CS135 Tutorial 07
Balanced Binary Trees

In class we’ve commented that sometimes there are advantages to a “balanced” binary tree – especially when searching a BST.

In A06 you wrote a function, `full?`, a function to determine whether every node had either 0 or 2 children. This is one measure of “balance”.

Today:
- A stricter definition of balance
- Build a balanced tree
- Determine whether an existing tree is balanced
- Search a BST for a range of values
Balanced Binary Trees

- There are several definitions of “balanced”. Here’s one:

- A binary tree is balanced if:
  - The number of nodes in the left and the right subtrees differ by at most 1
  - Both subtrees are also balanced.
  - An empty tree is balanced.
Balanced BST Data Definition

(define-struct node (key left right))
;; A Node is a (make-node Nat BalBST BalBST)
;; requires: all keys in left < key
;; all keys in right > key
;; |(# nodes in left) – (# nodes in right)| <= 1

;; A balanced binary tree (BalBST) is one of:
;; * empty
;; * Node
Building balanced binary search trees

Given a sorted list of number, build a balanced binary search tree.

```
(define-struct node (key left right))

;; (build-bal-bst slon) builds a balanced binary search tree from slon.
;; build-bal-bst: (listof Num) -> BalBST
;; requires: slon is sorted in increasing order
(define (build-bal-bst slon) ...)

(check-expect (build-bal-bst empty) empty)
(check-expect (build-bal-bst (list 1)) (make-node 1 empty empty))
(check-expect (build-bal-bst (list 1 2 3 4 5 6))
    (make-node 4
        (make-node 2 (make-node 1 empty empty) (make-node 3 empty empty))
        (make-node 6 (make-node 5 empty empty) empty)))
```
Required helper functions

;; (nth-elem lst n) produces the nth element in lst (counting from 0).
;; nth-elem: (listof X) Nat -> X
(define (nth-elem lon n)
    (cond [(zero? n) (first lon)]
          [else (nth-elem (rest lon) (sub1 n))]))

;; (take lon n) produces a list from the first n elements f lst.
;; take: (listof X) Nat -> (listof X)
(define (take lon n)
    (cond [(zero? n) empty]
          [else (cons (first lon) (take (rest lon) (sub1 n)))]))

;; (drop lon n) produces a list from the elements after the first n+1 elements
(define (drop lon n)
    (cond [(zero? n) (rest lon)]
          [else (drop (rest lon) (sub1 n))])))
Required helper functions

(define lst (list 0 1 2 3))

(check-expect (nth-elem lst 0) 0)
(check-expect (nth-elem lst 1) 1)
(check-expect (nth-elem lst 3) 3)
(check-expect (take lst 0) empty)
(check-expect (take lst 1) (list 0))
(check-expect (drop lst 0) (list 1 2 3))
(check-expect (drop lst 1) (list 2 3))
(check-expect (drop lst 3) empty)

(check-expect (append (take lst 0) (list (nth-elem lst 0)) (drop lst 0)) lst)
(check-expect (append (take lst 1) (list (nth-elem lst 1)) (drop lst 1)) lst)
(check-expect (append (take lst 2) (list (nth-elem lst 2)) (drop lst 2)) lst)
(check-expect (append (take lst 3) (list (nth-elem lst 3)) (drop lst 3)) lst)
(balanced? bt)

- A binary tree is balanced if:
  - The number of nodes in the left and the right subtrees differ by at most 1
  - Both subtrees are also balanced.
  - An empty tree is balanced.

We are not checking if the BST ordering property holds.

<table>
<thead>
<tr>
<th>(check-expect (balanced? empty) ...)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(check-expect (balanced? (make-node 10 empty empty)) ...)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
(check-expect
(balanced?
  (make-node 10
    (make-node 5 empty (make-node 8 empty empty))
    empty)
  empty))
...

(check-expect
(balanced?
  (make-node 10
    (make-node 5 empty (make-node 8 empty empty))
    (make-node 20 empty empty)))
...

(check-expect
(balanced?
  (make-node 10
    (make-node 5 empty (make-node 8 empty empty))
    (make-node 20 (make-node 14 empty empty)
    (make-node 25 empty empty))))
...
Count-range

Write a function, `(count-range bst lo hi)`. It produces the number of keys in \([lo, hi]\) – between \(lo\) and \(hi\), inclusive.

For the BST shown:

(check-expect (count-range bst 0 50) 7)
(check-expect (count-range bst 11 25) 3)
(check-expect (count-range bst 8 8) 1)
(check-expect (count-range bst 11 13) 0)

Use the ordering property of BSTs.
Count-range – without ordering property

;; (count-range bst lo hi) counts the number of nodes with keys between
;; lo and hi, inclusive.
;; count-range: BST Nat Nat -> Nat
;; requires: lo <= hi
(define (count-range bst lo hi)
  (cond [(empty? bst) 0]
        [else (+ (cond [(< (node-key bst) lo) 0]
                    [(> (node-key bst) hi) 0]
                    [else 1])
             (count-range (node-left bst) lo hi)
             (count-range (node-right bst) lo hi)])))