Readings

- Lecture Notes Chapter 24: Tries
- Lecture Notes Chapter 25: Balanced Binary Search Trees

Problems

Problem 1

Given a set of $N$ strings, how can we find the longest common prefix between any two strings? Analyze the running time of your algorithm and justify its correctness.

Solution

Initialize an integer $\text{count}$ to 0 that will keep track of the longest prefix seen so far, and initialize a string to $\epsilon$ to keep track of the return value.

Add all of the strings into a trie, with one modification. For each string inserted, track the maximum depth into the trie before any new nodes have been initialized (meaning some other string has already been initialized along this path). If the depth is greater than $\text{count}$, update $\text{count}$ and store the relevant sub-string.

**Running time and correctness:** This will take no more than (and hopefully much less than) $O(m)$ time, where $m$ is the total length of the strings inserted. (The construction time of a trie.) Correctness follows from the lemma that two strings $w_1$ and $w_2$ share a common prefix if the insertion of the second, $w_2$, involves walking a path from the root of the trie to a node $v$, such that the terminal node of $w_1$ is a descendant of $v$. This, in turn, follows from there being a unique path from the root of a trie given any string $w$, the length of the path being equal to the length of the string.

Problem 2

Given some string $S$, how can we find the longest repeated substring? What about if we want the longest that is repeated $k$ times?

Solution

First we build a suffix trie from the string, taking $O(|S|^2)$ (or just $O(|S|)$ using Ukkonen’s Algorithm). Now, our problem becomes the same as in problem 1: we want to find the longest repeated prefix. To see why this is true, call the longest repeated subsequence of $S$, $s = \{s_1, s_2, ..., s_l\}$. Because this sequence is repeated, there must be two suffixes of the form: $\{s_1, s_2, ..., s_1, ..., S_{\text{end}}\}$, where $S_{\text{end}}$ is the last character of $S$. These two suffixes will share the same prefix, and their shared prefix is exactly the longest repeated substring.

So, we can just repeat the algorithm given for problem 1, which will find us the longest repeated substring in the suffix trie. Note that this is equivalent to finding the deepest node which has at least 2 children. For the longest repeated substring which occurs $k$ times, we would want to use a similar procedure to the auto-complete program of keeping track of sizes. Our problem is then discovering the deepest node which has size $\geq k$. By the same logic as before, the prefix of this deepest node must be present in the original string $\geq k$ times. To actually find this node, we can keep track during construction or just perform BFS after construction, both of which will not increase our runtime past the $O(|S|^2)$ (or $O(|S|)$ using Ukkonen’s Algorithm) we already have for construction.
Problem 3
Design an algorithm to decide if a given binary tree is a valid binary search tree.

Solution
This problem is a bit trickier than it may at first seem. The first thing that may come to mind is to simply visit every node in the tree and check that the current node’s key is greater than the key of its left child and less than the key of its right child. However, this algorithm does not work on the following tree:

```
3
/ \
2  5
/ \
1  4
```

We can see that every node is in the correct order in relation to its immediate children, but 4 should not be in the left subtree of 3. So we will have to refine our algorithm a bit.

We want to use a similar algorithm, but we need a way of knowing the valid values that can appear in the current subtree of the BST. So we can define a recursive algorithm that begins at the root of the tree and works down. Each recursive call takes in a node to visit, and a minimum and maximum value that define the range of values that can appear in the subtree rooted at that node. We begin at the root $x$ and set $min = -\infty$ and $max = \infty$. Now we check the keys of the children of $x$ to make sure that they are in the correct order in comparison to $x$ and ensure that the keys are in the range strictly between $min$ and $max$. If these conditions are not met, then return false. Otherwise, recursively call this algorithm on the left child of $x$ with the same $min$ value but with $max = x.key$. We also recursively call the algorithm on the right child of $x$ with the same $max$ value but with $min = x.key$. If this procedure visits the entire tree without returning false, then the binary tree is indeed a BST.

Another solution, which is particularly elegant, is to simply perform an inorder traversal on the given tree and check that the elements are visited in sorted order. If they are not in sorted order, then the tree is not a BST. Otherwise, it is a BST. This gives us an easy $O(n)$ time algorithm to check if a binary tree is a binary search tree.

Problem 4
Perform a series of rotations to make 5 the root of the following unbalanced BST:

```
10
/ \
7  15
/ \
5  3
/ \
2  4
\  
1
```

Solution
First, rotate the root right
Then, rotate the root right again

\[
\begin{array}{c}
5 \\
/ \ \\
3 \\
/ \\
2 \\
/ \\
1
\end{array}
\]

Then, rotate the root right again

\[
\begin{array}{c}
5 \\
/ \\
3 \\
/ \\
2 \\
/ \\
1
\end{array}
\]

**Problem 5**

Draw the resulting AVL tree after inserting the following:

\{3, 4, 5, 8, 12, 17, 1\}

Draw the resulting tree after deleting 12.

**Solution**

\[
\begin{array}{c}
8 \\
/ \\
4 \\
/ \\
3 \\
/ \\
1
\end{array}
\]

This is the tree after inserting all of the keys. Note that we perform a left rotation after inserting 5, 12, and 17.

\[
\begin{array}{c}
4 \\
/ \\
3 \\
/ \\
1
\end{array}
\]

After deleting 12, the heights of its children are more than 1 apart, so we must perform a right rotation. This is the resulting tree.

**Additional Problems**

**Problem 6**

Show that any arbitrary \(n\)-node binary search tree can be transformed into any other arbitrary \(n\)-node binary search tree using \(O(n)\) rotations. Assume that both trees contain the same \(n\) distinct elements. (Hint: First show that at most \(n - 1\) right rotations suffice to transform the tree into a right-going chain, which is a binary search tree where every internal node only has a right child.)
Solution

Suppose we want to transform the tree $T_1$ into $T_2$. Our first objective is to transform $T_1$ into a right-going chain. We can do this by strategically applying right rotations to $T_1$. Consider the chain of nodes in $T_1$ that results from beginning at the root and following right child pointers until you reach a null element. We will call this our partial right-going chain. Note that the partial chain must contain at least 1 element because the root itself must be part of the chain. We will construct the full right-going chain by applying right rotations in $T_1$ that add a node to our partial chain each time (think about how this works). Note that each rotation will add 1 to the length of the partial chain. Since the partial chain starts with length at least 1 and there are $n$ elements in the tree, we will need at most $n - 1$ right rotations to construct the full right-going chain.

Now, by the same reasoning, we can also convert $T_2$ to the same right-going chain by applying at most $n - 1$ right rotations. But note that any right rotation can be reversed by a corresponding left rotation. In other words, the sequence of operations used to convert $T_2$ to a right-going chain can be inverted to convert a right-going chain to $T_2$. Thus, we can convert $T_1$ to $T_2$ by converting $T_1$ to a right-going chain and then applying the sequence of rotations that convert the right-going chain to $T_2$. This will require at most $2(n - 1) = O(n)$ rotations.