50001 - Algorithm Analysis and Design - Lecture 15

Oliver Killane

29/11/21

Lecture Recording

Lecture recording is available here

Randomized Treaps

By using a random value for priority when inserting values into the treap, we can ensure a high likelihood of balancing, without complex balancing being required.

We can use this to create a randomized quicksort.

```haskell
import System.Random (StdGen, mkStdGen, random)
-- node random :: StdGen -> (Int, StdGen)

data RTreap a = RTreap StdGen (Treap a)

insert :: Ord a => a -> RTreap a -> RTreap a
insert x (RTreap seed t) = RTreap seed' (pinsert x p t)
  where (p, seed') = random seed

-- note 42 is used for
empty :: RTreap a
empty = RTreap (mkStdGen 42) Empty

-- Build up tree, requires O(n log n)
fromList :: Ord a => [a] -> RTreap a
fromList xs = foldr insert empty xs

-- Linear time conversion (use treap tolist)
toList :: RTreap a -> [a]
toList (RTreap _ t) = tolist t

-- Randomiozed Quicksort O(n log n)
-- Effectively the random priorities are the partitions, first pivot is the
-- highest priority.
rquicksort :: Ord a => [a] -> [a]
rquicksort = toList . fromList
```

Randomized Binary Trees

We can balance a binary tree without using a treap, by inserting at the root (and rotating the tree to ensure it is ordered) with a certain probability.

```haskell
import System.Random (StdGen, mkStdGen, randomR)

data BTree a = Empty | Node (BTree a) a (BTree a)

insert :: Ord a => a -> BTree a -> BTree a
insert x Empty = Node Empty x Empty
insert x t@(Node l y r)
  | x == y = t
  | x < y = Node (insert x l) y r
  | otherwise = Node l y (insert x r)

```

```haskell
-- basic lefty/right rotations
rotr :: BTree a -> a -> BTree a -> BTree a
rotr (Node a x b) y c = Node a x (Node b y c)
rotr _ _ _= error "(rotr): left was empty"

rotl :: BTree a -> a -> BTree a -> BTree a
rotl a x (Node b y c) = Node (Node a x b) y c
rotl _ _ _= error "(rtol): right was empty"


-- Insert to the root of the tree (maintaining order)
insertRoot :: Ord a => a -> BTree a -> BTree a
insertRoot x Empty = Node Empty x Empty
insertRoot x t@(Node l y r)
  | x == y = t
  | x < y = rotr (insertRoot x l) y r
  | otherwise = rotl l y (insertRoot x r)

-- Randomized binary tree
data RBTree a = RBTree StdGen Int (BTree a)

empty :: RBTree a
empty = RBTree (mkStdGen 42) 0 Empty

-- chance of 1 / n+1 of inserting at root.
insert' :: Ord a => a -> RBTree a -> RBTree a
insert' x (RBTree seed n t) = RBTree seed' (n+1) (f x t)
  where
    f = case p of
      0 -> insertRoot
      _ -> insert
    (p, seed') = randomR (0,n) seed
```

For every insert we have chance $\frac{1}{n+1}$ of inserting at the root of the tree. Then this occurs, the contents are rotated to ensure the tree's ordering is maintained.

This means that there is a very high probability of balance being maintained, however correct results are only returned when distinct elements are inserted at most once.