Self-Balancing BSTs

EECS 214

November 6, 2015
Take-aways

- What is the $BST$ property?
- How do $BST$ lookup, insertion, and deletion work?
- Why does balance matter?
A basic binary tree

A binary tree, describing structure but not content:

; An [BinTree X] is one of:
; -- (leaf)
; -- (branch [BinTree X] X [BinTree X])
(define-struct leaf [])
(define-struct branch [left element right])
The \textit{BST property}

To be a BST, a binary search tree needs to be ordered:

; [BST Integer] \to Boolean
(define (int-bst? tree)
  (int-bst-within? -INF.0 tree +INF.0))

; Number IntBST Number \to Boolean
(define (within? min tree max)
  (or
   (leaf? tree)
   (and
    (< min (element tree) max)
    (within? min (left tree) (element tree))
    (within? (element tree) (right tree) max))))
Two helpful definitions

; An [Ord X] is a function [X X -> Boolean]
; Invariant: must be a total order on Xs
Two helpful definitions

; An [Ord X] is a function [X X -> Boolean]
; Invariant: must be a total order on Xs

; A [Maybe X] is one of:
; -- X
; -- #false
Binary search

The BST property enables binary search:

\[ \text{Ord } X \times [\text{BST } X] \rightarrow [\text{Maybe } X] \]

\[
\text{(define } \text{(lookup } \text{lt? } \text{needle } \text{haystack)} \\
\text{(cond} \\
\text{[(leaf? haystack) #false]} \\
\text{[(lt? needle (element haystack))} \\
\text{ (lookup } \text{lt? } \text{needle (left haystack))]} \\
\text{[(lt? (element haystack) needle) } \\
\text{ (lookup } \text{lt? } \text{needle (right haystack))]} \\
\text{[else} \\
\text{ (element haystack)]})\]
\]
Insertion is similar

; [Ord X] X [BST X] -> [BST X]
define (insert lt? new tree)
  (cond
    [(leaf? tree) (make-branch LEAF new LEAF)]
    [(lt? new (element tree))
      (make-branch (insert lt? new (left tree))
        (element tree)
        (right tree))]
    [(lt? (element tree) new)
      (make-branch (left tree)
        (element tree)
        (insert lt? new (right tree)))]
    [else
      (make-branch (left tree) new (right tree))])}
Binary search is $O(\log n)$, right?

Start with the empty tree.
Binary search is $\mathcal{O}(\log n)$, right?

Start with the empty tree. Insert 1.
Binary search is $O(\log n)$, right?

Start with the empty tree. Insert 1. Insert 2.
Binary search is $O(\log n)$, right?

Start with the empty tree. Insert 1. Insert 2. Insert 3.
Binary search is $O(\log n)$, right?

Start with the empty tree. Insert 1. Insert 2. Insert 3. Insert 4.
Binary search is $\mathcal{O}(\log n)$, right?

Binary search is $O(\log n)$, right?

Binary search is $O(\log n)$, right?

Binary search is $O(\log n)$, right?

We need some balance!

There are a variety of self-balancing trees:

- Red-black trees
- Splay trees
- 2-3 trees
- 2-4 trees
- B trees
- and so on...
The AVL property

An AVL tree is *height balanced*: For every node, the heights of its left and right subtrees can differ by at most 1.
The AVL property

An AVL tree is height balanced: For every node, the heights of its left and right subtrees can differ by at most 1

We keep the balance in each node:

; An [AVLTree X] is one of:
; -- (leaf)
; -- (branch Balance [AVLTree X] X [AVLTree X])
; where Balance is the integer interval [-1, 1]
;
; Invariant: for all nodes n,
; (= (balance n)
;  (- (height (right n)) (height (left n))))
(define-struct leaf [])
(define-struct branch [balance left element right])
Big theme!

Local properties induce global properties.
Take-aways

- What is the BST property?
- How do BST lookup, insertion, and deletion work?
- Why does balance matter?