50001 - Algorithm Analysis and Design - Lecture 10

Oliver Killane

18/11/21

Lecture Recording

Lecture recording is available here

List lookup

```haskell
(!!) :: [a] -> Int -> a
(x:xs) !! 0 = x
(_: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:

```haskell
data Tree a = Tip | Leaf a | Node Int (Tree a) (Tree a)

node :: Tree a -> Tree a -> Tree a
node l r = Node (size l + size r) l r

instance List Tree where
  toList :: Tree a -> [a]
  toList Tip = []
  toList (Leaf n) = [n]
  toList (Fork n l r) = toList l ++ toList r

  -- Invariant: size Node n a b = n = size a + size b
  length :: Tree a -> Int
  length Tip = 0
  length (Leaf _) = 1;
  length (Node n _ _) = n;

  (!!) :: Tree a -> Int -> a
  (Leaf x) !! 0 = x
  (Node n l r) !! k
    | k < m = l!!k
    | otherwise = r!! (k-m)
    where m = length l

  -- case for Tip !! n or Leaf !! >0
  _ !! _ = error "(!!): Cannot get list index"
```

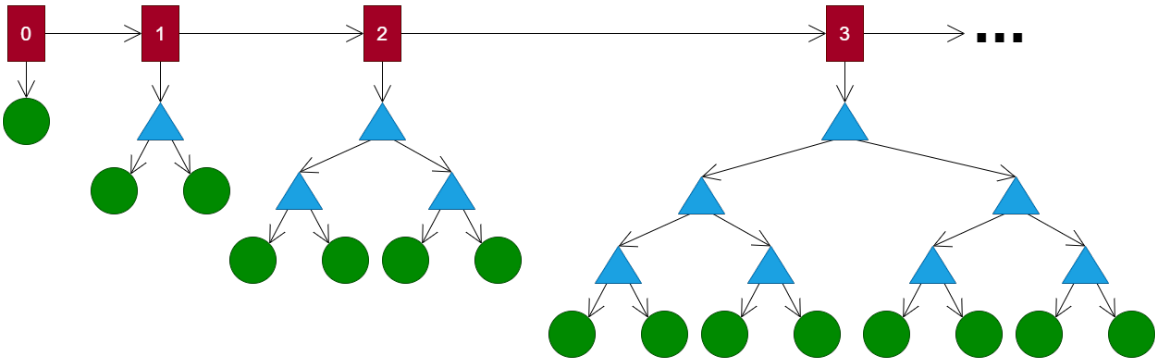
This costs $O(\log n)$ as each recursive call acts on half of the remaining list.

However we have 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 $Tree\ a = Leaf\ x \mid Node\ n\ l\ r$.

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.

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

type RAList a = [Tree a]

instance List RAList where
  toList :: RAList a -> [a]
  toList (RaList ls) = concatMap toList ls

  fromList :: [a] -> RAList a
  fromList = foldr (:) empty

  empty :: RAList a
  empty = []

  (:) :: a -> RAList a -> RAList a
  n : [] = [Leaf n]
  n : (RAList ls) = RAList (insertTree (Leaf n) ls)
    where
      insertTree :: Tree a -> [Tree a] -> [Tree a]
      insertTree t ([])     = [t]
      insertTree t (Tip:ls) = t:ls
      insertTree t (t':ts)  = Tip: insertTree (node t t') ts

  length :: RAList a -> Int
  length (RAList ls) = foldr ((+) . length) 0 ls

  (!!) :: RAList a -> Int -> a
  (RAList []) !! _ = error "(!!): empty list"
  (RAList (x:xs)) !! n
    | isEmpty x = (RAList ts) !! k
    | n < m     = x !! n
    | otherwise = (RAList xs) !! (n-m)
    where m = length x
```