MCS-375: Algorithms: Analysis and Design Handout #DS9
San Skulrattanakulchai
Gustavus Adolphus College Sep 29, 2014

Augmented Binary Search Tree Algorithms

CLRS: Ch 14.1

Most applications of binary search trees manipulate more than one key.
Example: In a database of PC's, we want to know how many computers sell for between \$1,000 and \$1,500.

Dynamic Order Statistics

The ith smallest key in a set has rank i.

SELECT(`i`):
i is an integer between 1 and n, the number of nodes in T.
SELECT(`i`) returns the node whose key has rank i.

RANK(`x`):
x is a node in T.
RANK(`x`) returns the rank of `x.key`.

So we can use SELECT to find the median of a data set.
We will extend binary search trees to handle these operations (as well as all previous ones) in time $O(h)$, where h is the height of the tree.

Idea
Each node `x` has a field
`x.size` equal to the number of descendants of `x`.
It's easy to check that the rank of a node `x` equals
$\sum\{$ 1+`y.left.size` : `y` = `x` or `y` = any ancestor of `x` containing `x` in its right subtree $\}$.
It's convenient to have `NIL` node with `NIL.size` = 0.
This is even true for bst's and splay trees.

Algorithms

RANK(`x`):

Climb up x's access path, computing the above sum.

SELECT(`i`):

Starting at the root, descend to the element of rank `i`
by maintaining the rank of the node we are at (using the above sum)
and descending left if the rank is too big, right if too small

INSERT:

When adding a new leaf `x`, increase `u.size` by 1 for every node `u` on `x`'s access path.

DELETE:

When splicing out node `y`, decrease `u.size` by 1 for every node `u` on `y`'s access path.

Extension to red-black trees

These algorithms extend to red-black trees, achieving time $O(\log n)$ for all operations.

The only change is when we do a rotation (in INSERT or DELETE).
A rotation only changes the size of 2 nodes.
It is easy to compute `x.size` for any node `x` using the identity
`x.size` = 1 + `x.left.size` + `x.right.size`.

The algorithms extend to splay trees similarly with similar (amortized) time bounds.

Splitting & Joining

Sometimes we need to combine search trees, and also to break them up.

JOIN(T_1, x, T_2): (CLRS Problem 13–2)
T_1 & T_2 are search trees, and x is a node.
`x.key` is $\geq$ every key in T_1 and $\leq$ every key in T_2.
JOIN returns a search tree consisting of the keys in T_1, T_2, and x.
The operation is *destructive*: both T_1 and T_2 are destroyed.

SPLIT(T, x):
T is a search tree containing node x.
`split` returns search trees T_1, T_2 and node x.
where T_1 contains all nodes in T with `key` $<$ `x.key`
and T_2 contains all nodes in T with `key` $>$ `x.key`.
We assume distinct keys.
This operation is also *destructive*.

Red-black trees can implement JOIN, SPLIT, and all previous operations in time $O(\log n)$ worst-case time per operation.
Splay trees can implement them in time $O(\log n)$ amortized time per operation.

An example application of JOIN is to use it to find a minimum spanning tree.