Announcements:

- PS3 due Thursday 11:59PM
    - Testing will start sometime Friday morning
    - Return is likely to be delayed due to Fall break
- Quiz #3 in class Tue Oct 18
- Prelim #1 comments
    - Good: induction
    - Bad: Dijkstra
    - Ugly: user defined types (!)

Binary search trees

A binary tree is easy to define inductively in OCaml. We will use the following definition which represents a node as a triple of a value and two children, and which explicitly represents leaf nodes.

```
type 'a tree = TNode of 'a * 'a tree * 'a tree | TLeaf
```

A **binary search tree** is a binary tree with the following representation invariant: For any node *n*, every node in the left subtree of *n* has a value less than that of *n*, and every node in the right subtree of *n* has a value more than that of *n*.

Note that this is a **rep invariant**! The type system doesn't enforce this but you need it to be true.

Given such a tree, how do you perform a lookup operation?

Start from the root, and at every node, if the value of the node is what you are looking for, you are done; otherwise, recursively look up in the left or right subtree depending on the value stored at the node.

In code:

```
let rec contains x = function
    TLeaf -> false
  | TNode (y, l, r) ->
      if x=y then true else if x < y then contains x l else contains x r
```

Note the use of the keyword **`function`** so that the variable used in the pattern matching need not be named. This is equivalent to (unneccessarily) naming a variable and then using **`match`**:

```
let rec contains x t =
  match t with
      TLeaf -> false
    | TNode (y, l, r) ->
        if x=y then true else if x < y then contains x l else contains x r
```

Adding an element is similar: you perform a lookup until you find the empty node that should contain the value.

This is a nondestructive update, so as the recursion completes, a new tree is constructed that is just like the old one except that it has a new node (if needed):

```
let rec add x = function
    TLeaf -> TNode (x, TLeaf, TLeaf) (* When get to leaf, put new node
there *)
  | TNode (y, l, r) as t -> (* Recursively search for value *)
      if x=y then t
      else if x > y then TNode (y, l, add x r)
      else (* x < y *) TNode (y, add x l, r)
```

What is the running time of those operations? Since **`add`** is just a **`lookup`** with an extra constant-time node creation, we focus on the **`lookup`** operation. Clearly, the run time of **`lookup`** is $O(h)$, where h is the height of the tree.

What's the worst-case height of a tree? Clearly, a tree of n nodes all in a single long branch (imagine adding the numbers 1,2,3,4,5,6,7 in order into a binary search tree). So the worst-case running time of lookup is still $O(n)$ (for n the number of nodes in the tree).

What is a good shape for a tree that would allow for fast lookup?

A **perfect binary tree** has the largest number of nodes n for a given height h: $n = 2^{h+1}-1$. Therefore $h = \lg(n+1)-1 = O(\lg n)$.

```
            ^                      50
            |                  /          \
            |             25                    75
 height=3   |           /     \             /     \
   n=15     |         10       30         60       90
            |        /   \    /   \      /   \    /   \
            V       4    12  27   40    55   65  80   99
```

If a tree with n nodes is kept balanced, its height is $O(\lg n)$, which leads to a lookup operation running in time $O(\lg n)$.

How can we keep a tree balanced? It can become unbalanced during element addition or deletion. Most balanced tree schemes involve adding or deleting an element just like in a normal binary search tree, followed by some kind of tree surgery to rebalance the tree. Some examples of balanced binary search tree data structures include

- AVL (or height-balanced) trees (1962)
- 2-3 trees (1970's)
- Red-black trees (1970's)

In each of these, we ensure asymptotic complexity of O(lg n) by enforcing a stronger invariant on the data structure than just the binary search tree invariant.

- Red black trees:

- Recall that BST's work well when balanced,
  - But if you insert in the wrong order (sorted) you basically get a stupid representation of a list

- Solution: self-balancing trees
  - AVL trees, or red-black trees are most popular
- They are guaranteed to stay approximately balanced
  - Achieve this by rotations

- RB trees have the longest path from the root to any leaf is no more than twice the shortest path
- RB tree ***rep invariant***:
  - A. Nodes are red or black
  - B. Root and leaves are black
  - C. Children of red are black
  - D. Every path from a node to a leaf beneath it has the same number of black nodes (but not necessarily red)

- Notes:
  - You don't need red nodes.
    - So the shortest path from root to leaf will be purely black.
  - Because of C, the longest path will be B-R-B-R…B, which has m black nodes and m-1 red ones.

```
type color = Red | Black

type 'a rbtree = Node of color * 'a * 'a rbtree * 'a rbtree | Leaf

let rec mem x = function
    Leaf -> false
  | Node (_, y, left, right) ->
      x = y || (x < y && mem x left) || (x > y && mem x
right)
```

  - Like a BST except nodes also have a color
- Insert works like BST (find where it should go, insert a leaf)
  - But what about color?

- Three steps:
  - 1. Replace leaf with red node with two leaves underneath it
    - Both are black, so C is true

  - 2. Balance the result, i.e. ensure that C is true again
    - For example, if the parent of the leaf was red, we now have red under red and C is false

  - 3. For the root to be black

- Balancing is the tricky part.
    - We need to ensure that a red node has no red children.
    - There are 4 cases we need to handle, but we do the same thing.

```
        Bz
       /  \
     Ry    d
    /  \
  Rx    c
 /  \
a    b
```

```
    Bz
   /  \
 Rx    d
 / \
a   Ry
   /  \
  b    c
```

```
  Bx
 /  \
a    Rz
    /  \
  Ry    d
  / \
 b   c
```

```
  Bx
 /  \
a    Ry
    /  \
   b    Rz
       /  \
      c    d
```

```
      Ry
     /  \
   Bx    Bz
  / \   / \
 a   b c   d
```

```
let balance = function
    Black, z, Node (Red, y, Node (Red, x, a, b), c), d
  | Black, z, Node (Red, x, a, Node (Red, y, b, c)), d
  | Black, x, a, Node (Red, z, Node (Red, y, b, c), d)
  | Black, x, a, Node (Red, y, b, Node (Red, z, c, d)) ->
      Node (Red, y, Node (Black, x, a, b), Node (Black, z, c, d))
  | a, b, c, d ->
      Node (a, b, c, d)

let insert x s =
  let rec ins = function
      Leaf -> Node (Red, x, Leaf, Leaf)
    | Node (color, y, a, b) as s ->
     if x < y then balance (color, y, ins a, b)
     else if x > y then balance (color, y, a, ins b)
     else s
  in
    match ins s with
     Node (_, y, a, b) ->
       Node (Black, y, a, b)
      | Leaf -> (* guaranteed to be nonempty *)
       raise (Failure "RBT insert failed with ins returning leaf")
```