Consider the following data definition & function:

\[ \text{A BST is either} \]
\[ \text{false, or} \]
\[ \text{(make-node num BST BST)} \]
\[ \text{where all els in left are} \leq \text{keys} \leq \text{all els in right} \]

\[
\begin{align*}
\text{define struct node (key left right)} \\
\text{(define in? num BST -> bool)} \\
\text{[false? abst]} \\
\text{[true? abst]} \\
\text{[]}
\end{align*}
\]

Consider an arbitrary pair of arguments. By inspection, we can see that \( \text{in?} \) performs fewer than some constant number of steps unless it recurs. Therefore, we focus on the recursive case(s).

Let \( T(k) \) be an upper-bound on the time taken by \( \text{in?} \) on BSTs of size \( k \), where the size of a BST is the number of keys in it (recursively). Then

\[
\begin{align*}
T(0) &= c \\
T(k) &\leq c' + \max \\
&\left\{ T(\text{left}) \quad \text{if first any} < \text{key} \right\} \\
&\left\{ T(\text{right}) \quad \text{if first any} > \text{key} \quad \text{for some} \ c' < k \right\}
\end{align*}
\]

where left & right are the sizes of the corresponding sub-trees. But how large might they be? The BST data definition only says something about the values in the left & right branches, not how many such values there might be. At worst, the sub-tree chosen might have \( k-1 \) elements (excluding the key that \( k \)). Thus

\[
T(k) \leq c' + T(k-1) \quad \text{for} \ k > 1
\]

leading to a linear search algorithm.

We could do significantly better by guaranteeing that the two sub-trees were of the same size (or at most off by one); then, \( \max (T(\text{left}), T(\text{right})) = \lceil k/2 \rceil \), so

\[
T(k) \leq c' + T(\lceil k/2 \rceil) \quad \text{for} \ k \geq 1
\]

leading to a logarithmic search algorithm.

This is desired - but how can we ensure the two sub-trees are always (almost) the same size? Stay tuned!