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 !

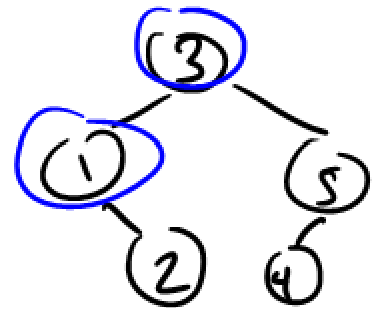
OPERATIONS	RUNNING TIME	
SEARCH	θ(log(n))	
SELECT	O(log(n))	Up from O(1)
MIN/MAX	O(log(n))	
PRED/SUCC	O(log(n))	
RANK	O(log(n))	
OUTPUT IN SORTED ORDER	O(n)	
INSERT	O(log(n))	new
DELETE	O(log(n))	

Also supported by hash tables

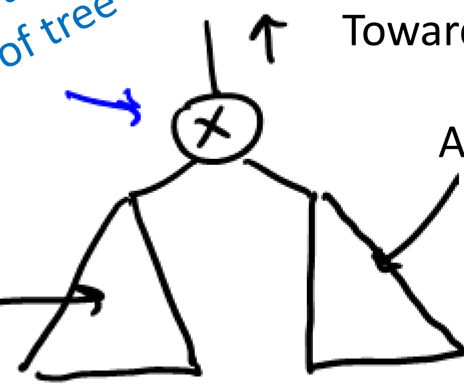
Also supported by heaps

Binary Search Tree Structure

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

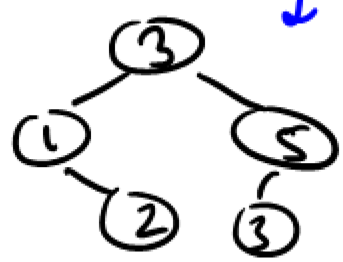


SEARCH TREE PROPERTY :
(should hold at every node of the search tree)



The Height of a BST

Height = 2



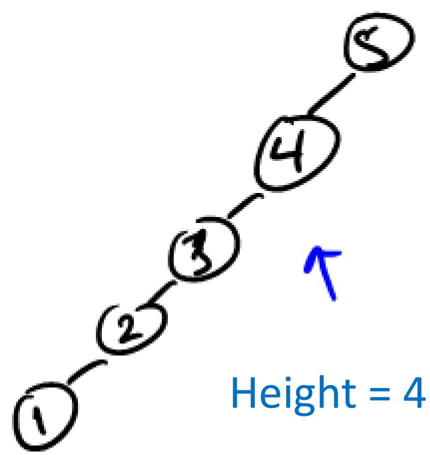
Note : many possible trees for a set of keys.

(aka depth) longest root-leaf path

Note : $\boxed{\text{height}}$ could be anywhere from $\boxed{\sim \log_2 n}$ to $\boxed{\sim n}$

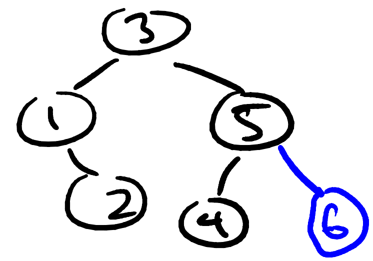
Worst case, a chain

Best case, perfectly balanced



Height = 4

Searching and Inserting



<u>To Search for key k in tree T</u>

-- 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

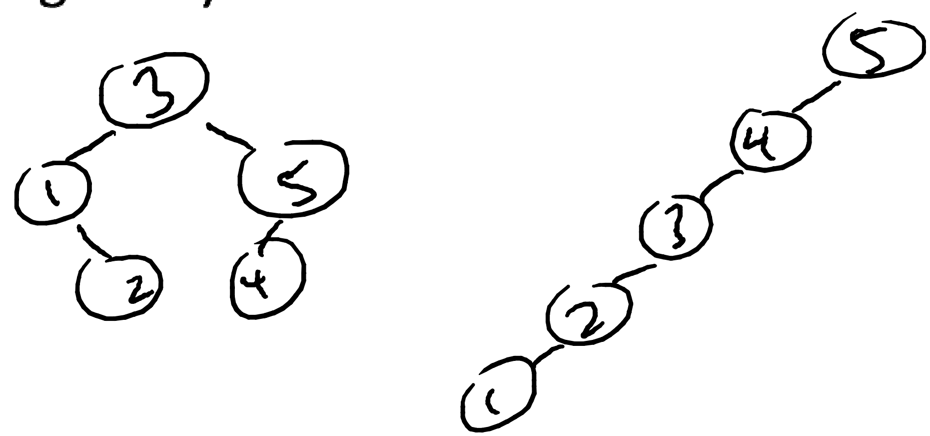
<u>To Insert a new key k into a tree T</u>

<u>Exercise</u> : preserves search tree property!

-- search for k (unsuccessfully)

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

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

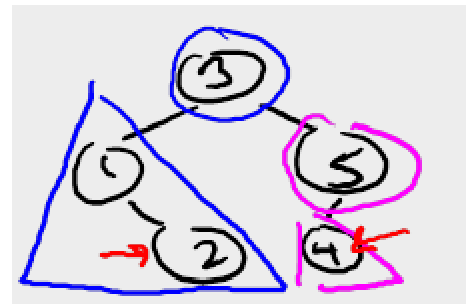


- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

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.

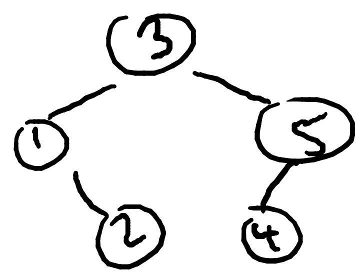
Happens first time you "turn left"

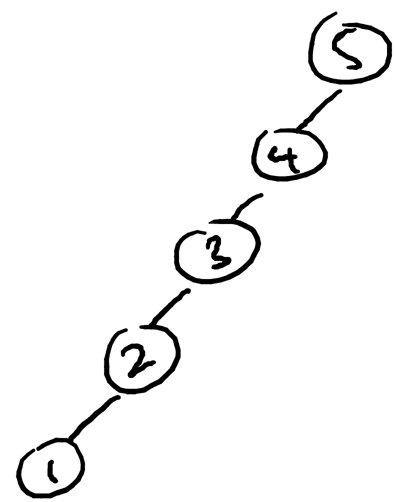
Exercise : prove this works

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

<u>TO PRINT OUT KEYS IN INCREASING ORDER</u>

-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]

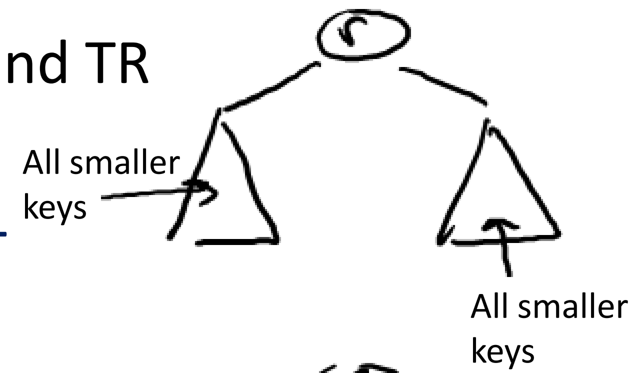
-Print out r's key

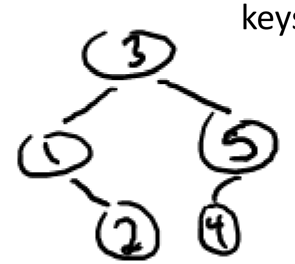
-Recurse on TR

 [prints out keys of TR in increasing order]

<u>RUNNING TIME</u>

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





Deletion

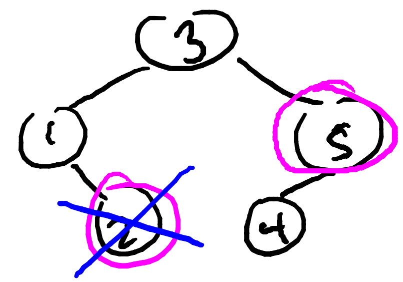
TO DELETE A KEY K FROM A SEARCH TREE

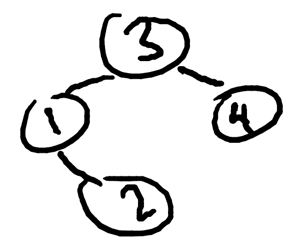
- SEARCH for k

EASY CASE (k's node has no children)

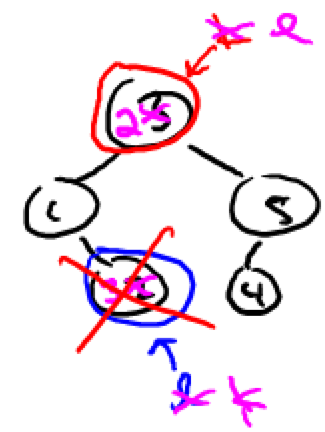
-Just delete k's node from tree, done

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 !

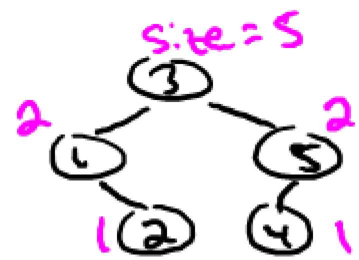
NOTE : in it's new position, k has no right child !
 => easy to delete or splice out k's new node

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

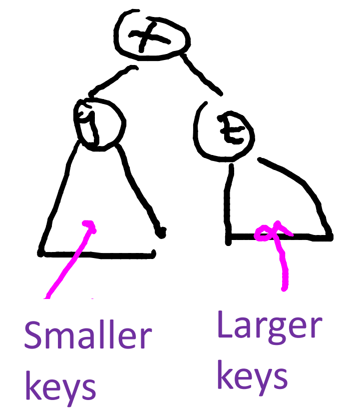
x itself

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

Select and Rank (con'd)

HOW TO SELECT Ith 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 ith 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 ?