Design and Analysis
of Algorithms I

Data Structures

Binary Search Tree Basics

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

OPERATIONS	RUNNING TIME
SEARCH	$\theta(\log(n))$
SELECT	$O(\log(n))$
MIN/MAX	$O(\log(n))$
PRED/SUCC	$O(\log(n))$
RANK	$O(\log(n))$
OUTPUT IN SORTED ORDER	$O(n)$
INSERT	$O(\log(n))$
DELETE	$O(\log(n))$

Also supported by hash tables

Also supported by heaps

Up from O(1)

new

Binary Search Tree Structure

-- exactly one node per key
-- most basic version :
each node has
-- left child pointer
-- right child pointer
-- parent pointer

SEARCH TREE PROPERTY :
(should hold at every node of the search tree)

Tim Roughgarden

The Height of a BST

Height = 2

Note : many possible trees for a set of keys.

(aka depth) longest root-leaf path

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

Worst case, a chain

Best case, perfectly balanced

Height = 4

Searching and Inserting

To Search for key k in tree T

-- start at the root

-- traverse left / right child pointers as needed

If k < key at current node If k > key at current node

-- return node with key k or NULL, as appropriate

To Insert a new key k into a tree T

Exercise : preserves search tree property!

-- search for k (unsuccessfully)

-- rewire final NULL ptr to point to new node with key k

Tim Roughgarden

The worst-case running time of Search (or Insert) operation in a binary search tree containing n keys is…?

- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

Min, Max, Pred, And Succ

To compute the minimum (maximum) key of a tree

- Start at root
- Follow left child pointers (right ptrs, for maximum) untill you cant anymore (return last key found)

To compute the predecessor of key k

- Easy case : If k's left subtree nonempty, return max key in left subtree
- Otherwise : follow parent pointers until you get to a key less than k.

Happens first time you "turn left"

Exercise : prove this works

Tim Roughgarden

The worst-case running time of the Max operation in a binary search tree containing n keys is…?

- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

In-Order Traversal

TO PRINT OUT KEYS IN INCREASING ORDER

-Let r = root of search tree, with subtrees TL and TR

- recurse on TL

[by recursion (induction) prints out keys of TL in increasing order]

-Print out r's key

-Recurse on TR

[prints out keys of TR in increasing order]

RUNNING TIME

O(1) time, n recursive calls => O(n) total

Deletion

TO DELETE A KEY K FROM A SEARCH TREE

- SEARCH for k

EASY CASE (k's node has no children)

-Just delete k's node from tree, done

MEDIUM CASE (k's node has one child)

(unique child assumes position previously held by k's node)

Deletion (con'd)

DIFFICULT CASE (k's node has 2 children)

-Compute k's predecessor l
  [i.e., traverse k's (non-NULL) left child ptr, then right child ptrs until no longer possible]
- SWAP k and l !

NOTE : in it's new position, k has no right child !
 => easy to delete or splice out k's new node

Exercise : at end, have a valid search tree !

RUNNING TIME : θ(height)

Select and Rank

Idea : store a little bit of extra info at each tree node about the tree itself (i.e., not about the data)

Example Augmentation : size(x) = # of tree nodes in subtree rooted at x.

Note : if x has children y and z,

then size(y) + size(z) + 1

Population in left subtree — Right subtree — x itself

Also : easy to keep sizes up-to-date during an Insertion or Deletion (you check!)

Select and Rank (con'd)

HOW TO SELECT I^{th} ORDER STATISTIC FROM AUGMENTED SEARCH TREE (with subtree sizes)

- start at root x, with children y and z
- let a = size(y) [a = 0 if x has no left child]
- if a = i-1, return x's key
- if a >= I, recursively compute i^{th} order statistic of search tree rooted at y
- if a < i-1 recursively compute $(i-a-1)^{th}$ order statistic of search tree rooted at z

RUNNING TIME = θ(height).

[EXERCISE : how to implement RANK ?

Design and Analysis of Algorithms I

Data Structures

Balanced Search Trees: Supported Operations

Sorted Arrays: Supported Operations

3	6	10	11	17	23	30	36

BUT WHAT ABOUT
INSERTIONS + DELETIONS ?
(would take θ(n) time)

OPERATIONS	RUNNING TIME
SEARCH	θ(log(n))
SELECT (given order statistic I)	O(1)
MIN/MAX	O(1)
PRED/SUCC (given pointer to a key)	O(1)
RANK (i.e., # of keys less than or equal to a given value)	O(log(n))
OUTPUT IN SORTED ORDER	O(n)

Tim Roughgarden

Balanced Search Trees: Supported Operations

Raison d'etre : like sorted array + fast (logarithmic) inserts + deletes !

OPERATIONS	RUNNING TIME
SEARCH	$\theta(\log(n))$
SELECT	$O(\log(n))$
MIN/MAX	$O(\log(n))$
PRED/SUCC	$O(\log(n))$
RANK	$O(\log(n))$
OUTPUT IN SORTED ORDER	$O(n)$
INSERT	$O(\log(n))$
DELETE	$O(\log(n))$

Also supported by hash tables

Also supported by heaps

Up from O(1)

new

Tim Roughgarden

Data Structures

Red-Black Trees

Design and Analysis
of Algorithms I

Binary Search Tree Structure

-- exactly one node per key
-- most basic version :
each node has
-- left child pointer
-- right child pointer
-- parent pointer

SEARCH TREE PROPERTY :
(should hold at every node of the search tree)

The Height of a BST

Height = 2

Note : many possible trees for a set of keys.

(aka depth) longest root-leaf path

Note : height could be anywhere from $\sim \log_2 n$ to $\sim n$

Worst case, a chain

Best case, perfectly balanced

Height = 4

Balanced Search Trees

Idea : ensure that height is always O(log(n)) [best possible]
$\Rightarrow$Search / Insert / Delete / Min / Max / Pred / Succ will then run in O(log(n)) time [n = # of keys in tree]

Example : red-black trees [Bayes '72, Guibas-Sedgewick '78]

[see also AUL trees, splay trees, B trees]

Red-Black Invariants

1. Each node red or black
2. Root is black
3. No 2 reds in a row
   [red node => only black children]
4. Every root-NULL path has same number of black nodes

Like in an
unsuccessful search

Example #1

Claim : a chain of length 3 cannot be a red-black tree

Proof

Example #2

Height Guarantee

Claim : every red-black tree with n nodes has height $\leq 2\log_2(n+1)$

Proof : Observation : if every root-NUL path has >= k nodes, then tree includes (at the top) a perfectly balanced search tree of depth k-1.

=> Size n of the tree must Be at least $2^k - 1$

[k = 3]

Height Guarantee (con'd)

Story so far : size $n \geq 2^k - 1$, where k = minimum # of nodes on root – NULL path

=> $k \leq log_2(n+1)$

Thus : in a red-black tree with n nodes, there is a root-NULL path with at most $\log_2(n+1)$ black nodes.

By 4th Invariant : every root-NULL path has $\leq \log_2(n+1)$ black nodes

By 3rd Invariant : every root-NULL path has $\leq 2\log_2(n+1)$ total nodes.

Q.E.D.

Which of the search tree operations have to be re-implemented so that the Red-Black invariants are maintained?

- ○ Search
- ○ Delete
- ○ Insert and Delete
- ○ None of the above

Data Structures

Insertion In A Red-Black Tree

Design and Analysis of Algorithms I

High-Level Plan

Idea for Insert/Delete: Proceed as in a normal binary search tree, then recolor and/or perform rotations until invariants are restored.

Insert(x):

1. Insert x as usual (makes x a leaf).
2. Try coloring x red.
3. If x's parent y is black, done.
4. Else y is red $\Rightarrow$ y has a black parent w.

Insertion

Case 1

Case 1: The other child z of x's grandparent w is also red.
$\Rightarrow$ Recolor y, z black and w red. [key point: does not break invariant (4)]
$\Rightarrow$ Either restores invariant (3) or propagates the double red upward.
$\Rightarrow$ Can only happen $O(\log n)$ times. [If you reach the root, recolor it black $\Rightarrow$ Preserves invariant (4)].

Case 2

Case 2: Let x, y be the current double-red, x the deeper node. Let $w = x$'s grandparent. Suppose w's other child is NULL or is a black node z.

Exercise/case analysis (details omitted): Can eliminate double-red [$\Rightarrow$ All invariants satisfied] in $O(1)$ time via 2-3 rotations+ recolorings.

Data Structures

Rotations

Design and Analysis
of Algorithms I

Left Rotations

Key primitive: *Rotations*. (Common to all balanced search tree implementations: red-black, AVL, B+, etc.)

Idea: Locally rebalance subtrees at a node in $O(1)$ time.

Left rotation: (of a parent x and right child y)

Right Rotations

Right rotation:

Nice properties: Search tree property maintain, can implement in $O(1)$ time.