iterate over an AVL tree in sorted order in $\mathcal{O}(n)$ time.

(ii) AVL trees never need to resize, so you don't have to worry about insertions occasionally being very slow when the hash table needs to resize.

(iii) In some cases, comparing keys may be faster than hashing them. (But note that AVL trees need to make $\mathcal{O}(\log n)$ comparisons while hash tables only need to hash each key once.)

(iv) AVL trees *may* be faster than hash tables in the worst case since they guarantee $\mathcal{O}(\log n)$, compared to a hash table's $\mathcal{O}(n)$ if every key is added to the same bucket. But remember that this only applies to pathological hash functions. In most cases, hash tables have better asymptotic runtime ($\mathcal{O}(1)$) than AVL trees, and in practice $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ have roughly the same performance.

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

Solution:

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

However, if you know that `insert` will be called more often than `find`, or if you know the keys will be inserted in a random enough order that the BST will stay balanced, you may prefer a BST since it avoids the small runtime overhead of checking tree balance properties and performing rotations. (Note that this overhead is a constant factor, so it doesn't matter asymptotically, but may still affect performance in practice.)

BSTs are also easier to implement and debug than AVL trees.

(d) 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 $h + 1$ (remember that height is the number of edges, so we visit $h + 1$ nodes going from the root to the farthest away leaf); the min number is 1 (when the element we're looking for is just the root).

(e) **Challenge Problem:** Consider an AVL tree with n nodes and a height of h. Now, consider a single call to `insert(...)`. What's the maximum possible of nodes `insert(...)` ends up visiting? The minimum possible? Don't count the new node you create or the nodes visited during rotation(s).

Solution:

The max number is $h + 1$. Just like a `get`, we may have to traverse to a leaf to do an insertion.

To find the minimum number, we need to understand which elements of AVL trees we can do an insertion at, i.e. which ones have at least one `null` child.

In a tree of height 0, the root is such a node, so we need only visit the one node.
In an AVL tree of height 1, the root can still have a (single) `null` child, so again, we may be able to do an insertion visiting only one node.

On taller trees, we always start by visiting the root, then we continue the insertion process in either a tree of height $h - 1$ or a tree of height $h - 2$ (this must be the case since the the overall tree is height h and the root is balanced). Let $M(h)$ be the minimum number of nodes we need to visit on an insertion into an AVL tree of height h. The previous sentence lets us write the following recurrence

$$M(h) = 1 + \min\{M(h - 1), M(h - 2)\}$$

The 1 corresponds to the root, and since we want to describe the minimum needed to visit, we should take the minimum of the two subtrees.

We could simplify this recurrence and try to unroll it, but it's easier to see the pattern if we just look at the first few values:

$$M(0) = 1, M(1) = 1, M(2) = 1 + \min\{1, 1\} = 2, M(3) = 1 + \min\{1, 2\} = 2, M(4) = 1 + \min\{2, 2\} = 3$$

In general, $M()$ increases by one every other time h increases, thus we should guess the closed-form has an $h/2$ in it. Checking against small values, we can get an exactly correct closed-form of:

$$M(h) = \lfloor h/2 \rfloor + 1$$

which is our final answer.

Note that we need a very special (as empty as possible) AVL tree to have a possible insertion visiting only $\lfloor h/2 \rfloor + 1$ nodes. In general, an AVL of height h might not have an element we could insert that visits only $\lfloor h/2 \rfloor + 1$. For example, a tree where all the leaves are at depth h is still a valid AVL tree, but any insertion would need to visit $h + 1$ nodes.

8. 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, 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 be a reasonable option to investigate as well.

big-O analysis only cares about very large values of n, since we only have 200 trains, big-O analysis might not be the right way to analyze this problem. Even if the binary search tree ends up being degenerate, searching through a linked list of only 200 element is realistically going to be a fast operation.

What's actually best will depend on the libraries you already have written, what hardware you actually run on, and how you want to balance code that will be sustainable if you get more trains vs. code that will be easy to understand and check for bugs right now.

9. Testing and debugging

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.

```
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
- If the hash table is full, the while loop will never end
- 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 middle bug is not trivial, but we have seen many examples of how to fix this. 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`
- There should be a size field, with `ensureCapacity()` functionality.
- 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 hash-codes 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).

Food For Thought

10. 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.

(a) 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.

There would obviously be some error checking embedded to avoid problems if the input isn't of valid format, but we typically don't include error checking in algorithm descriptions or pseudocode.

(b) Give pseudocode for this algorithm.

Solution:

```
int evaluateExpr(String expr)
    Stack s = new Stack()
    String[] exprs = expr.split(',')

    for (String s : exprs)
        if s == '+'
            int num1 = s.pop()
            int num2 = s.pop()
            s.push(num1 + num2)
        else if ... // other expressions not included, very similar to +
        else // no more exprs, probably a number
            s.push(parseInt(s))

    return s.pop()
```

11. 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 $\mathcal{O}(n)$ time, each of the n inserts take at most $\mathcal{O}(\log(n))$ time. So, the total runtime would be $\mathcal{O}(n \log(n))$.

If we want to be clever, we can actually do this in $\mathcal{O}(n)$ time. The key is to create the new AVL tree with a traversal instead of a set of insertions.

First, traverse through the BST in-order and store each element into an array. Note that the elements are now stored in the array in sorted order.

We can then perform what amounts to a post-order traversal on a new, balanced tree. We save the median of our current (sorted) list of elements, then make two recursive calls, one for the left half of the array, creates the left subtree (containing all the elements smaller than the median). The other call recusively creates a balanced right subtree (containing all the elements larger than the median). We can then create a node for the median, pointing to the results of the recursive calls.

In the base case, we have zero or one elements remaining and we return null or construct and returns a new node object, respectively.

The resulting tree will be balanced, and since the initial list was sorted, it will be a valid BST as well, thus the result is an AVL tree.

The code really is just a "traversal" of the new AVL tree it creates, so the running time is $\mathcal{O}(n)$, as claimed.

(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.

(As a reminder: 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)
```

Challenge Problems

12. Random hash functions

In class we talked about various strategies to minimize collisions. In this question we discuss how to use randomness to "spread out" collisions from a small set of very bad inputs into a larger set of almost-always-fine inputs. The last two parts of this problem are beyond the scope of this course, but are interesting nonetheless.

For simplicity, assume our keyspace (the set of possible keys) is the set $\{0, 1, 2, \ldots, 2^{30} - 1\}$. Suppose we have a hashtable of size 2^{10}. Let a be an odd integer less than 2^{30}.

Consider the hash function

$$h_a(x) = \left\lfloor \frac{(ax) \mod 2^{30}}{2^{20}} \right\rfloor$$

Notice that the function changes depending on the value of a we choose, so this is really a set of possible functions.

(a) Show that for any a, h_a outputs an integer between 0 and $2^{10} - 1$ (i.e. we can use this as a hash function for our table size)

Solution:

The numerator of the fraction is always a number from 0 to $2^{30} - 1$ (after we do the mod operation). Dividing by 2^{20} moves the number into range 0 to $2^{10} - 1/2^{20}$. When we take the floor, we round the numbers down to the next integer, so the range of possible outputs becomes 0 to $2^{10} - 1$, i.e. exactly the indices for a 0-indexed table of size 2^{10}.

(b) Choose $a = 1$, i.e. the hash function simplifies to

$$h_1(x) = \left\lfloor \frac{x \mod 2^{30}}{2^{20}} \right\rfloor$$

For this function, find a large set of elements that all hash to 0.

Solution:

The keys $\{0, 1, 2, \ldots, 2^{20} - 1\}$ all hash to 0 (modding by 2^{30} doesn't affect small values, and the floor rounds any number less than 1 to 0). For $a = 1$ this function hashes contiguous sets of 2^{20} numbers into each bin. For other hash functions, it is harder to find this set, but every hash function has this problem: if the key-space is much larger than the size of the table, there must be a large number of values that all collide.

(c) Let x, y be any of the two elements you found in the last part. Choose a few thousand values of a, and check whether $h_a(x) = h_a(y)$ for each of them (write code for this part). For what fraction of these hash functions do x, y collide? If the values of the hash function were totally random, how often would you expect collisions?

Solution:

The exact number of collisions you see will depend on which values of a, x, y you check, but you should see between 0.1% and 0.2% of hash functions causing a collision.

If the outputs of the hash function were truly random, we would see a collision with probability equal to $\frac{1}{\text{table size}}$, i.e. $1/1024$ or about $.1\%$ of the time. So the output we're seeing as we change a is *nearly* as good as really random outputs.

(d) The following statement is true (explaining why is beyond the scope of the course): For any x, y if you choose a at random, the probability that $h_a(x) = h_a(y)$ is at most $2/2^{10}$.

Use this fact, or your observations in the last part, to explain why you might decide to choose a random a instead of just choosing $a = 1$ (hint: imagine you know someone is using the hash function with $a = 1$, how can you use the first part to slow their code down? Can you do the same for a random a?)

Solution:

A user will have a fixed set of keys they need to use. If these happen to be keys that all hash to the same place, the hash table will have very poor performance, and the user has no hope of fixing this (because they can't really change their keys). On the other hand, if we choose a random hash function, with high probability (say 99.9%) the fixed set of keys won't be a problem (though there is still a small chance of getting the poor performance).

Occasionally, we worry about an attacker intentionally giving us bad data to try to break our code. If an attacker knows your hash function, they can do what we did in the first part, and find a set of inputs that will slow down your hash table. On the other hand, if you're choosing a hash function randomly, there is no single set of inputs that can cause bad performance. Choosing a function randomly lets us "spread out" the bad behavior across inputs. Said a different way:

- With a single, fixed function for most inputs, things work great; but there are some very bad inputs on which things work terribly.
- If you choose a random hash function, every input has a very high probability of good performance, but on many inputs there is a very small chance of bad performance.

There are a lot of mathematical caveats here (e.g. you need a good set of hash functions to choose from, your choice of hash function needs to be random enough that it can't be predicted, etc.) but we don't have time to go into them. See http://jeffe.cs.illinois.edu/teaching/algorithms/notes/12-hashing.pdf for more information on this hashing scheme, and randomized hashing in general, including the formal mathematics.