Deleting from an AVL Tree

To delete from an AVL tree, follow the same procedure as removal from a binary search tree. Then, starting at the node that was removed, move up to the root, recalculating heights if necessary. For each node $z$, if it is unbalanced, rotate the subtree rooted at that node:

1. Let $z$ be (a pointer to) the unbalanced node we found.
2. Let $y$ be the child of $z$ with greater height (hint: this is never an ancestor of the node you deleted. Why?). Why are ties impossible?
3. Let $x$ be the child of $y$ with greater height. In the event of a tie, choose $x$ so that we’ll have a single rotation instead of a double rotation

Then perform the same rotation you would do for that formation on insertion. Unlike insertion, however, this only fixes the problem locally – it might be unbalanced higher up. You need to continue this until the tree is balanced globally.

**Question 1.** Starting from the AVL tree above, what does the tree look like if we remove 32?

**Question 2.** What if we had deleted 17 instead?

**Question 3.** What if we had deleted 78 and 88 (in either order) instead?

**Question 4.** Starting with an empty AVL tree, do the following operations, in sequence:
INSERT: 1, 2, 3, 12, 9, 13, 7, 4, 6, 5, 8
DELETE: 4, 1
INSERT: 1, 14, 11
DELETE: 3, 13, 12, 11, 14, 2, 3, 7, 8, 9

Tries
Suppose we have a binary search tree with \( n \) nodes, where each key is a string with \( k \) characters.

**Question 5.** For large \( k \), would it be accurate to say that find takes \( O(\log n) \) time? What would be a more accurate runtime analysis?

Trie, from retrieval are intended specifically for string keys which are very long.

This significantly cuts down on comparison on each level, as you’re only looking at one character. Eventually you’ll look at the whole string, but only once rather than \( \log n \) times.

You can imagine there being a lot of wasted space in a Trie, such as when there are only two strings, each of 100 characters. You can improve on this by making a Compressed Trie.

Each node in a compressed trie has at least one of the following properties:

- It is the root node.
- It is a word node.
- It has at least two children.

**Question 6.** Construct a compressed trie with the following words: ten, tent, then, tense, tens, tenth

**Network Routing:** Incoming packets have a destination IP address, such as 132.125.73.60. Quickly determine the output port to forward the request through.

The IP address is actually written in binary, so the above IP address is 10000100.01111101.01001001.00111100

**Question 7.** What would be a good data structure to solve the Network Routing problem?

If you expect your trie to be dense, consisting of strings consisting of the letters a-z, then the following node structure would be appropriate:

```cpp
template <typename V>
struct TrieNode {
    V* value;
    TrieNode<V>* children[26];
};
```

Since the trie is dense, you don’t expect to take much advantage of a compressed trie, and you don’t expect to waste much space. This allows you to traverse each node in constant time.

If you expect the Trie to be sparse, just use a linked list of TrieNodes as the children.

Alternatively, allocate a dynamically-sized array to store the children, allowing for binary search and no wasted space. The cost: when you insert a new value, it will take some adjustment. Choose your implementation based on what operations you expect to be done!