50001 - Algorithm Analysis and Design - Lecture 11

Oliver Killane

18/11/21

Lecture Recording

Lecture recording is available here

Equality

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

Eq is the typeclass for equality, any instance of this class should ensure the equality satisfied the laws:

$$\begin{array}{rl} \text{reflexivity} & x == x \\ \text{transitivity} & x == y \land y == z \Rightarrow x == z \\ \text{symmetry} & x == y \Rightarrow y == x \end{array}$$

We also expect the idescernability of identicals (Leibniz Law):

$$x == y \Rightarrow f\ x == f\ y$$

For a set-like interface we have a member function:

```haskell
(in) :: Eq a => a -> Set a -> Bool
```

If we assume only that Eq holds, the complexity is $O(n)$ as we must potentially check all members of the set. To get around this, we use ordering.

Orderings

The Ord typeclass allows us to check for inequalities:

```haskell
class Eq a => Ord a where
    (<=) :: a -> a -> Bool
    (<)  :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    (>)  :: a -> a -> Bool
```

We must try to ensure certain properties hold, for example for to have a partial order we require:

$$\begin{array}{rl} \text{reflexivity} & x \leq x \\ \text{transitivity} & x \leq y \land y \leq z \Rightarrow x \leq z \\ \text{antisymmetry} & x \leq y \land y \leq x \Rightarrow x == y \end{array}$$

There are also total orders (all elements in the set are ordered compared to all others), for which we add the constraint:

$$\text{connexity} \quad x \leq y \lor y \leq x$$

Ordered Sets

```haskell
class OrdSet ordset where
  empty     :: ordset n
  insert    :: Ord a => a -> ordset a -> ordset a
  member    :: Ord a => a -> ordset a -> Bool
  fromList  :: Ord a => [a] -> ordset a
  toList    :: Ord a => ordset a -> [a]
```

We can implement this class for Trees:

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

instance OrdSet Tree where
  empty :: Tree n
  empty = Tip

  insert :: Ord a => a -> Tree a -> Tree a
  insert x Tip = Node Tip x Tip
  insert x (Node l y r)
    | x == y     = t
    | x < y      = Node (insert x l) y r
    | otherwise = Node l y (insert x r)

  member :: Ord a => a -> Tree a -> Bool
  member x Tip = False
  member x (Node l y r)
    | x == y     = True
    | x < y      = member x l
    | otherwise = member x r

  fromList :: Ord a => [a] -> Tree a
  fromList = foldr insert empty

  toList :: Ord a => Tree a -> [a]
  toList Tip          = []
  toList (Node l y r) = toList l ++ y:toList r
```

However the worst case here is still $O(n)$ as we do not balance the tree as more members are inserted. If the members are added in order, the tree devolves to a linked list.

We need a way to create a tree that self balances.

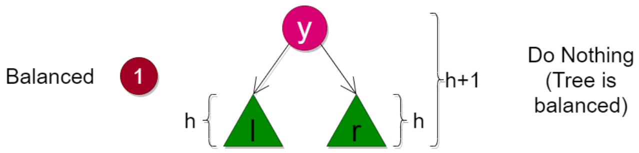
Binary Search Trees (AVL Trees)

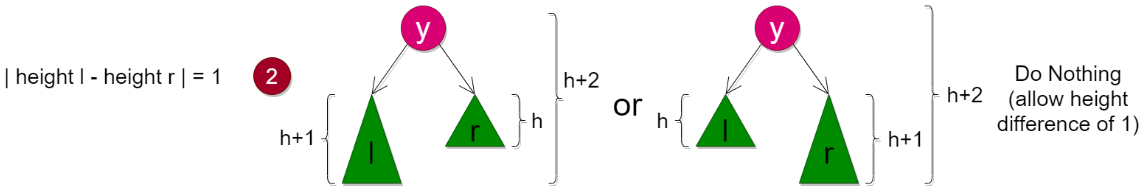
```haskell
-- H for height! The int stored at a node is the height of that tree.
data HTree a = HTip | HNode Int (HTree a) a (HTree a)

height :: HTree a -> Int
height HTip = 0
height (HNode h _ _ _) = h

hnode :: HTree a -> a -> HTree a -> HTree a
hnode l x r = HNode h l x r
  where h = max (height l) (height r) + 1
```

When inserting into the tree we must keep the tree balanced such that no subtree's left is more than one higher than its' right.

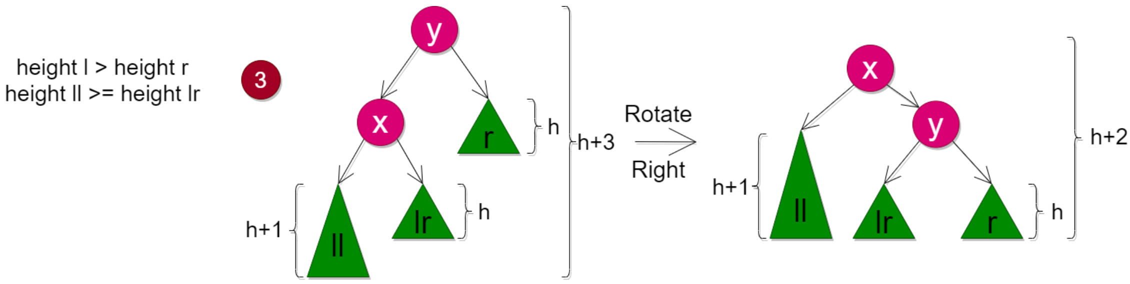
```haskell
insert :: Ord a => a -> HTree a -> HTree a
insert x HTip = hnode Tip x Tip
insert x t@(HNode _ l y r)
  | x == y    = t
  | x < y     = balance (insert x l) y r
  | otherwise = balance l y (insert x r)
```

We must rebalance the tree after insertion, this must consider the following cases:



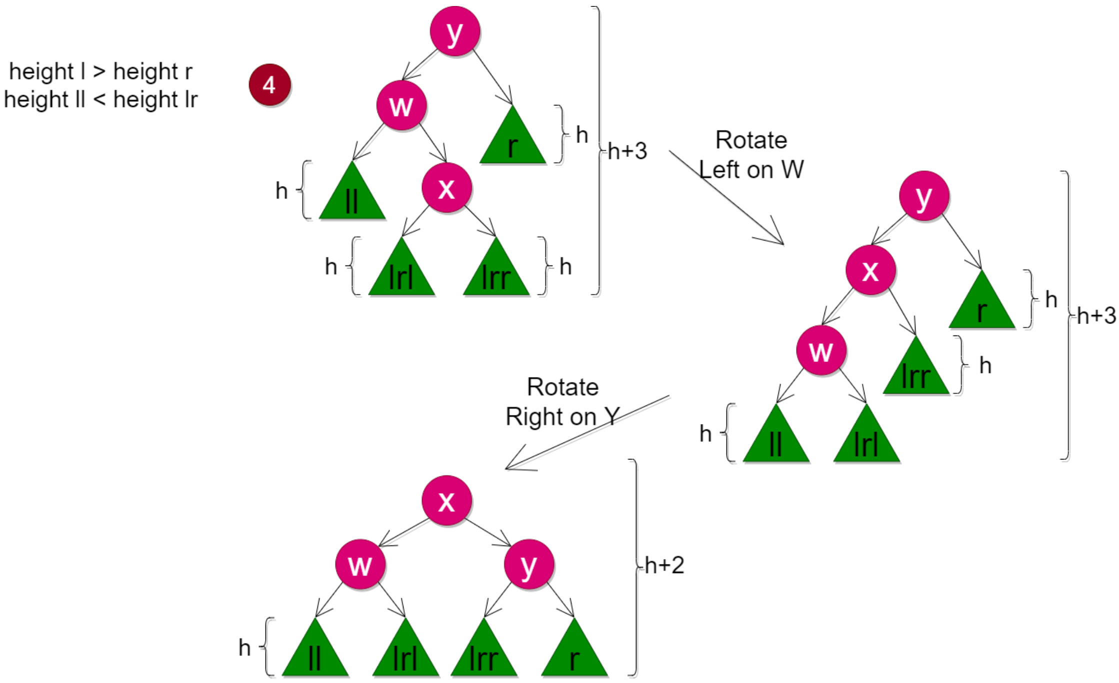


When the tree's balanced invariant has been broken, we must follow these cases:




```haskell
rotr :: HTree a -> HTree a
rotr (HNode _ (HNode _ ll x lr) y r) = hnode ll x (hnode lr y r)
```

When the right subtree's right subtree is higher:




```haskell
rotl :: HTree a -> HTree a
rotl (HNode _ ll w (HNode _ lrl x lrr))
  = hnode (hnode ll w lrl) x lrr

-- so for the example: HNode _ wt y r -> rotr ( hnode (rotl wt) y r)
```

We can use rotate left and rotate right to for the two cases where the right subtree is 2 or more higher than the left.

```haskell
balance :: Ord a => HTree a -> a -> HTree a -> HTree
balance l x r
  | lh == lr || abs (lr - lr) == 1 = t
  | lh > lr    = rotr(hnode (if height ll < height lr then rotl l else l) x r)
  | otherwise = rotl(hnode l x (if height rl > height rr then rotr r else r))
  where
    lh = height l
    rl = height r
    (HNode _ ll _ lr) = l
    (HNode _ rl _ rr) = r
```