LECTURE 10 : Random Access Lists

"I'll just keep playing back
These fragments of time
Everywhere I go
These moments will shine"

— Daft Punk, Fragments of Time,
Random Access Memories, 2013

```
> inc :: Binary -> Binary
> inc []       = [I]
> inc (O : bs) = I : bs
> inc (I : bs) = O : inc bs
```

Lists have an expensive lookup:

```
> (!!) :: [a] -> Int -> a
> (x:xs) !! 0 = x
> (x:xs) !! k = xs !! (k-1)
```

$\in O(n)$

We will use a different representation:

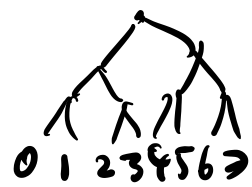
```
> data Tree a = Leaf a
              | Node Int (Tree a) (Tree a)
```

↑ size of the tree

```
> size :: Tree a -> Int
> size (Leaf x)        = 1
> size (Node n lt rt)  = n
>                      = size lt + size rt
```

We use a smart constructor to ensure that the size invariant is true:

```
> node :: Tree a -> Tree a -> Tree a
> node lt rt = Node (size lt + size rt) lt rt
```




```
> (!!) :: Tree a -> Int -> a
> Leaf x !! 0 = x
> Node n lt rt !! k
>     | k < m      = lt !! k
>     | otherwise  = rt !! k-m
>     where m = size lt
```

Our goal is to insert elements into trees, just like we did incr for Binary numbers:

Binary numbers were of the form:

$$b_0 \, b_1 \, b_2 \, \dots \, b_n$$

This represents the number:

$$2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \dots + 2^n b_n$$

We will have a datastructure containing "perfect" trees:

$$[t_0, t_1, t_2, \dots, t_n]$$

where

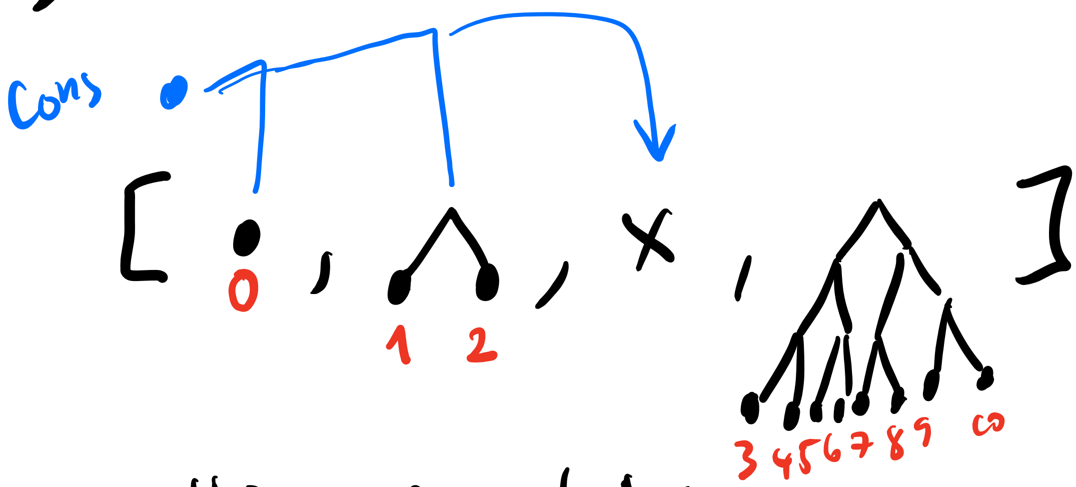
$$\begin{aligned}
size(t_0) &= 2^0 \quad \text{or} \quad 0 \\
size(t_1) &= 2^1 \quad \text{or} \quad 0 \\
size(t_2) &= 2^2 \quad \text{or} \quad 0 \\
&\vdots \\
size(t_n) &= 2^n \quad \text{or} \quad 0
\end{aligned}$$

This is the idea behind a random access list:

```
> type RAList a = [Maybe (Tree a)]
```

Subject to the invariance on the tree sizes above.

```
> instance List RAList where
>   ...
>   toList :: RAList a -> [a]
>
```



this corresponds to:

[I , I , O , I] :: Binary.

```
> toList :: RAList a -> [a]
> toList xs = concat (map to xs)
>   where
>     to :: Maybe (Tree a) -> [a]
>     to Nothing = []
>     to (Just t) = toList t
```

```
toList :: Tree a -> [a]
toList (Leaf x) = [x]
toList (Node n lt rt) =
    toList lt ++ toList rt
```

```
> (!!) :: RAList a -> Int -> a
> (Nothing : ts) !! k = ts !! k
> (Just t : ts) !! k
>     | k < m      = t !! k
>     | otherwise  = ts !! k-m
>     where m = size t
```

!! on Tree

!! on RAList

Remember:

```
> inc :: Binary -> Binary
> inc []     = [I]
> inc (O:bs) = I:bs
> inc (I:bs) = O : inc bs
```

```
> cons :: a -> RAList a -> RAList a
> cons x xs = consT (Leaf x) xs
>   where
>     consT :: Tree a -> RAList a -> RAList a
>     consT t [] = [Just t]
>     consT t (Nothing : ts) = Just t : ts
>     consT t (Just t' : ts) =
>           Nothing : consT (node t t') ts
```