Advanced Programming Handout 6

Purely Functional Data Structures:

A Case Study in Functional Programming

Persistent vs. Ephemeral

- An *ephemeral* data structure is one for which only one version is available at a time: after an update operation, the structure as it existed before the update is lost.
- A *persistent* structure is one where multiple versions are simultaneously accessible: after an update, both old and new versions can be used.

Persistent vs. Ephemeral

- In imperative languages, most data structures are ephemeral.
  - It is generally accepted that persistent variants, when they are possible at all, will be more complex to code and asymptotically less efficient.
- In purely functional languages like Haskell, *all* data structures are persistent!
  - Since there is no assignment, there is no way to destroy old information. When we are done with it, we just drop it on the floor and let the garbage collector take care of it.
- So one might worry that efficient data structures might be hard or even impossible to code in Haskell.

Enter Okasaki

- Interestingly, it turns out that many common data structures have purely functional variants that are easy to understand and have exacly the same asymptotic efficiency as their imperative, ephemeral variants.
- These structures have been explored in an influential book, *Purely Functional Data Structures,* by Chris Okasaki, and in a long series of research papers by Okasaki and others.

Simple Example

- To get started, let’s take a quite simple trick that was known to functional programmers long before Okasaki...

Functional Queues

- A *queue* (of values of type `a`) is a structure supporting the following operations:

```
enqueue :: a -> Queue a -> Queue a
dequeue :: Queue a -> (a, Queue a)
```

- We expect that each operation should run in O(1) --- i.e., constant --- time, no matter the size of the queue.

Naive Implementation

- A queue of values of type **a** is just a list of **a**s:

  ```
  type Queue a = [a]
  ```

- To dequeue the first element of the queue, use the head and tail operators on lists:

  ```
  dequeue q = (head q, tail q)
  ```

- To enqueue an element, append it to the end of the list:

  ```
  enqueue e q = q ++ [e]
  ```

Naive Implementation

- This works, but the efficiency of `enqueue` is disappointing: each enqueue requires O(n) cons operations!

Of course, cons operations are not the only things that take time! But counting just conses actually yields a pretty good estimate of the (asymptotic) wall-clock efficiency of programs. Here, it is certainly clear that the real efficiency can be no *better* than O(n).

Better Implementation

- Idea:
  - Represent a queue using *two* lists:
    1. the “front part” of the queue
    2. the “back part” of the queue *in reverse order*
- E.g.:
  - `([1,2,3],[7,6,5,4])` represents the queue with elements 1,2,3,4,5,6,7
  - `([],[3,2,1])` and `([1,2,3],[])` both represent the queue with elements 1,2,3

Better Implementation

- To enqueue an element, just cons it onto the back list.
- To dequeue an element, just remove it from the front list...
- *...unless* the front list is empty, in which case we reverse the back list and use it as the front list from now on.

Better Implementation

```
data Queue a = Queue [a] [a]

enqueue :: a -> Queue a -> Queue a
enqueue e (Queue front back) =
  Queue front (e:back)

dequeue :: Queue a -> (a, Queue a)
dequeue (Queue (e:front) back) =
  (e, (Queue front back))
dequeue (Queue [] back) =
  dequeue (Queue (reverse back) [])
```

Efficiency

- Intuition: a `dequeue` may require O(n) cons operations (to reverse the back list), but this cannot happen too often.

Efficiency

In more detail:

- Note that each element can participate in <u>at most one</u> list reversal during its "lifetime" in the queue.
- When an element is enqueued, we can "charge two tokens" for two cons operations. One of these is performed immediately; the other we put "in the bank."
- At every moment, the number of tokens in the bank is equal to the length of the back list.
- When we find we need to reverse the back list to perform a `dequeue`, we will always have just enough tokens in the bank to pay for all of the cons operations involved.

Efficiency

- So we can say that the *amortized cost* of each enqueue operation is two conses.
- The amortized cost of each dequeue is zero (i.e., no conses --- just some pointer manipulation).

Caveat: This efficiency argument is somewhat rough and ready --- it is intended just to give an intuition for what is going on. Making everything precise requires quite a bit of work, especially in a lazy language.

Moral

- We can implement an ephemeral queue data structure whose operations have the same (asymptotic, amortized) efficiency as the standard (double-pointer) imperative implementation.

Binary Search Trees

- Suppose we want to implement a type **`Set a`** supporting the following operations:

```
empty :: Set a
member :: a -> Set a -> Bool
insert :: a -> Set a -> Set a
```

- One very simple implementation for sets is in terms of binary search trees...

Binary Search Trees

```
data Set a =
    E                                -- empty
  | T (Set a) a (Set a)              -- nonempty

empty = E

member x E = False
member x (T a y b)
            | x < y   = member x a
            | x > y   = member x b
            | True    = True

insert x E = T E x E
insert x (T a y b)
            | x < y   = T (insert x a) y b
            | x > y   = T a y (insert x b)
            | True    = T a y b
```

Quick Digression on Patterns

- The insert function is a little hard to read because it is not immediately obvious that the phrase "`T a y b`" in the body just means "return the input."

```
insert x E = T E x E
insert x (T a y b)
              | x < y  = T (insert x a) y b
              | x > y  = T a y (insert x b)
              | True   = T a y b
```

Quick Digression on Patterns

- Haskell provides *@-patterns* for such situations.

```
insert x E = T E x E
insert x t@(T a y b)
            | x < y   = T (insert x a) y b
            | x > y   = T a y (insert x b)
            | True    = t
```

- The pattern "`t@(T a y b)`" means "check that the input value was constructed with a `T` and bind its parts to `a`, `y`, and `b`, and <u>additionally</u> let `t` stand for the whole input value in what follows..."

Quick Digression on Types

- We said we wanted our set operations to have these types:

```
empty  :: Set a
member :: a -> Set a -> Bool
insert :: a -> Set a -> Set a
```

- But do they?

Quick Digression on Types

The implementation of `member`

```
member x E = False
member x (T a y b)
              | x < y   = member x a
              | x > y   = member x b
              | True    = member x b
```

uses the comparison operations `<` and `>` on elements of `a`.

But it does not make sense to compare elements of arbitrary types! (What would it mean to compare two functions, for example??)

Quick Digression on Types

- The actual type of `<` is

  `(<) :: Ord a => a -> a -> Bool`

- "`Ord a =>`" is a *qualifier* and "`Ord a => a -> a -> Bool`" is a *qualified type*. It can be read "if the type `a` is ordered, then `<` can take two `a`'s and return a boolean."
- Examples of ordered types include numbers, characters, strings, etc.
- User-defined types can also be declared to be ordered. We will return to this in (much) more detail when we discuss SOE chapter 12.

Quick Digression on Types

- Similarly, the types of our set operations are really:

```
empty  :: Set a
member :: Ord a => a -> Set a -> Bool
insert :: Ord a => a -> Set a -> Set a
```

Quick Digression on Types

(end of digressions)

Balanced Trees

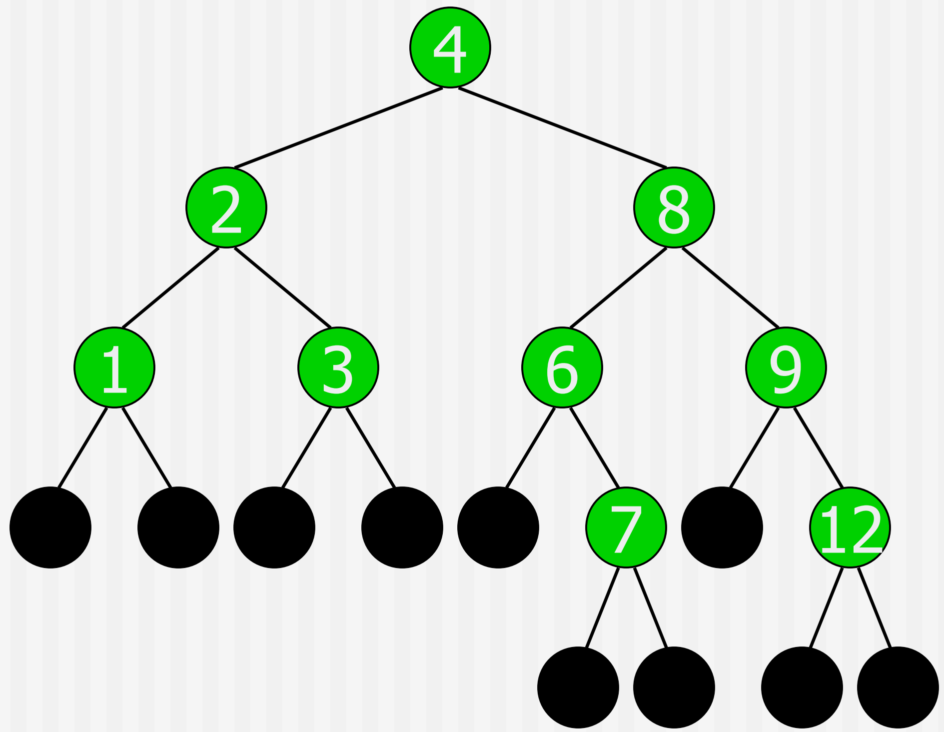
- If our sets grow large, we may find that the simple binary tree implementation is not fast enough: in the worse case, each insert or member operation may take O(n) time!
- We can do much better by keeping the trees balanced.
- There are many ways of doing this. Let's look at one fairly simple (but still very fast) one that you have probably seen before in an imperative setting: red-black trees.

Red-Black Trees

- A red-black tree is a binary search tree where every node is additionally marked with a color (red or black) and in which the following invariants are maintained...

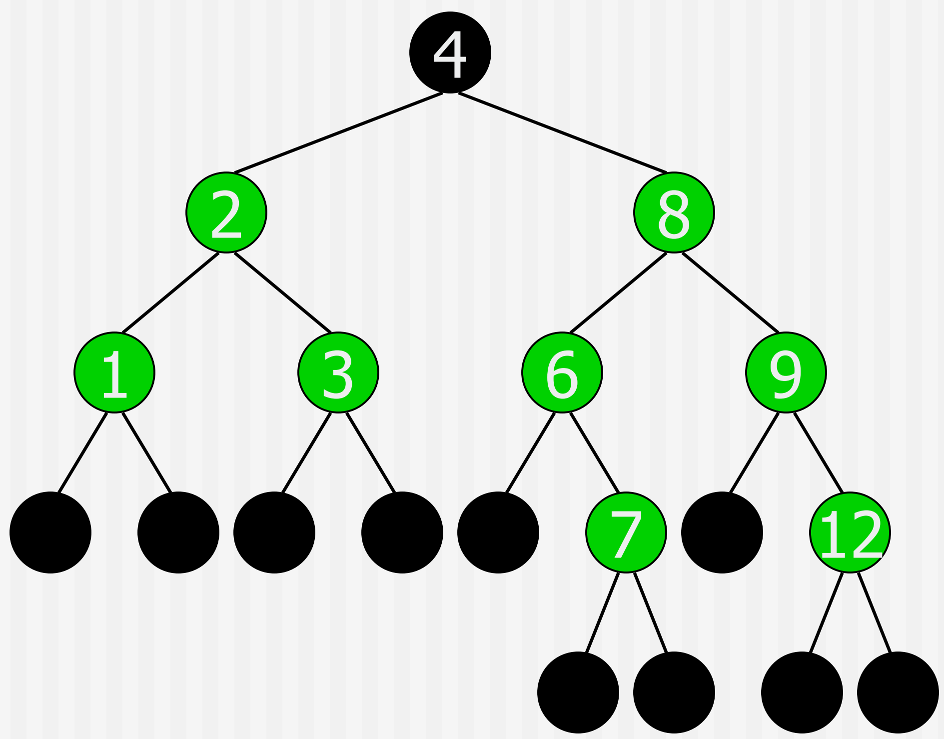
Invariants

- The empty nodes at the leaves are considered black.



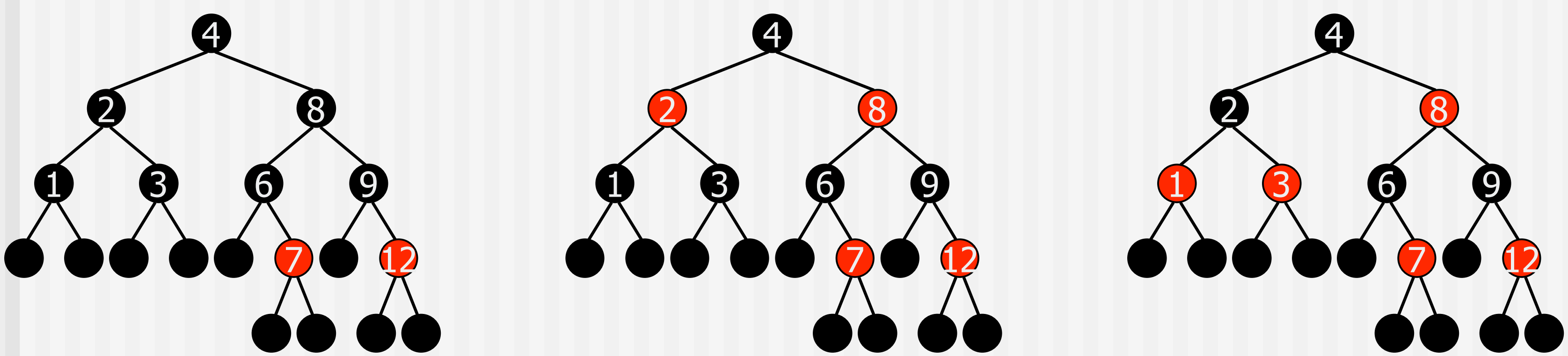
Invariants

- The root is always black.



Invariants

- From each node, every path to a leaf has the same number of black nodes.
- Red nodes have black children



Invariants

- Together, these invariants imply that every red-black tree is "approximately balanced," in the sense that the longest path to an empty node is no more than twice the length of the shortest.
- From this, it follows that all operations will run in $O(\log_2 n)$ time.

Now let's look at the details...

Type Declaration

- The data declaration is a straightforward modification of the one for unbalanced trees:

```
data Color = R | B

data RedBlackSet a =
    E
  | T Color
      (RedBlackSet a)
      a
      (RedBlackSet a)
```

Membership

- The empty tree is the same as before. Membership testing requires just a trivial change.

```
empty = E

member x E = False
member x (T _ a y b)
    | x < y   = member x a
    | x > y   = member x b
    | True    = member x b
```

Insertion is more interesting...

Insertion

- Insertion is implemented in terms of a recursive auxiliary function `ins`, which walks down the tree until it either gets to an empty leaf node, in which case it constructs a new (red) node containing the value being inserted...

```
ins E = T R E x E
```

Insertion

- ... or discovers that the value being inserted is already in the tree, in which case it returns the input unchanged:

```
ins s@(T color a y b)
   | x < y   = ...
   | x > y   = ...
   | True    = s
```

- The recursive cases are where the real work happens...

Insertion

- In the recursive case, `ins` determines whether the new value belongs in the left or right subtree, makes a recursive call to insert it there, and rebuilds the current node with the new subtree.

```
ins s@(T color a y b)
   | x < y  = ... (T color (ins a) y b)
   | x > y  = ... (T color a y (ins b))
   | True   = s
```

Insertion

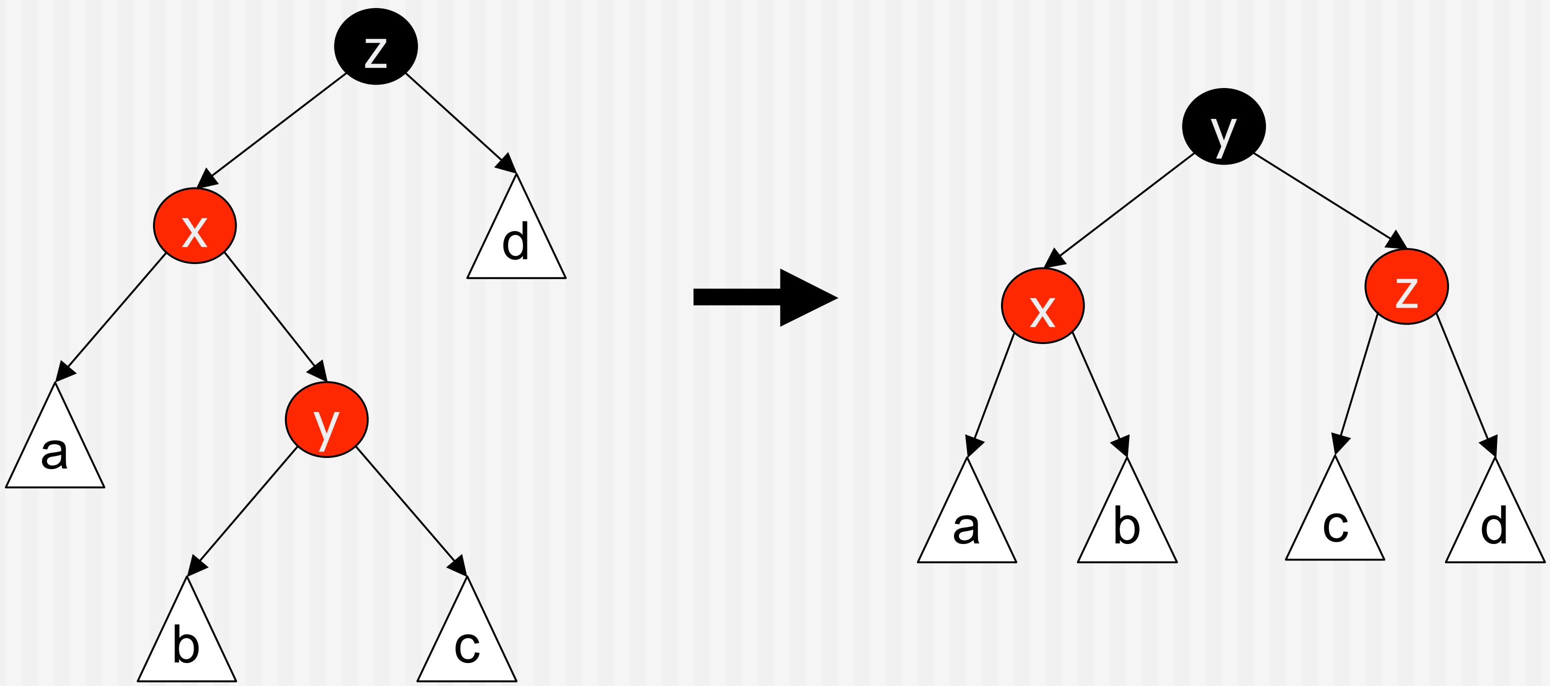
- Before returning it, however, we may need to rebalance to maintain the red-black invariants. The code to do this is encapsulated in a helper function **balance**.

```
ins s@(T color a y b)
  | x < y   = balance (T color (ins a) y b)
  | x > y   = balance (T color a y (ins b))
  | True    = s
```

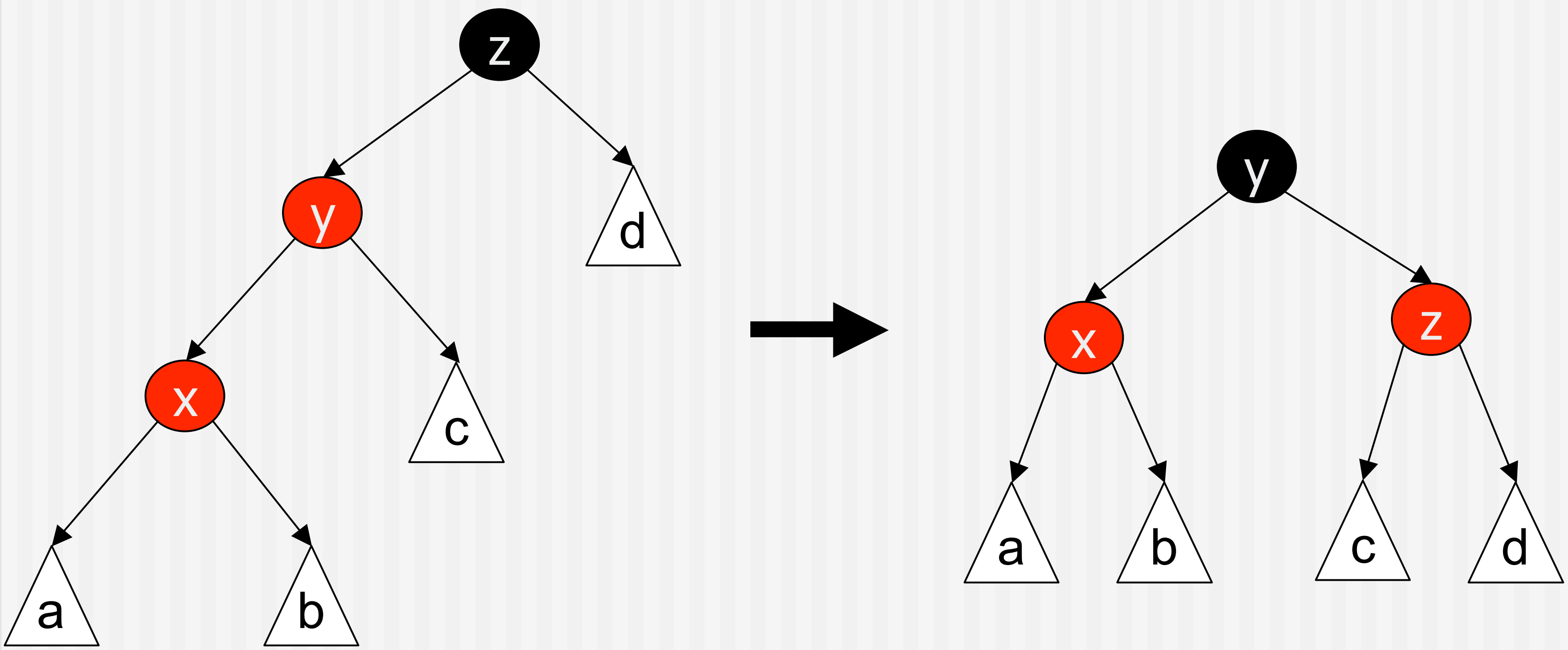
Balancing

- The key insight in writing the balancing function is that we do not try to rebalance as soon as we see a red node with a red child. Instead, we return this tree as-is and wait until we are called with the black parent of this node.
- I.e., the job of the `balance` function is to rebalance trees with a black-red-red path starting at the root.
- Since the root has two children and four grandchildren, there are four ways in which such a path can happen.

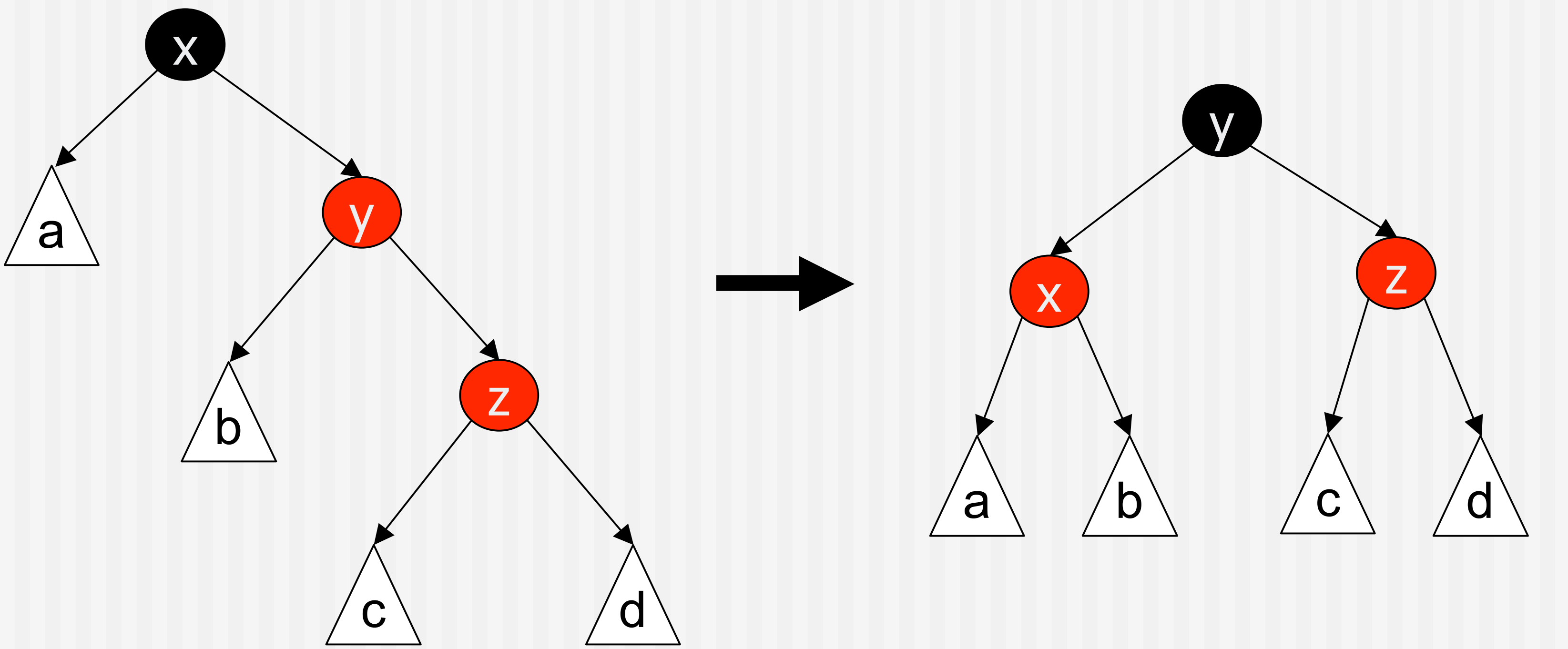
Balancing



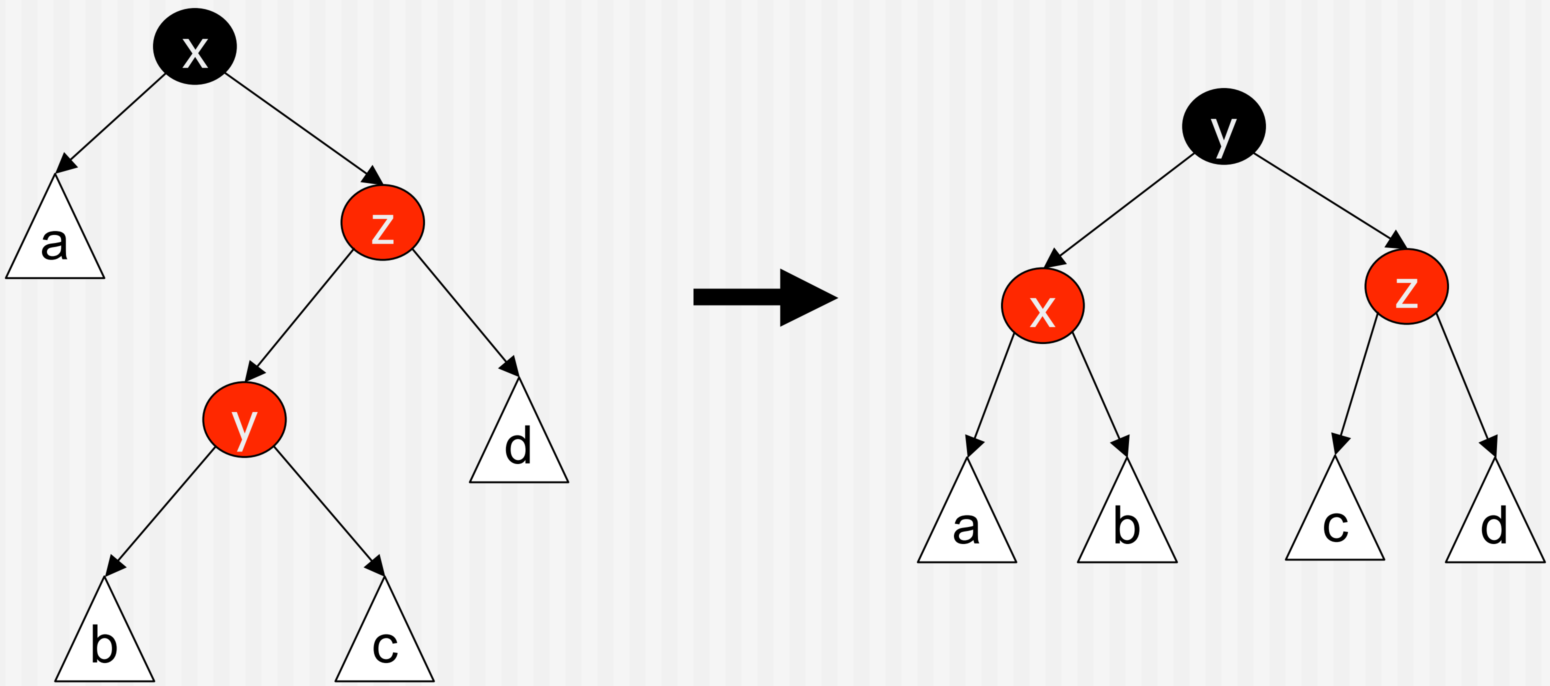
Balancing



Balancing



Balancing



All that remains is to turn these pictures into code...

Balancing

```
balance (T B (T R (T R a x b) y c) z d)
         = T R (T B a x b) y (T B c z d)

balance (T B (T R a x (T R b y c)) z d)
         = T R (T B a x b) y (T B c z d)

balance (T B a x (T R (T R b y c) z d))
         = T R (T B a x b) y (T B c z d)

balance (T B a x (T R b y (T R c z d)))
         = T R (T B a x b) y (T B c z d)

balance t = t
```

One Final Detail

- Since we only rebalance black nodes with red children and grandchildren, it is possible that the `ins` function could return a red node with a red child as its final result.
- We can fix this by forcing the root node of the returned tree to be black, regardless of the color returned by `ins`.

Final Version

```
insert x t = makeRootBlack (ins t)
  where
    ins E = T R E x E
    ins s@(T color a y b)
      | x < y  = balance (T color (ins a) y b)
      | x > y  = balance (T color a y (ins b))
      | True   = s
    makeRootBlack (T _ a y b) = T B a y b
```

The Whole Banana

```
data Color = R | B
data RedBlackSet a = E | T Color (RedBlackSet a) a (RedBlackSet a)

empty = E

member x E = False
member x (T _ a y b)
   | x < y     = member x a
   | x > y     = member x b
   | otherwise = True

balance (T B (T R (T R a x b) y c) z d) = T R (T B a x b) y (T B c z d)
balance (T B (T R a x (T R b y c)) z d) = T R (T B a x b) y (T B c z d)
balance (T B a x (T R (T R b y c) z d)) = T R (T B a x b) y (T B c z d)
balance (T B a x (T R b y (T R c z d))) = T R (T B a x b) y (T B c z d)
balance t = t

insert x t = colorRootBlack (ins t)
  where
    ins E = T R E x E
    ins s@(T color a y b)
      | x < y     = balance (T color (ins a) y b)
      | x > y     = balance (T color a y (ins b))
      | otherwise = s
    colorRootBlack (T _ a y b) = T B a y b
```

For Comparison...

```
/* This function can be called only if K2 has a left child */
    /* Perform a rotate between a node (K2) and its left child */
    /* Update heights, then return new root */

    static Position
    SingleRotateWithLeft( Position K2 )
    {
        Position K1;

        K1 = K2->Left;
        K2->Left = K1->Right;
        K1->Right = K2;

        return K1;  /* New root */
    }

    /* This function can be called only if K1 has a right child */
    /* Perform a rotate between a node (K1) and its right child */
    /* Update heights, then return new root */

    static Position
    SingleRotateWithRight( Position K1 )
    {
        Position K2;

        K2 = K1->Right;
        K1->Right = K2->Left;
        K2->Left = K1;

        return K2;  /* New root */
    }


    /* Perform a rotation at node X */
    /* (whose parent is passed as a parameter) */
    /* The child is deduced by examining Item */

    static Position
    Rotate( ElementType Item, Position Parent )
    {

        if( Item < Parent->Element )
            return Parent->Left = Item < Parent->Left->Element ?
                SingleRotateWithLeft( Parent->Left ) :
                SingleRotateWithRight( Parent->Left );
        else
            return Parent->Right = Item < Parent->Right->Element ?
                SingleRotateWithLeft( Parent->Right ) :
                SingleRotateWithRight( Parent->Right );
    }
```

```
static Position X, P, GP, GGP;

static
void HandleReorient( ElementType Item, RedBlackTree T )
{
    X->Color = Red;         /* Do the color flip */
    X->Left->Color = Black;
    X->Right->Color = Black;

    if( P->Color == Red )  /* Have to rotate */
    {
        GP->Color = Red;
        if( (Item < GP->Element) != (Item < P->Element) )
            P = Rotate( Item, GP );  /* Start double rotate */
        X = Rotate( Item, GGP );
        X->Color = Black;
    }
    T->Right->Color = Black;  /* Make root black */
}
```

```
RedBlackTree
Insert( ElementType Item, RedBlackTree T )
{
    X = P = GP = T;
    NullNode->Element = Item;
    while( X->Element != Item )  /* Descend down the tree */
    {
        GGP = GP; GP = P; P = X;
        if( Item < X->Element )
            X = X->Left;
        else
            X = X->Right;
        if( X->Left->Color == Red && X->Right->Color == Red )
            HandleReorient( Item, T );
    }

    if( X != NullNode )
        return NullNode;  /* Duplicate */

    X = malloc( sizeof( struct RedBlackNode ) );
    if( X == NULL )
        FatalError( "Out of space!!!" );
    X->Element = Item;
    X->Left = X->Right = NullNode;

    if( Item < P->Element )  /* Attach to its parent */
        P->Left = X;
    else
        P->Right = X;
    HandleReorient( Item, T ); /* Color it red; maybe rotate */

    return T;
}
```