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 exactly 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:

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

  ```haskell```

- 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:
  
  ```haskell```
type Queue a = [a]
```

- To dequeue the first element of the queue, use the `head` and `tail` operators on lists:
  
  ```haskell```
dequeue q = (head q, tail q)
```

- To enqueue an element, append it to the end of the list:
  
  ```haskell```
enqueue e q = q ++ [e]
```

Better Implementation

- Idea:
  
  ```haskell```
  - Represent a queue using two lists:
    1. the “front part” of the queue
    2. the “back part” of the queue in reverse order
  ```haskell```
  
  - E.g.:
    ```haskell```
    - `([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
  ```haskell```

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 **at most one** 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.

**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 more work, especially in a lazy language.

**Moral**

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

**Binary Search Trees**

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

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

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

**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.”
Quick Digression on Patterns

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

  ```haskell
  insert x E = T x E
  insert x t@(T a y b) = T (insert x 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, bind its parts to a, y, and b, and additionally let t stand for the whole input value in what follows...”

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.

- The root is always black.

- 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:

```haskell
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:

```haskell
empty = E

case member x a of
  | x < y  = member x a
  | x > y  = member x b
  | True   = True
```

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...

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

- 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.

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

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

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

\begin{verbatim}
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
\end{verbatim}

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 \texttt{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.

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

balance \( (T_s (T_R (T_R a \times b) y c) z d) \)  
\( = T_R (T_B a \times b) y (T_B c z d) \)

balance \( (T_s (T_R a x (T_R b y c)) z d) \)  
\( = T_R (T_B a \times b) y (T_B c z d) \)

balance \( (T_B a x (T_R b y c)) z d) \)  
\( = T_R (T_B a \times b) y (T_B c z d) \)

balance \( t = t \)

The Whole Banana

For Comparison...

One Final Detail

- Since we only rebalance black nodes with red children and grandchildren, it is possible that the \texttt{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 \texttt{ins}.

Final Version

\[ \text{insert } x \ t = \text{makeRootBlack} \ (\text{ins } t) \]

where

\[ \text{ins } E = T \ R \ E \times E \]
\[ \text{ins } s@(T \color{} a y b) \]
\[ | \ x < y \ = \text{balance} \ (T \color{} (\text{ins } a) y b) \]
\[ | \ x > y \ = \text{balance} \ (T \color{} a y (\text{ins } b)) \]
\[ | \ \text{True} \ = s \]

\[ \text{makeRootBlack} \ (T \_ a y b) = T \ B \ a y b \]

For the Whole Banana

\[ \text{data RedBlackSet } a = E | T \color{} \ (\text{RedBlackSet } a) \ a \ (\text{RedBlackSet } a) \]
\[ \text{empty } = E \]
\[ \text{member } x \ E = \text{false} \]
\[ \text{member } x \ (T \color{} a y b) \]
\[ | \ x < y \ = \text{member } x \ a \]
\[ | \ x > y \ = \text{member } x \ b \]
\[ | \ \text{otherwise} = \text{true} \]

\[ \text{balance} \ (T \color{} (T \ R a \times b) y c) z d) \]
\[ = T_R (T_B a \times b) y (T_B c z d) \]

\[ \text{balance} \ (T_B a x (T_R b y (T_R c z d))) \]
\[ = T_R (T_B a \times b) y (T_B c z d) \]

\[ \text{balance} \ (T_B a x (T_R b y c)) z d) \]
\[ = T_R (T_B a \times b) y (T_B c z d) \]

\[ \text{balance} \ (T_B a x (T_R b y c) z d) \]
\[ = T_R (T_B a \times b) y (T_B c z d) \]

\[ \text{balance } t = t \]

\[ \text{insert } x \ t = \text{colorRootBlack} \ (\text{ins } t) \]

where

\[ \text{ins } E = T \ R \ E \times E \]
\[ \text{ins } s@(T \color{} a y b) \]
\[ | \ x < y \ = \text{balance} \ (T \color{} (\text{ins } a) y b) \]
\[ | \ x > y \ = \text{balance} \ (T \color{} a y (\text{ins } b)) \]
\[ | \ \text{otherwise} = s \]

\[ \text{colorRootBlack} \ (T \_ a y b) = T \ B \ a y b \]