Section 05: Solutions

1. Asymptotic Analysis

(a) **Applying definitions**

For each of the following, choose a c and n_0 which show $f(n) \in \mathcal{O}(g(n))$. Explain why your values of c and n_0 work.

(i) $f(n) = 5000n^2 + 6n\sqrt{n}$ and $g(n) = n^3$

Solution:

We are trying to find a c and n_0 such that $5000n^2 + 6n\sqrt{n} \leq cn^3$ is true for all values of $n \geq n_0$. We start by finding a c and n_0 separately for each term in $5000n^2 + 6n\sqrt{n}$.

$$\begin{aligned} 5000n^2 &\leq 5000n^3 && \text{for all } n \geq 1 \\ 6n\sqrt{n} = 6n^{3/2} &\leq 6n^3 && \text{for all } n \geq 1 \end{aligned}$$

Both inequalities are true as long as $n \geq 1$, so we can add them together to get

$$5000n^2 + 6n\sqrt{n} \leq (5000 + 6)n^3 = 5006n^3$$

for all $n \geq 1$.

So, we can pick $n_0 = 1$ and $c = 5006$. This is not the only solution; many others are possible.

(ii) $f(n) = n(4 + \log(n))$ and $g(n) = n^2$

Solution:

After simplifying $f(n)$, we have $f(n) = 4n + n\log(n)$. We will go term-by-term to find a c and n_0 such that $4n + n\log(n) \le cn^2$ is true for all $n \ge n_0$.

$$\begin{aligned} 4n &\le 4n^2 && \text{for all } n \ge 1 \\ n\log(n) &\le n^2 && \text{for all } n \ge 1 \end{aligned}$$

We add the inequalities together to get

$$4n + n\log(n) \le (4+1)n^2 = 5n^2$$

for all $n \ge 1$.

So, we can pick $n_0 = 1$ and $c = 5$.

(iii) $f(n) = 2^n$ and $g(n) = 3^n$

Solution:

As before, we must find a c and n_0 such that $2^n \le c3^n$ for all $n \ge n_0$.

Notice that since $2 \ge 3$, multiplying together 2 n times is always going to be smaller the multiplying together 3 n times (so long as n is a positive number.

So, we can pick $c = 1$ and $n_0 = 1$.

(b) **Runtime Analysis**

For each of the following, give a Θ-bound for runtime of the algorithm/operation:

(i) Best-case **get** in a binary search tree of size n.

Solution:

$\Theta(1)$ when the data we're looking for is at the root.

(ii) Best-case **put** in a hash table with size n and a current $\lambda = 1$ if the collision resolution is:

- Separate chaining
- Linear Probing

Solution:

For separate chaining, $\Theta(1)$. For linear probing, $\Theta(n)$. (If $\lambda = 1$, we must resize first.)

(iii) Pop a value off a stack containing n elements implemented as an array.

Solution:

$\Theta(1)$

(iv) Finding the minimum value in a BST of size n.

Solution:

In the worst case, $\Theta(n)$ (the tree could be degenerate, skewing left).

In the average case, where the tree is balanced, $\Theta(\log(n))$.

In the best case, $\Theta(1)$ (the tree could be degenerate, skewing right).

(v) Finding the minimum value in a AVL tree of size n.

Solution:

$\Theta(\log(n))$

(vi) Print out values in AVL tree of size n.

Solution:

$\Theta(n)$

(vii) Iterating through and printing every element in an array list using a for loop and the `get(i)` method.

Solution:

$\Theta(n)$

(viii) Pop on a stack containing n elements implemented as a singly-linked list.

Solution:

The answer depends on how the stack is implemented.

If we pop from the front, $\Theta(1)$. If we pop from the end (which would be a somewhat silly decision), $\Theta(n)$.

(ix) Inserting a value into an AVL tree of size n, where the value you are inserting is smaller than any other values currently in the tree.

Solution:

$\Theta(\log(n))$

2. Eyeballing Big-Θ bounds

For each of the following code blocks, what is the worst-case runtime? Give a big-Θ bound. You do not need to justify your answer.

(a)
```
void f1(int n) {
    int i = 1;
    int j;
    while(i < n*n*n*n) {
        j = n;
        while (j > 1) {
            j -= 1;
        }
        i += n;
    }
}
```

Solution:

$\Theta(n^4)$

One thing to note that the while loop has increments of $i+=n$. This causes the outer loop to repeat n^3 times, not n^4 times.

(b)
```
int f2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            System.out.println("j = " + j);
        }
        for (int k = 0; k < i; k++) {
            System.out.println("k = " + k);
            for (int m = 0; m < 100000; m++) {
                System.out.println("m = " + m);
            }
        }
    }
}
```

Solution:

$\Theta(n^2)$

Notice that the last inner loop repeats a small constant number of times – only 100000 times.

(c)
```
int f3(n) {
    count = 0;
    if (n < 1000) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < i; k++) {
                    count++;
                }
            }
        }
    } else {
        for (int i = 0; i < n; i++) {
            count++;
        }
    }
    return count;
}
```

Solution:

$\Theta(n)$

Notice that once n is large enough, we always execute the 'else' branch. In asymptotic analysis, we only care about behavior as the input grows large.

(d)
```
void f4(int n) {
    IList<Integer> arr = new DoubleLinkedList<>();
    for (int i = 0; i < n; i++) {
        if (list.size() > 20) {
            list.remove(0);
        }
        list.add(i);
    }
    for (int i = 0; i < list.size; i++) {
        System.out.println(list.get(i));
    }
}
```

Solution:

$\Theta(n)$

Note that `arr` would have a constant size of 20 after the first loop. Since this is a `DoubleLinkedList`, `add` and `remove` would both be $\Theta(1)$.

3. Recurrences and Summations

Find an expression for the total work of the following expressions using the Tree Method, then simplify that summation to a closed form. If possible, check that the big-$\mathcal{O}$ is correct with Master Theorem.

Hint: Just as a reminder, here are the steps you should go through for **any** Tree Method Problem:

i. Draw the recurrence tree.

ii. What is the size of the **input** to each node at level i? What is the amount of **work** done by each node at the i-th *recursive* level? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the input should equal n.

iii. What is the total number of nodes at level i? As in class, we call the root level $i = 0$. This means that at $i = 0$, your expression for the total number of nodes should equal 1.

iv. What is the total work done across the i-th *recursive* level?

v. What value of i does the last level of the tree occur at?

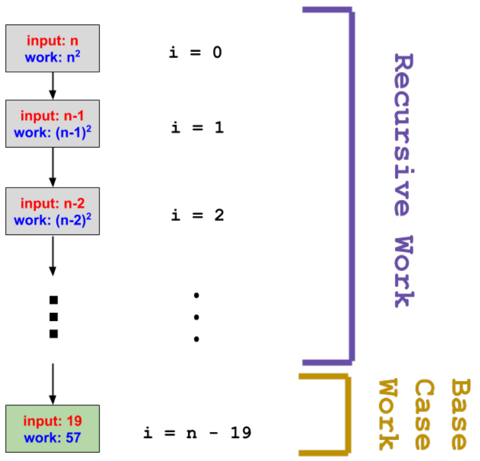
vi. What is the total work done across the base case level of the tree (i.e. the last level)?

vii. Combine your answers from previous parts to get an expression for the total work.

(a) $T(n) = \begin{cases} T(n-1) + n^2 & \text{if } n > 19 \\ 57 & \text{otherwise} \end{cases}$

Solution:

(i) Here's a drawing of the tree:



(ii) The input size at level i is $n - i$ (since we subtract the input by 1 at each level). This makes the work in each recursive case node $(n-i)^2$.

(iii) The number of nodes at any level i is 1 (since each node has a single child).

(iv) Multiplying the work per recursive case node by the recursive nodes per level, we get: $1 \cdot (n-i)^2$.

(v) The last level of the tree is when $n - i = 19$. Solving for i gives us $i = n - 19$.

(vi) The number of nodes in the base case level is 1, and each node does 57 work, so the total work is $1 \cdot 57$.

(vii) Summing up work across all recursive levels and then adding in the base case work, we get:

$$57 + \sum_{i=0}^{(n-19)-1} (n-i)^2$$

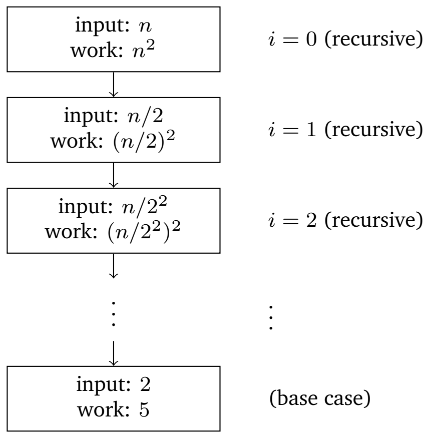
(viii) We now simplify the above summation to a closed form:

$$\begin{aligned}
&57 + \sum_{i=0}^{n-20} (n-i)^2 \\
&= 57 + \sum_{i=0}^{n-20} n^2 - 2in + i^2 \\
&= 57 + \sum_{i=0}^{n-20} n^2 - \sum_{i=0}^{n-20} 2in + \sum_{i=0}^{n-20} i^2 \\
&= 57 + n^2 \sum_{i=0}^{n-20} 1 - 2n \sum_{i=0}^{n-20} i + \sum_{i=0}^{n-20} i^2 \\
&= 57 + n^2(n-19) - 2n\frac{(n-19)(n-20)}{2} + \frac{(n-19)(n-20)(2[n-19]-1)}{6}
\end{aligned}$$

We now have a closed form since there's no more recursion and no more summations. If we just want a big-O expression, the dominating term is $\mathcal{O}\left(n^3\right)$. Unfortunately, we can't check our answer with Master Theorem: the recurrence is not in the proper form!

(b) $T(n) = \begin{cases} T(n/2) + n^2 & \text{if } n \geq 4 \\ 5 & \text{otherwise} \end{cases}$

Solution:

(i) 

We assume the input size is a power of 2, so the base case input size is 2 instead of 3.

(ii) The input size is $n/2^i$ since we divide the input by 2 at each level, so the work at each recursive node is $(n/2^i)^2 = n^2/2^{2i}$.

(iii) There is only 1 node at every level.

(iv) The total work at each recursive level is $1 \cdot n^2/2^{2i}$.

(v) The last level of the tree is when $n/2^i = 2$, so $i = \log_2 n - 1$.

(vi) Since there is only 1 node at the base case level, the total work is $1 \cdot 5$.

(vii)

$$5 + \sum_{i=0}^{\log_2 n - 2} \frac{n^2}{2^{2i}}$$

(viii)

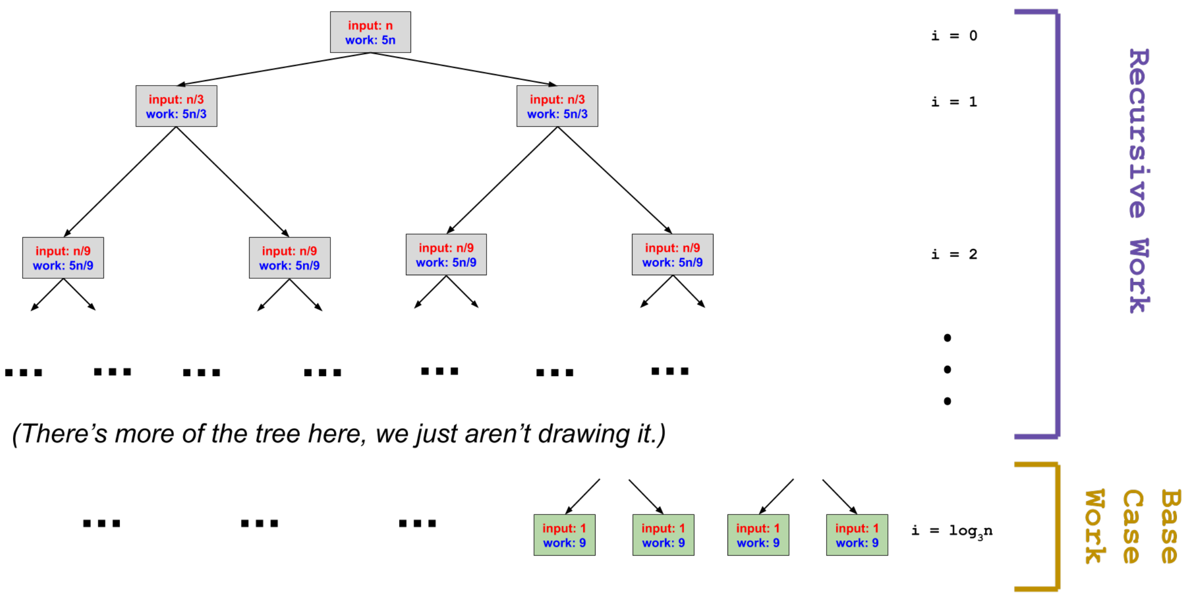
$$\begin{aligned} 5 + \sum_{i=0}^{\log_2 n - 2} \frac{n^2}{2^{2i}} &= 5 + n^2 \sum_{i=0}^{\log_2 n - 2} \frac{1}{4^i} \\ &= 5 + n^2 \sum_{i=0}^{\log_2 n - 2} \left(\frac{1}{4}\right)^i \\ &= 5 + n^2 \frac{1/4^{\log_2 n - 1} - 1}{1/4 - 1} \end{aligned}$$

This is a closed form, so we can stop simplifying here. It's a good exercise to simplify this expression to the point that you can clearly tell what the big-$\mathcal{O}$ is. You can check with Master Theorem (since $\log_2(1) < 2$) that the answer is $\mathcal{O}\left(n^2\right)$. Hint: Rewrite $1/4^{\log_2(n)-1}$ as $1/\left(2^{\log_2(n/2)}\right)^2$.

(c) $T(n) = \begin{cases} 2T(n/3) + 5n & \text{if } n > 1 \\ 9 & \text{otherwise} \end{cases}$

Solution:

(i) Here's a drawing of the tree:



(ii) The input size at level i is $n/3^i$ (since we divide the input by 3 at each level). This makes the work in each recursive case node $5 \cdot (n/3^i)$.

(iii) The number of nodes at level i is 2^i (since each node has two children).

(iv) Multiplying the work per recursive case node by the recursive nodes per level, we get: $5 \cdot \frac{n}{3^i} \cdot 2^i = 5n \cdot \left(\frac{2}{3}\right)^i$.

(v) The last level of the tree is when $n/3^i = 1$. Solving for i gives us $i = \log_3(n)$.

(vi) Work across the base case level: The number of nodes (from previous parts) is $2^{\log_3(n)}$, and each node does 9 work, so the total work is $9 \cdot 2^{\log_3(n)}$.

(vii) Summing up work across all recursive levels and then adding in the base case work, we get:

$$\sum_{i=0}^{\log_3(n)-1} 5n \left(\frac{2}{3}\right)^i + 9 \cdot 2^{\log_3(n)}$$

(viii) Simplifying to a closed form:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_3(n)-1} 5n\left(\frac{2}{3}\right)^i + 9\cdot 2^{\log_3(n)} \\
&= 5n\sum_{i=0}^{\log_3(n)-1} \left(\frac{2}{3}\right)^i + 9\cdot 2^{\log_3(n)} \\
&= 5n\frac{\left(\frac{2}{3}\right)^{\log_3(n)} - 1}{2/3-1} + 9\cdot 2^{\log_3(n)}
\end{aligned}$$

This is a "closed form" so we can stop simplifying here.

If we want to check our big-$\mathcal{O}$, we can simplify further to find the dominating term:

$$\begin{aligned}
T(n) &= 5n\cdot\frac{\left(\frac{2}{3}\right)^{\log_3(n)} - 1}{2/3-1} + 9\cdot 2^{\log_3(n)} \\
&= 5n\cdot\frac{2^{\log_3(n)}/3^{\log_3(n)} - 1}{-1/3} + 9\cdot 2^{\log_3(n)} \\
&= -15n\cdot\left(n^{\log_3(2)}/n - 1\right) + 9\cdot n^{\log_3(2)} \\
&= 15n - 15n^{\log_3(2)} + 9\cdot n^{\log_3(2)}
\end{aligned}$$

Notice that $\log_3(2) < 1$, so $15n$ is the dominating term, and the running time is $\mathcal{O}(n)$. Applying Master Theorem, we check that $\log_3(2) < 1$, so our expression should be $\mathcal{O}(n)$, which matches our answer.

4. Modeling

Consider the following method. Let n be the integer value of the `n` parameter, and let m be the length of `DoubleLinkedList`. You may assume that $n > 7$.

```
public int mystery(int n, DoubleLinkedList<Integer> list) {
    if (n < 7) {
        System.out.println("???");
        int out = 0;
        for (int i = 0; i < n; i++) {
            out += i;
        }
        return out;
    } else {
        System.out.println("???");
        System.out.println("???");
        out = 0;
        for (int i : list) {
            out += 1;
            for (int j = 0; j < list.size(); j++) {
                System.out.println(list.get(j));
            }
        }
        return out + 2 * mystery(n - 4, list) + 3 * mystery(n / 2, list);
    }
}
```

Note: your answer to all three questions should be a recurrence, possibly involving a summation. You do not need to find a closed form.

(a) Construct a mathematical function modeling the *approximate* worst-case runtime of this method in terms of n and m.

Solution:

$$T(n,m) = \begin{cases} 1 & \text{when } n < 7 \\ m^3 + T(n-4,m) + T(n/2,m) & \text{otherwise} \end{cases}$$

(b) Construct a mathematical function modeling the *exact* integer output of this method in terms of n and m.

Solution:

$$I(n,m) = \begin{cases} \sum_{i=0}^{n-1} i & \text{when } n < 7 \\ m + 2I(n-4,m) + 3I(n/2,m) & \text{otherwise} \end{cases}$$

(c) Construct a mathematical function modeling the *exact* number of lines printed out in terms of n and m.

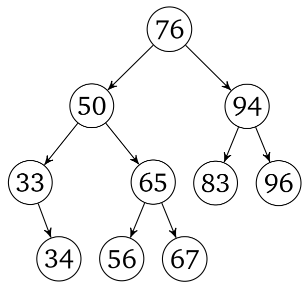
Solution:

$$P(n,m) = \begin{cases} 1 & \text{when } n < 7 \\ 2 + m^2 + P(n-4,m) + P(n/2,m) & \text{otherwise} \end{cases}$$

5. AVL/BST

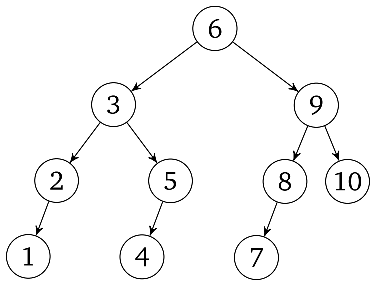
(a) Insert $\{94, 33, 50, 76, 96, 67, 56, 65, 83, 34\}$ into an initially empty AVL tree.

Solution:



(b) Insert $\{6, 5, 4, 3, 2, 1, 10, 9, 8, 6, 7\}$ into an initially empty AVL tree.

Solution:



(c) Suppose you insert 7 elements into a BST. What are the possible heights after the insertions? (Think about different orders of inserting elements).

Solution:

Any height from 2 (a tree where each internal node has exactly two children) to 6 (a degenerate tree) is possible.

(d) If you insert 7 elements into an AVL tree, what are the possible heights of the tree?

Solution:

An AVL tree with 7 elements could be height 2 or 3. It cannot be height 4: if the height is 4, then for the root to be balanced, one subtree must have height 3 and the other at least 2. A height 2 AVL tree needs at least 4 elements (or the root of the subtree won't be balanced), so in our 7 element tree, we have 4 elements for one subtree, plus the root, which leaves only 2 elements for the other subtree. But that subtree was supposed to be height 3, so there just aren't enough elements to fill it out.

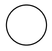
Notice that the AVL tree doesn't guarantee you the minimum possible height (there are AVL trees with 7 elements of height 3), but it does avoid the worst case.

(e) More generally, what is the minimum number of nodes in an AVL tree of height 4? Draw an instance of such an AVL tree.

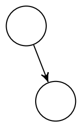
Solution:

We can approach this problem iteratively (or recursively, depending on your point of view).

First, what is the smallest possible AVL tree of height 0? Well, just a single node:



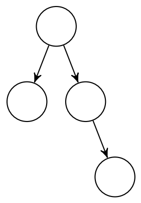
What about the smallest possible AVL tree of height 1? With some thought, we arrive at this answer:



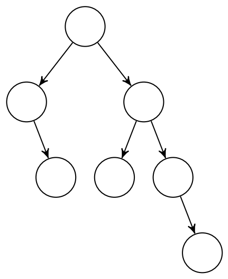
What about the smallest possible AVL tree of height 2? Well, if we think about this critically, we know that our tree must contain at least one subtree of height 1 (if it's any other height, our AVL tree wouldn't have height 2). If we're trying to minimize the number of nodes, we might as well make the other subtree have a height of 0. That minimizes the total number of nodes.

Now, if only we know what the smallest possible AVL trees of height 1 and 0 were...

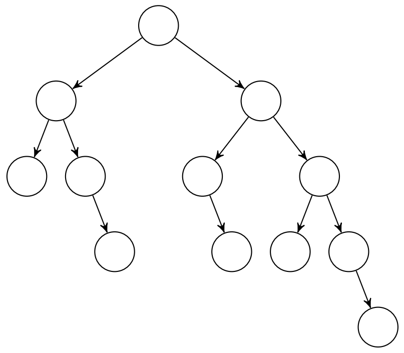
We can reuse our answers above to get:



We repeat to get the smallest possible AVL tree of height 3: combine together the smallest possible AVL trees of heights 1 and 2:



We repeat one more time, for the smallest possible AVL tree of height 4:



If we count the number of nodes, we get 12.

More generally, we can find the number nodes in the smallest possible AVL tree by computing the following recurrence:

$$\text{minNumNodes}(h) = \begin{cases} 1 & \text{if } h = 0 \\ 2 & \text{if } h = 1 \\ 1 + \text{minNumNode}(n-2) + \text{minNumNodes}(n-1) & \text{otherwise} \end{cases}$$

(f) Describe an implementation of `delete()` for a BST.

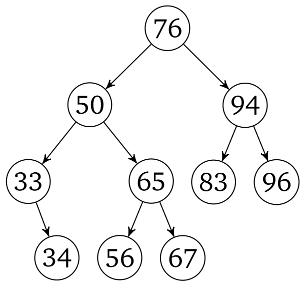
Solution:

There are a few different cases we have to consider when deleting data from a tree with really rigid structure rules like a BST or AVL tree. Let's start with the easiest.

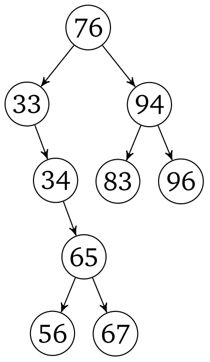
When the node we want to delete is a leaf node, where it has no left or right children, then great! Just set the reference pointing to it to null. Easy.

What about a branch node, such that the node has a left or right child (but not both)? Hmm, well that gets a little trickier, but no worries, we just substitute that deleted node with its singular child. No biggie.

But what about a branch or root node with two children? Which one gets to be the replaced node? We'll get to the lazy solution soon, but let's say you *absolutely must* replace the node. Maybe your project manager is super concerned with that small amount of memory. What should that replaced tree look like? Say we had a tree that looked like this:



And then let's say we want to call `delete(50)`. What should the replaced subtree look like? Well, we want the tree to keep its BST property for either BST or AVL tree, of course. This means that the replaced subtree's root must be such a value that it maintains sortedness (its left values are less, right values are greater). There are a few possible solutions that depend on implemenation, but one resulting tree could be:



Now, this resulting tree seems like a stretch, but hear me out. All we did was replace root with the left subtree, and delegate the right subtree to be the *right-most thing* in the left subtree. This is still a valid BST tree. You can flip this around to replace root with the right subtree and delegate the left subtree tot be the left-most thing in the right subtree, of course. You do you.

The easiest way to actually implement this is to use recursion, where `delete()` is called with both a key and a node, and the return value is the node we want to replace the given node with. If we don't want to replace the given node with anything, we can just return the given node itself. In pseudocode, we could write `delete()` like this:

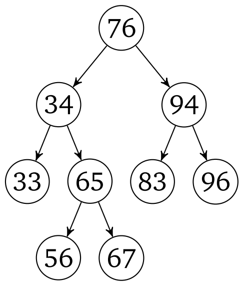
```
function delete($key$)
    Set the root node of the BST to the result of calling delete($key$, root node)
function delete($key$, $node$)
    if $node$ is null then return null
    else if $key$ is less than $node$'s key then
        Set $node$'s left node to the result of calling delete($key$, left node)
        return $node$
    else if $key$ is greater than $node$'s key then
        Set $node$'s right node to the result of calling delete($key$, right node)
        return $node$
                ▷ If we reach this point in the method, we know that $node$ is the one we want to delete.
    else if $node$ has no children then return null
    else if $node$ has one child then return $node$'s child
    else
        Let $r$ be $node$'s right child
        Traverse $node$'s left subtree to find its right-most node and let $x$ be this node
        Set $r$ as the right child of $x$
        return $node$'s left child
```

(g) Adapt `delete()` for a BST to work for an AVL tree. Don't worry if there are some edge cases that don't quite work yet, the next part will help with those.

Solution:

If you remember from the last problem, clearly we *had* a valid AVL tree, and now it's not. What the heck? The new tree is more unbalanced than any AVL tree we've seen before. If you'll notice, the 34 node is the bottom-most non-balanced node, but no insertion into an AVL tree would ever naturally produce that, so we never wrote rotations for this situation. Trying to make rotations fix this seems complicated. Let's try to find some other way.

Your PM is still yelling at you to save memory. One solution is to create a new AVL subtree, inserting each of the subtree's data one by one, and then replace the deleted node with this new AVL subtree. Without worrying about efficiency right now (you won't like it anyway), we could get a tree that looks like this:



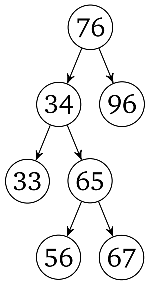
And all is well in the world.

If you really wanted to do some rotations, there is a correct way to do it, and it's been referenced in lecture a little bit but won't be reiterated here. Someone on the internet has done a very good job of explaining this better than we can here! This is a good resource, I highly recommend looking into the specifics of it here if you absolutely must rotate.

(h) Make a more robust (not buggy) `delete()` for an AVL tree, and try to be as (**hint**) lazy as possible.

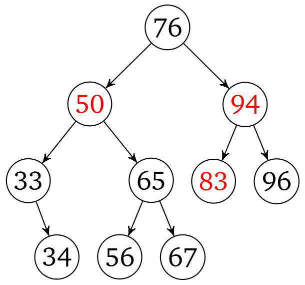
Solution:

What happens when you delete enough stuff that your tree becomes imbalanced? Despite your best efforts to balance the subtrees? Imagine calling `delete(83)` and then `delete(94)`. The tree would look like this:



And even though we balanced our subtrees, the tree as a whole is not balanced, and yet again, we have no rotations (unless you went to the resource above) to help us. There must be a better way.

Well, as it so happens, you've probably heard that programmers are *lazy*. This is true. Why else do we get computers to do stuff for us? Well, one super common way to implement `delete()` (in any structure, not just AVL trees) is to implement a *dirty bit*. This is where we mark a node as "deleted" ("she doesn't even go here!") but don't actually change the tree at all. Sure, sometimes we get extra stuff that we don't need, but this is far easier to implement, and after all three delete calls, we get a tree that looks like this:



We know which nodes "aren't really in the tree" and the tree is still a valid AVL tree. We didn't save the memory your PM wanted, but hopefully you can bake them some apologetic cookies.

6. Hash tables

(a) What is the difference between primary clustering and secondary clustering in hash tables?

Solution:

Primary clustering occurs after a hash collision causes two of the records in the hash table to hash to the same position, and causes one of the records to be moved to the next location in its probe sequence. Linear probing leads to this type of clustering.

Secondary clustering happens when two records would have the same collision chain if their initial position is the same. Quadratic probing leads to this type of clustering.

(b) Suppose we implement a hash table using double hashing. Is it possible for this hash table to have clustering?

Solution:

Yes, though the clustering is statistically less likely to be as severe as primary or secondary clustering.

(c) Suppose you know your hash table needs to store keys where each key's hash code is always a multiple of two. In that case, which resizing strategy should you use?

Solution:

Any strategy where the size of the table is not a multiple of two. For example, we could either make the hash table's array have an odd initial starting size then double to resize, or we could have use the prime doubling strategy.

If the table size *were* a multiple of two, we would end up using only half the available table entries (if we were using separate chaining) or have more collisions then usual (if we were using open addressing).

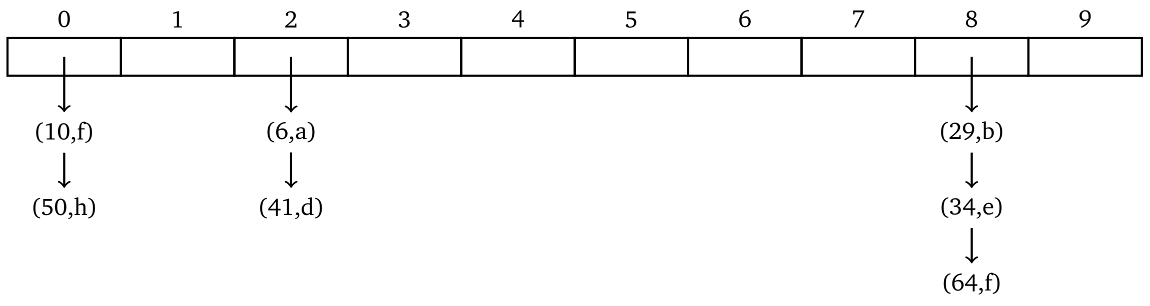
(d) Consider the following key-value pairs.

(6, a), (29, b), (41, d). (34, e), (10, f), (64, g), (50, h)

Suppose each key has a hash function $h(k) = 2k$. So, the key 6 would have a hash code of 12. Insert each key-value pair into the following hash tables and draw what their internal state looks like:

(i) A hash table that uses separate chaining. The table has an internal capacity of 10. Assume each bucket is a linked list, where new pairs are appended to the end. Do not worry about resizing.

Solution:



(ii) A hash table that uses linear probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10, f)	(64, g)	(6, a)	(41, d)	(50, h)				(29, b)	(34, e)

(iii) A hash table that uses quadratic probing, with internal capacity 10. Do not worry about resizing.

Solution:

0	1	2	3	4	5	6	7	8	9
(10,f)	(50, h)	(6,a)	(41,d)				(64, g)	(29,b)	(34,e)

(e) Consider the three hash tables in the previous question. What are the load factors of each hash table?

Solution:

$$\lambda = \frac{7}{10} = 0.7$$

7. Debugging

Suppose we are trying to implement an algorithm `isTree(Node node)` that detects whether some binary tree is a valid one or not, and returns `true` if it is, and `false` if it isn't. In particular, we want to confirm that the tree does not contain any *cycles* – there are no nodes where the `left` and `right` fields point to their parents.

Assume that the node passed into the method is supposed to be the root node of the tree.

(a) List at least four different test cases for each problem. For each test case, be sure to specify what the input is (drawing the tree, if necessary), and what the expected output is (assuming the algorithm is implemented correctly).

Solution:

Some examples of inputs. On an exam, be sure to provide a wide variety of test cases. Some of your cases should test the "happy" case, some cases should test weird inputs, some cases should test invalid/incorrect input...

(i) Input: null
Output: true

(ii) Input: a regular, standard tree
Output: true

(iii) Input: a degenerate tree that looks like a linked list
Output: true

(iv) Input: a tree containing a node where either the `left` or `right` pointers is pointing to itself
Output: false

(v) Input: a tree containing a node where either the `left` or `right` pointers is pointing to some parent node
Output: false

(vi) Input: a tree containing a node where the `left` or `right` pointers is pointing to a neighbor
Output: false

(b) Here is one (buggy) implementation in Java. List every bug you can find with this algorithm.

```java
public class Node {
    public int data;
    public Node left;
    public Node right;

    // We are simplifying what the equals method is supposed to look like in Java.
    // This method is technically invalid, but you may assume it's
    // implemented correctly.
    public boolean equals(Node n) {
        if (n == null) {
            return false;
        }
        return this.data == n.data;
    }

    public int hashCode() {
        // Pick a random number to ensure good distribution in hash table
        return Random.randInt();
    }
}

boolean isTree(Node node) {
    ISet<Node> set = new ChainedHashSet<>();
    return isTreeHelper(node, set)
}

boolean isTreeHelper(Node node, ISet<Node> set) {
    ISet<Node> set = new ChainedHashSet<>();
    if (set.containsKey(node)) {
        return false;
    } else {
        set.add(node);
        return isTreeHelper(node.left, set) || isTreeHelper(node.right, set);
    }
}
```

Solution:

Some bugs:

(i) The method does not attempt to handle the case where the input node is null

(ii) If the hashCode returns a random int every time it's called, it's going to play havoc with the set – we wouldn't be able to reliably insert the node anywhere.

(iii) We are trying to use the `ChainedHashSet` as a way of keeping track of every node object we've previously visited. However, we override the set with a new, empty one on each recursive call, wiping out our progress.

(iv) The "or" in the last line should be an "and" – if the left subtree or the right subtree isn't actually a tree, we should return 'false' right away. Currently, we return false only if *both* subtrees aren't actually trees.

8. Design Decisions

There isn't a design decisions problem in this handout, but you should review the questions from previous weeks. Some design questions in previous sections:

Section 1: Problems 2 and 3
Section 4: Problems 8 and 10