Announcements:

- Prelim tonight! 7:30-9:00 in Thurston 203/205
    - Handed back in section tomorrow
    - If you have a conflict you can take the exam at 5:45 but can't leave early. Please email me so we have a headcount. Show up in 4158 Upson at 5:30.

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. Should use repOK to check in debug version.

- 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 (unnecessarily) 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):

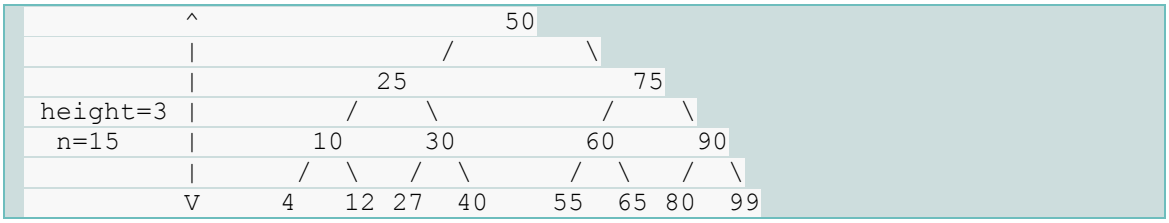
```
let rec add x = function
    TLeaf -> TNode (x, TLeaf, TLeaf) (* When get to leaf, put new node
there *)
  | TNode (y, l, r) as t -> (* Recursively search for value *)
      if x=y then t
      else if x > y then TNode (y, l, add x r)
      else (* x < y *) TNode (y, add x l, r)
```

- 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 $h = \lg(n+1)-1 = O(\lg n)$.



- 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 (this is kind of a non-operational rep invariant)

- Here are the new conditions we add to the binary search tree representation invariant:

    1. There are no two adjacent red nodes along any path.
    2. Every path from the root to a leaf has the same number of black nodes. This number is called the *black height* (BH) of the tree.

- If a tree satisfies these two conditions, it must also be the case that every subtree of the tree also satisfies the conditions. If a subtree violated either of the conditions, the whole tree would also.
- Additionally, we require that the root of the tree be colored black. This can always be enforced by simply setting its color to black; doing this does not cause any other invariants to be violated.

- Notes:
    - You don't need red nodes.
        - So the shortest path from root to leaf will be purely black.

- With these invariants, the longest possible path from the root to an empty node would alternately contain red and black nodes;
    - therefore it is at most twice as long as the shortest possible path, which only contains black nodes.
    - If n is the number of nodes in the tree, the longest path cannot have a length greater than twice the length of the paths in a perfect binary tree: $2 \log n$, which is $O(\log n)$.
    - Therefore, the tree has height $O(\log n)$ and the operations are all asymptotically logarithmic in the number of nodes.

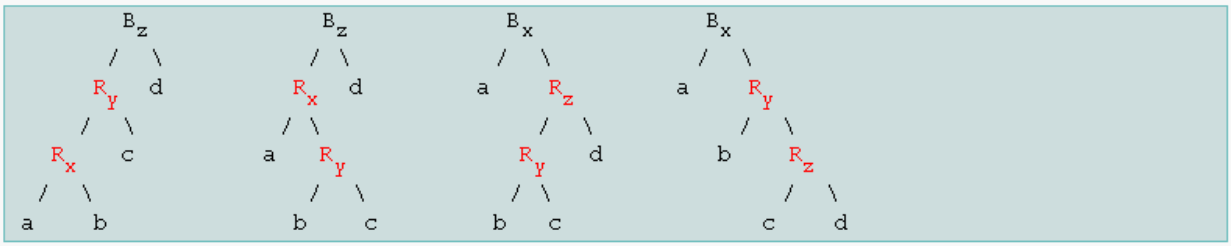
- Another way to see this is to think about just the black nodes in the tree.
- Suppose we snip all the red nodes out of the trees and reconnect each black node to its closest black ancestor.
- Then we have a tree whose leaves are all at depth BH, and whose branching factor ranges between 2 and 4. Such a tree must contain at least $\Omega(2^{BH})$ nodes, and so must the whole tree when we add the red nodes back in. If n is $\Omega(2^{BH})$, then black height BH is $O(\log n)$. But invariant 1 says that the longest path is at most $h = 2BH$. So h is $O(\log n)$ too.

- How do we check for membership in red-black trees? Exactly the same way as for general binary trees.

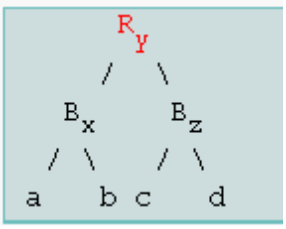
```
type color = Red | Black

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)
```

- More interesting is the insert operation.
- As with standard binary trees, we add a node by replacing the leaf found by the search procedure.
    - We also color the new node red to ensure that invariant 2 is preserved.
    - However, this may destroy invariant 1 by producing two adjacent red nodes.
- In order to restore the invariant, we consider not only the new red node and its red parent, but also its (black) grandparent. The next figure shows the four possible cases that can arise.



Notice that in each of these trees, the values of the nodes in a, b, c, d must have the same relative ordering with respect to x, y, and z: a < x < b < y < c < z < d. Therefore, we can transform the tree to restore the invariant locally by replacing any of the above four cases. (This works in part because we did some clever renaming of variables.)



Be sure to compare the below with, e.g., the Wikipedia description of how RB trees work in an imperative programming language.

```
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")
```