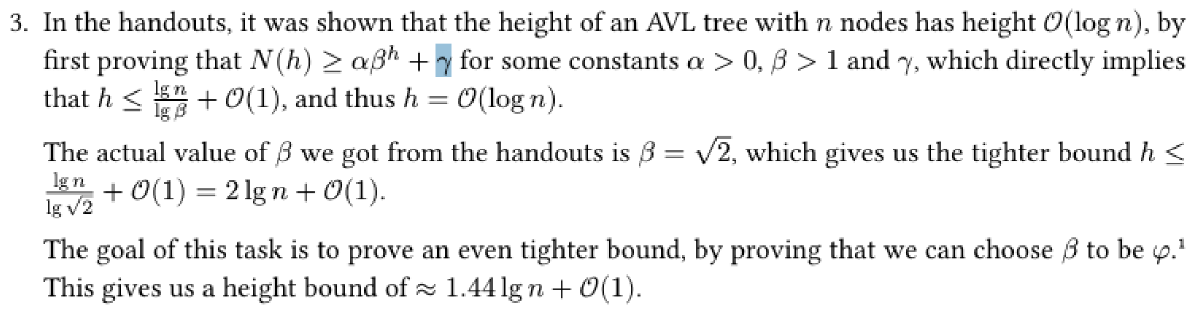
done with 1 and 2 in submissions, starting from 3



3. In the handouts, it was shown that the height of an AVL tree with n nodes has height $\mathcal{O}(\log n)$, by first proving that $N(h) \geq \alpha\beta^h + \gamma$ for some constants $\alpha > 0$, $\beta > 1$ and γ, which directly implies that $h \leq \frac{\lg n}{\lg \beta} + \mathcal{O}(1)$, and thus $h = \mathcal{O}(\log n)$.

   The actual value of β we got from the handouts is $\beta = \sqrt{2}$, which gives us the tighter bound $h \leq \frac{\lg n}{\lg \sqrt{2}} + \mathcal{O}(1) = 2 \lg n + \mathcal{O}(1)$.

   The goal of this task is to prove an even tighter bound, by proving that we can choose β to be φ.[1] This gives us a height bound of $\approx 1.44 \lg n + \mathcal{O}(1)$.

3.1. Let f be defined recursively as f(0) = 1, f(1) = 2 and for n >= 2,

f(n) = f(n-1) + f(n-2) + 1.

Prove that N(h) >= f(h) for all h >= 0.

We prove by strong induction on h.

[Base Cases]

h = 0: N(0) = 1 = f(0)
h = 1: N(1) = 2 = f(1)
Thus, N(h) >= f(h) holds.

[Inductive Step] Assume for all k < h that N(k) >= f(k).

For h >= 2, an AVL tree of height h with minimal nodes must have one child of height h-1 and the other of height at least h-2. Hence:

N(h) >= N(h-1) + N(h-2) + 1

By the inductive hypothesis, N(h) >= f(h-1) + f(h-2) + 1.

But by definition of f: f(h-1)+f(h-2)+1=f(h).

Therefore, N(h) >= f(h), completing the induction and the proof.

3.2. Show that the equation $x^2 = x + 1$ has exactly two solutions, both of which are real numbers, and find the largest one. Call this constant b.

Suppose r is a solution, so that $r^2 = r + 1$. Then we have:

$$r^2 - r = 1$$
$$4r^2 - 4r = 4$$
$$4r^2 - 4r + 1 = 5$$
$$(2r - 1)^2 = 5$$
$$2r - 1 = \{\sqrt{5}, -\sqrt{5}\}$$
$$r \in \{(1+\sqrt{5})/2, (1-\sqrt{5})/2\}$$

Thus, the only possible solutions are $(1+\sqrt{5})/2$, $(1-\sqrt{5})/2$, we can verify that these solutions work. We pick the largest of the two: $1+\sqrt{5})/2 = \phi$.

3.3. Show that there exists a constant a > 0 such that $f(n) >= ab^n$ for all n >= 0.

We have $b = \phi$, the positive root of $x^2 = x+1$. Choose a = 1. We prove by strong induction that $f(n) >= b^n$ for all n >= 0.

[Base Cases]

$n = 0$: $f(0) = 1 >= b^0 = 1$
$n = 1$: $f(1) = 2 >= b^1 \approx 1.618$

Thus, the base cases hold for n= 0, 1.

[Inductive Step] Assume for all k < n that $f(k) >= b^k$. For n >= 2, by definition of f:

$$f(n) = f(n-1) + f(n-2) + 1.$$

By the inductive hypothesis: $f(n) >= b^{n-1} + b^{n-2} + 1$.

since $b^2 = b + 1$, we have $b^{n-1} + b^{n-2} = b^{n-2}(b+1) = b^{n-2}\, b^2 = b^n$

Thus, $f(n) >= b^n + 1 > b^n$

Therefore, $f(n) >== b^n$ for all n >= 0, proving the claim with a = 1.

3.4. Show that $N(h) >= ab^h$ for all h >= 0 (for the a chosen in 3.3)

Combining 3.1 and 3.3, we get $N(h) >= f(h) >= ab^h$ for every h >= 0.

3.5. Conclude that an AVL tree with n nodes has height <= c lg n + O(1) where c = 1/lg ϕ.

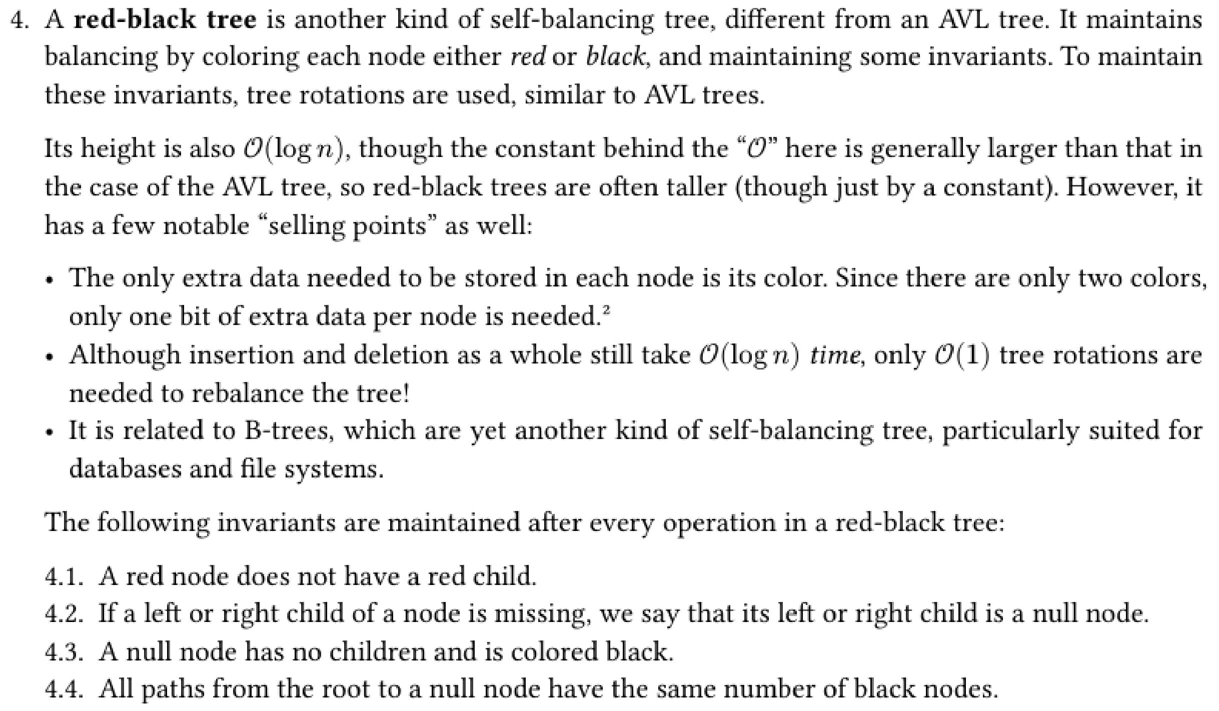
Let T be a tree with n nodes, and suppose its height is h. Note that n >= N(h) by definition of N. Using 3.4, we have

$$n >= ab^h = \phi^h$$

Taking logarithms, we get lg n >= h lg ϕ, implying

$$h <= 1/\lg\phi * \lg n,$$

which is what we wanted to prove.



4. A **red-black tree** is another kind of self-balancing tree, different from an AVL tree. It maintains balancing by coloring each node either *red* or *black*, and maintaining some invariants. To maintain these invariants, tree rotations are used, similar to AVL trees.

   Its height is also $\mathcal{O}(\log n)$, though the constant behind the "$\mathcal{O}$" here is generally larger than that in the case of the AVL tree, so red-black trees are often taller (though just by a constant). However, it has a few notable "selling points" as well:

   - The only extra data needed to be stored in each node is its color. Since there are only two colors, only one bit of extra data per node is needed.[2]
   - Although insertion and deletion as a whole still take $\mathcal{O}(\log n)$ *time*, only $\mathcal{O}(1)$ tree rotations are needed to rebalance the tree!
   - It is related to B-trees, which are yet another kind of self-balancing tree, particularly suited for databases and file systems.

   The following invariants are maintained after every operation in a red-black tree:

   4.1. A red node does not have a red child.
   4.2. If a left or right child of a node is missing, we say that its left or right child is a null node.
   4.3. A null node has no children and is colored black.
   4.4. All paths from the root to a null node have the same number of black nodes.

4. Prove that any tree with n nodes satisfying the red-black tree properties above has height <= 2 lg n + O(1).

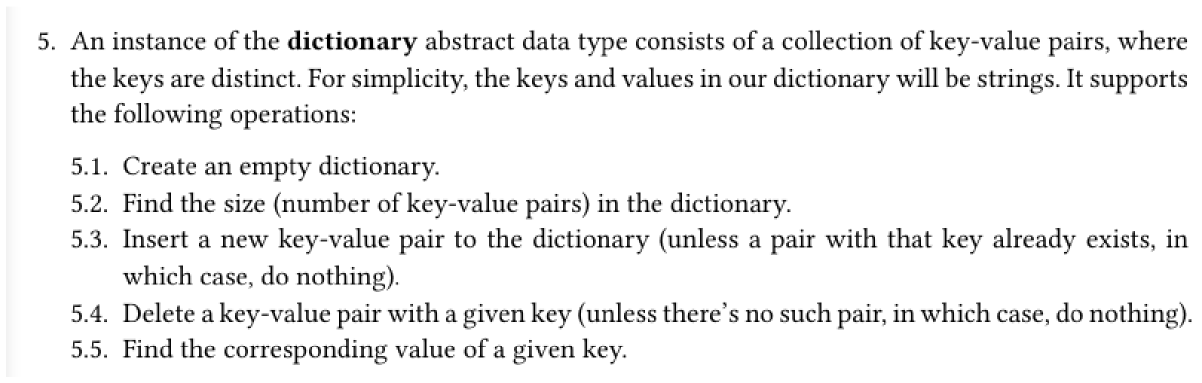
Let c be the black-height of the tree, the number of black nodes on any path from root to null.

By induction, any red-black tree with black-height c has at least $2^c - 1$ nodes >= $2(2^{c-1} - 1) + 1 = 2^c - 1$.

If root is red, then both children are black with black-height c, with even more nodes. Thus, n >= $2^c - 1$. On any root-to-null path, no two reds are consecutive, so at least half the nodes are black (rounded up).

Therefore, h <= 2c.

Combining h <= 2c <= $2\log_2(n+1)$ = 2lgn + O(1). Hence, height <= 2lgn + O(1).



5. An instance of the **dictionary** abstract data type consists of a collection of key-value pairs, where the keys are distinct. For simplicity, the keys and values in our dictionary will be strings. It supports the following operations:

   5.1. Create an empty dictionary.
   5.2. Find the size (number of key-value pairs) in the dictionary.
   5.3. Insert a new key-value pair to the dictionary (unless a pair with that key already exists, in which case, do nothing).
   5.4. Delete a key-value pair with a given key (unless there's no such pair, in which case, do nothing).
   5.5. Find the corresponding value of a given key.

5. Explain how to use AVL trees to implement a dictionary such that the first two operations take O(1) time, and the last three operations take O(log n) time.

For this data structure, we store key-value pairs in an AVL tree, ordered by key. Each node contains: key, value, left, right, height. We maintain a separate size attribute, tracking the total number of nodes.

5.1. To create an empty dictionary, we initialize root = null and size = 0. This takes O(1).
5.2. Provided we've maintained the size correctly throughout the operations, all we need to do is return the size attribute. This takes O(1).
5.3. To insert a key-value pair into the AVL tree:

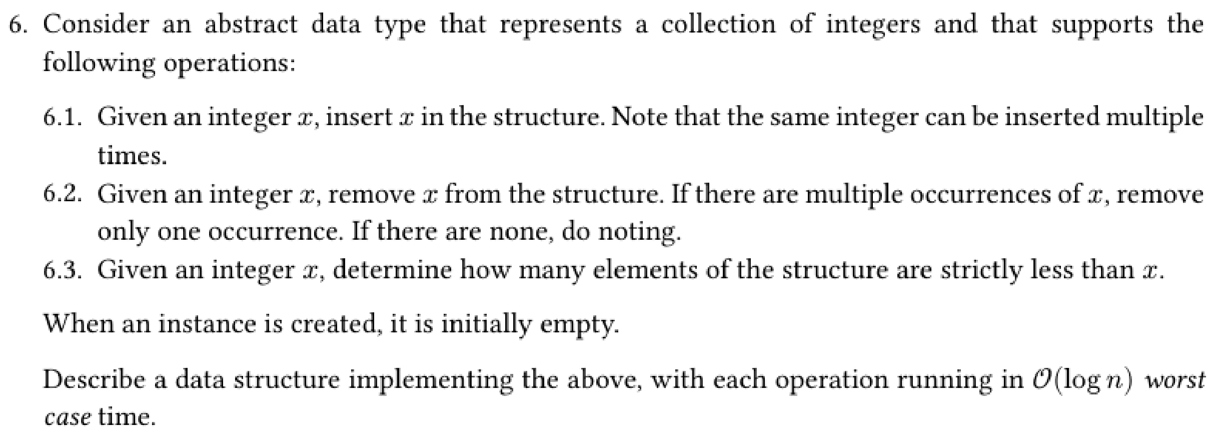
- Perform standard AVL insert using key for ordering.
- If the key exists, update its value, return False.
- If the key does not exist, create a new node, increment size by 1, and return True.
- After insertion, perform rebalancing along the path.
- This takes O(log n).

5.4. To delete a key-value pair with a given key:

- Perform standard AVL delete using key.
- If key found: remove node, decrement size by 1, return True.
- If key not found: return False
- After deletion, perform rebalancing along the path.
- This takes O(log n).

5.5. To find a value given a key:
- We perform standard BST search using key.
- If found: return associated value.
- If not, return null or identity value.
- This takes O(log n).

AVL property ensures height O(log n), so all tree operations run in O(log n). The size variable is updated on every insertion/deletion, so it is maintained correctly.



6. Consider an abstract data type that represents a collection of integers and that supports the following operations:

6.1. Given an integer x, insert x in the structure. Note that the same integer can be inserted multiple times.
6.2. Given an integer x, remove x from the structure. If there are multiple occurrences of x, remove only one occurrence. If there are none, do noting.
6.3. Given an integer x, determine how many elements of the structure are strictly less than x.

When an instance is created, it is initially empty.

Describe a data structure implementing the above, with each operation running in $\mathcal{O}(\log n)$ *worst case* time.

6. Describe a data structure implementing the above, with each operation running in O(log n) worst case time.

We implement an AVL tree, augmented as follows:
- Each node stores: value, count (multiplicity), left, right, height, size (total nodes in subtree)
- Multiple occurrences of same value stored in a single node with count > 1.

For node x:

size(x) = size(left) + size(right) + count(x)
height(x) = max(height(left), height(right)) + 1

We update the parameters above during rotations/rebalancing in O(1).

6.1. Inserting x:
- Search for node with value x
- If found: increment count by 1
- If not found: create new node with count = 1
- Update size along path, rebalance if needed.
- This takes O(log n).

6.2. Removing one occurrence of x:

- Search for node with value x
- If not found: do nothing
- If found and count > 1: decrement count by 1
- If found and count = 1 delete node, using the standard AVL deletion.
- Update size along path, rebalance if needed.
- This takes O(log n).

6.3. Count elements strictly lesser than x:

- We use the function below:

```
function count_less(node, x):
    if node is null: return 0
    if x <= node.value:
        return count_less(node.left, x)
    else: // x > node.value
        return size(node.left) + node.count + count_less(node.right, x)
```

- Running it from the root, and traversing at most one path.
- This takes O(log n).

We now discuss the correctness of this implementation. AVL property guarantees height O(log n). The size augmentation enables O(log n) order statistics. All operations update size/height correctly during rotations.