50001 - Algorithm Analysis and Design - Lecture 10

Oliver Killane

18/11/21

List lookup

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

As you ca see **!!** costs $O(n)$ as it may traverse the entire list.

If we want this access to be faster, we can use trees:

```
1  data Tree a = Tip | Leaf a | Node Int (Tree a) (Tree a)
2
3  node :: Tree a -> Tree a -> Tree a
4  node l r = Node (size l + size r) l r
5
6  instance List Tree where
7    toList :: Tree a -> [a]
8    toList Tip = []
9    toList (Leaf n) = [n]
10   toList (Fork n l r) = toList l ++ toList r
11
12   -- Invariant: size Node n a b = n = size a + size b
13   length :: Tree a -> Int
14   length Tip = 0
15   length (Leaf _) = 1;
16   length (Node n _ _) = n;
17
18   (!!)  ::  Tree a -> Int -> a
19   (Leaf x) !! 0 = x
20   (Node n l r) !! k
21      | k < m = l !! k
22      | otherwise = r !! (k-m)
23     where m = length l
24
25   -- case for Tip !! n or Leaf !! >0
26   _ !! _ = error "(!!): Cannot get list index"
```

This costs $O(\log n)$ as each recursive call acts on half of the remaining list.

However we hav difficulty with insertion:

Insert Quickly	$n : t = node(Leaf\,n)t$	Effectively becomes a linked list, $\log n$ search time ruined.
Insert Slowly	Rebalance tree (e.g **AVL** tree)	Complex and no longer $O(1)$ insert.

Random Access Lists

A list containing elements that are either nothing, or a perfect tree with size the same as 2^{index}.
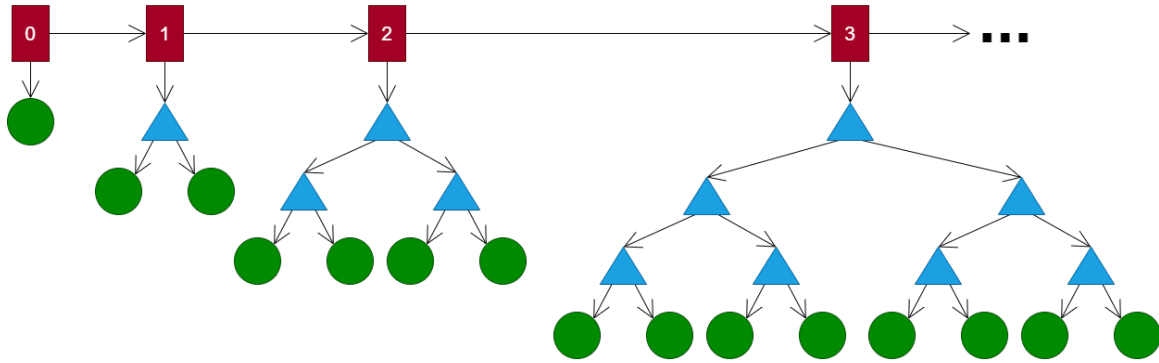
The empty tree can be represented by a *Tip* value (from the notes), or using type $Maybe(Tree)$ (from the lecture) where $Tree\ a = Leaf\ x\ |\ Node\ n\ l\ r$.

When we add to a tree, we add to the first element of the **RAList**, if the invariant is breached (no longer perfect tree of size 2^0) it can be combined with the next list over (if empty, place, else combine and repeat).

This way while the worst case insert is $O(n)$, our amortized complexity is $O(1)$ much as with increment.

```haskell
1   data Tree a = Tip | Leaf a | Node Int (Tree a) (Tree a)
2
3   type RAList a = [Tree a]
4
5   instance List RAList where
6     toList :: RAList a -> [a]
7     toList (RaList ls) = concatMap toList ls
8
9     fromList :: [a] -> RAList a
10    fromList = foldr (:) empty
11
12    empty :: RAList a
13    empty = []
14
15    (:) :: a -> RAList a -> RAList a
16    n : [] = [Leaf n]
17    n : (RAList ls) = RAList (insertTree (Leaf n) ls)
18      where
19        insertTree :: Tree a -> [Tree a] -> [Tree a]
20        insertTree t []      = [t]
21        insertTree t (tip:ls) = t:ls
22        insertTree t (t':ts)  = Tip: insertTree (node t t') ts
23
24    length :: RAList a -> Int
25    length (RAList ls) = foldr ((+) . length) 0 ls
26
27    (!!) :: RAList a -> Int -> a
28    (RAList []) !! _ = error "(!!): empty list"
29    (RAList (x:xs)) !! n
30      | n < m     = x || n
31      | otherwise = (RAList xs) !! (n-m)
32      where m = length x
```