Lecture 11: Balanced Binary Trees: Red-Black Trees

Sets and maps are important and useful abstractions. We've seen various ways to implement an abstract data type for sets and maps, since data structures that implement sets can be used to implement maps as well. Today we will look at an implementation of sets that is asymptotically efficient and useful in practice. This implementation is one of several *balanced binary tree* schemes.

Binary trees have two advantages above the asymptotically more efficient hash table: first, they support nondestructive update with the same asymptotic efficiency. Second, they store their values (or keys, in the case of a map) in order, which makes range queries and in-order iteration possible.

An important property of a search tree is that it can be used to implement an **ordered set** or ordered map easily: a set (map) that keeps its elements in sorted order. Although we will not consider such operations today, ordered sets generally provide operations for finding the minimum and maximum elements of the set, for iterating over all the elements between two elements, and for extracting (or iterating over) ordered subsets of the elements between a range.

Binary Search Trees

A binary tree is easy to define inductively in OCaml. We will use the following definition that represents an internal (nonleaf) node as a `TNode` containing a value of type `'a` and two children and a leaf as a `TLeaf`. Leaves do not contain data in this representation.

```
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 *n*'s value, and every node in the right subtree of *n* has a value greater than *n*'s value.

Given such a tree, how do you perform a lookup operation? Start from the root. At every node, if the value of the node is the value you are looking for, you are done. Otherwise, recursively search in the left or right subtree depending on whether the value you are looking for is less than or greater than the one stored at the current node, respectively. In code:

```
let rec contains x = function
    TLeaf -> false
  | TNode (y, l, r) ->
      x = y || (x < y && contains x l) || 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) ->
        x = y || (x < y && contains x l) || 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) (* If at a 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? It occurs with 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 $h = \log(n+1)-1 = O(\log n)$. (If unspecified, logs are always base 2, although it doesn't really matter because of the $O(\ )$.)

```
          ^                   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 $O(\log n)$, which leads to a lookup operation running in time $O(\log n)$.

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 $O(\log n)$ by enforcing a stronger invariant on the data structure than just the binary search tree invariant.

Red-Black Trees

Red-black trees are a fairly simple and very efficient data structure for maintaining a balanced binary tree. The idea is to strengthen the representation invariant so a tree has height logarithmic in n. To help enforce the invariant, we color each node of the tree either *red* or *black*. Where it matters, we consider the color of an empty tree to be black.

```
type color = Red | Black
type 'a rbtree =
  Node of color * 'a * 'a rbtree * 'a rbtree | Leaf
```

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.

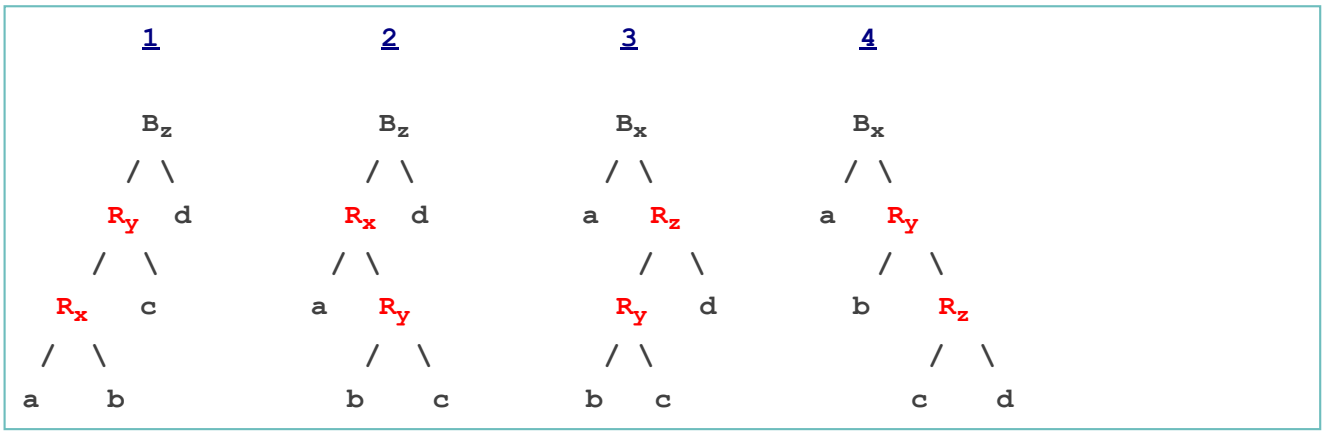
With these invariants, the longest possible path from the root to an empty node would alternately contain red and black nodes; therefore it is at most twice as long as the shortest possible path, which only contains black nodes. If n is the number of nodes in the tree, the longest path cannot have a 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 reconnecting 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.

```
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 with:

This balance function can be written simply and concisely using pattern matching, where each of the four input cases is mapped to the same output case. In addition, there is the case where the tree is left unchanged locally.

```
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)
```

This balancing transformation possibly breaks invariant 1 one level up in the tree, but it can be restored again at that level in the same way, and so on up the tree. In the worst case, the process cascades all the way up to the root, resulting in two adjacent red nodes, one of them the root. But if this happens, we can just recolor the root black, which increases the BH by one. The amount of work is $O(\log n)$. The `insert` code using `balance` is as follows:

```
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 *)
      failwith "RBT insert failed with ins returning leaf"
```

Removing elements

Removing an element from a red-black tree works analogously. We start with BST element removal and then do rebalancing. When an interior (nonleaf) node is removed, we simply splice it out if it has fewer than two nonleaf children; if it has two nonleaf children, we find the next value in the tree, which must be found inside its right child.

Balancing the trees during removal from red-black tree requires considering more cases. Deleting a black element from the tree creates the possibility that some path in the tree has too few black nodes, breaking the black-height invariant 2. The solution is to consider that path to contain a "doubly-black" node. A series of tree rotations can then eliminate the doubly-black node by propagating the "blackness" up until a red node can be converted to a black node, or until the root is reached and it can be changed from doubly-black to black without breaking the invariant.