Goals of this tutorial
You should be able to...

- Traverse Binary Trees (BT) and Binary Search Trees (BST).
- Write a Template for a BT/BST.

Announcements
- Midterm 2 is on Monday, November 4 at 7:00 PM. Look to “odyssey.uwaterloo.ca” for seating arrangements.
- Midterm Review Session times and locations are posted on Piazza.
- There is NO assignment due next week.
- The times and locations of office hours are posted on the “Office and Consulting Hours” page of the course website. Please email us at cs135@uwaterloo.ca to set up an appointment outside of these hours.
**Problem 1: Accumulate Primes**

Given a natural number \( n \), produce a list of all prime numbers in decreasing order down to 2.

Use the following code in your solution:

\[
\text{(has-factor? n lon) Determine whether the list lon contains at least one factor of n.}
\]

\[
\text{has-factor?: Nat (listof Nat) } \rightarrow \text{ Bool}
\]

\[
\text{(define (has-factor? n lon)}
\]

\[
\text{ (cond [[(empty? lon) false]

[else (or (= 0 (remainder n (first lon)))

(has-factor? n (rest lon))))]})
\]

**Problem 1: Helper Design Recipe**

\[
\text{(primes-up-to/acc i n primes-so-far) produces}
\]

\[
\text{a list of all primes less than or equal to n,}
\]

\[
\text{given the list of primes-so-far less than i}
\]

\[
\text{primes-up-to/acc: Nat Nat (listof Nat) } \rightarrow \text{ (listof Nat)}
\]

\[
\text{requires: primes-so-far is the list of primes < i}
\]

\[
\text{Example:}
\]

\[
\text{(check-expect (primes-up-to/acc 2 7 empty) (list 7 5 3 2))}
\]

\[
\text{(define (primes-up-to/acc i n primes-so-far)}
\]

\[
\text{...)}
\]

**Problem 2: Binary Trees**

Using the data definition provided write a template for a binary tree.

\[
\text{(define-struct bt (key left right))}
\]

\[
\text{A binary tree (BT) is any of:}
\]

\[
\text{* empty}
\]

\[
\text{* (make-bt Nat BT BT)}
\]
**Problem 2: Replace**
Using template for a bt write one function called `replace-bt` that consumes a binary tree and two numbers called `old-key` and `new-key`. It will replace every occurrence of `old-key` with a new bt holding the `new-key` value. If the `old-key` is not found in the binary tree, the original tree is produced.

---

**Problem 2: Design Recipe**

```scheme
;; (replace-bt tree old-key new-key) Consumes a tree, an
;; old-key, and a new-key producing a new tree with all
;; containing the old-key replaced with the new-key.
;; replace-bt: BT Nat Nat → BT
```

---

**Problem 2: Design Recipe**

```scheme
;; Example:
(check-expect (replace-bt (make-bt 1 (make-bt 2 empty empty) (make-bt 3 empty empty)) 3 0)
  (make-bt 1 (make-bt 2 empty empty) (make-bt 0 empty empty))
)
```

```scheme
;; Test:
(check-expect (replace-bt (make-bt 1 empty empty) 2 0)
  (make-bt 1 empty empty))
```
Review: Binary Search Trees

(define-struct bst (key left right))

;; A BST is one of:
;; * empty
;; * (make-bst Nat BST BST)
;; requires:
;; key > all left BST keys
;; key < all right BST keys

Clicker Question 1: Identify the BST

A) 3
   2
   1
B) 3
   2
   1
   3
C) 2
   3

D) 4
   2
   3
   5
E) 3
   2
   4
   4

Problem 3: Tree Range

The full range of a tree is defined by the difference between its smallest and its largest keys. Define a function that calculates the differences between the minimum and maximum keys found in a non-empty binary tree.

Your first step should be to write a helper function that finds the smallest key in a BST, call this function tree-min.