Dynamic Order Statistics

Operations

A set that supports dynamic order statistics has the operations

- `insert(S, x)`
- `delete(S, x)`
- `select(S, k)`
  - given an order statistic k, return the k^{th} item in the set
- `rank(S, x)`
  - return the order statistic of `x` (given as pointer/reference)

Note that `select` and `rank` are inverses!

- `select(S, rank(S, x)) = select(S, kx) = x`
- `rank(S, select(S, k)) = rank(S, xk) = k`

Naive Implementation

"Nobody wants the naive implementation"

- Danny Heap

At each node x, store the rank of x

- good for:
  - `select` : can go down the tree in $\Theta O(h)$ time to find the k^{th} element
  - `rank` : x stores its rank, so this takes $\Theta O(1)$ time
- bad for:
  - `insert` , `delete` : these change order, which means that

many ($\Theta(n)$) stored ranks must be changed

Better implementation

At each node x, store the size of the subtree rooted at x

- `RR(x)`: finds relative rank within subtree
    - `RR(x) => size(left(x)) + 1`

`select`

```
def select(S, k):
    if k == RR(x):
        return x
    elif k < RR(x):
        return select(left(x), k)
    else:
        return select(right(x), k - RR(x))
```

Efficiency

- in worst case, `Select` must recurse until it reaches a leaf
- thus its complexity is $\Theta(h)$
    - for a balanced tree, this is $\Theta(\log(n))$

`rank`

```
def rank(S, x):
    r = RR(x)
    y = x
    while y != root(T)
        if y == right(parent(y))
            r += RR(parent(y))
        y = parent(y)
    return r
```

Efficiency

- also goes down the tree, so takes $\Theta(\log(n))$ time

`insert`, `delete` - Maintaining Size

After insertion/deletion, must go upwards and:

- recalculate size field for each ancestor node
- rebalance the tree, i.e.
    - recalculate balance factors
    - perform necessary rotations
        - recalculate sizes after rotations
            - only sizes of rotated roots change, so a rotation still takes constant time

Efficiency

- just like normal AVL insert, takes $\Theta(\log(n))$ time