Analysis: BST

Consider the following data-definition & function:

```
; A BST is either
; - false, or
; - (make-node num BST BST)
(define-struct node (key left right)
where all elts in left are ≤ key ≤ all elts in right
```

```
; in?: num BST -> bool
(define (in? n abst)
  (cond
    [(false? abst) false]
    [(node? abst)
     (cond
       [(= n (bst-key abst)) true]
       [(< n (bst-key abst))
        (in? n (bst-left abst))]
       [(> n (bst-key abst))
        (in? n (bst-right abst))])]))
```

Consider an arbitrary pair of arguments. By inspection, we can see that 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 in? on BSTs of size k, where the size of a BST is the number of keys in it (recursively). Then

$$T(0) \leq c \quad \text{for some } c$$

$$T(k) \leq c' + \max \begin{cases} T(\text{left}) & \text{if first arg} < \text{key} \\ T(\text{right}) & \text{if first arg} > \text{key} \end{cases} \quad \text{for some } c' \text{ \& } k \geq 1$$

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 is). Thus

$$T(k) \leq c' + T(k-1) \quad \text{for } k \geq 1$$

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

$$T(k) \leq c' + T(\lfloor k/2 \rfloor) \quad \text{for } k \geq 1$$

leading to a logarithmic search algorithm.

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