Design and Analysis of Algorithms I

Data Structures

Binary Search Tree Basics

Balanced Search Trees: Supported Operations

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

OPERATION	RUNNING TIME
SEARCH	$O(\log n)$
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

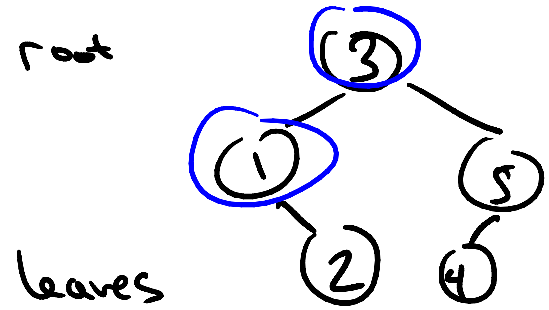
also supported by heaps

up from $O(1)$

new

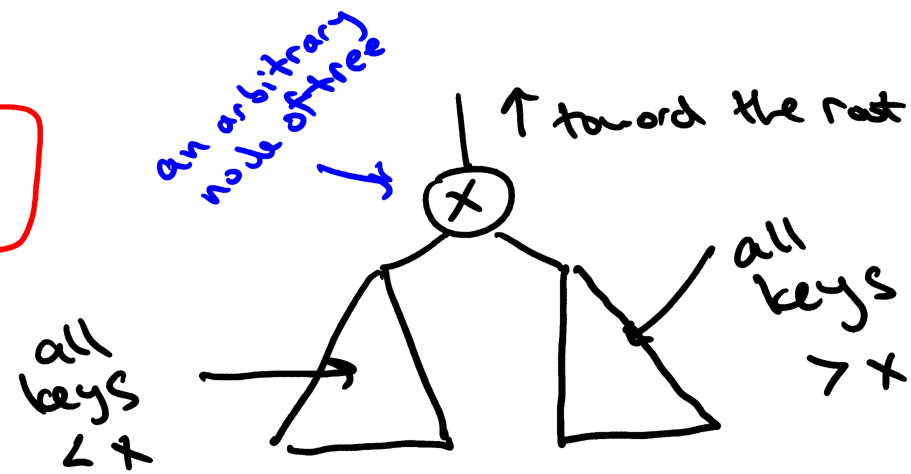
Binary Search Tree Structure

- 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)



The Height of a BST

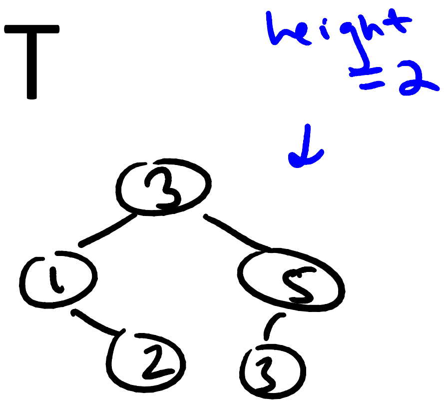
Note: many possible trees for a set of keys.

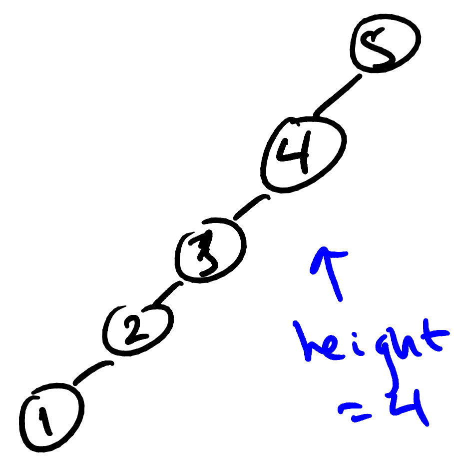
Note: height could be anywhere from $\approx \log_2 n$ to $\approx n$.

(aka depth) longest root-leaf path

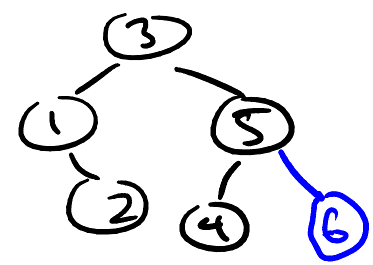
best case, perfectly balanced

worst case, a chain





Searching and Inserting



To SEARCH for key k in tree T

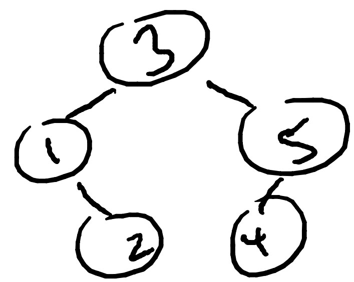
- start at the root
- traverse left/right child pointers as needed
  - left: if k < key at current node
  - right: if k > key at current node
- return node with key k or NULL, as appropriate

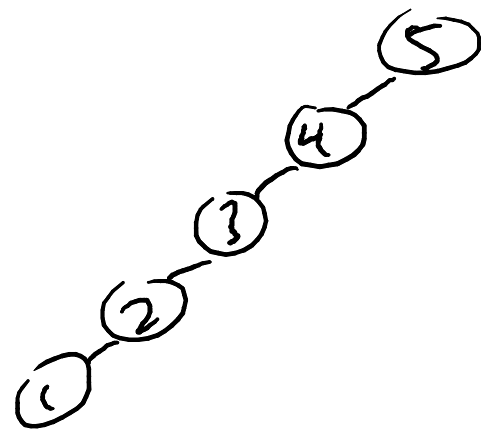
To INSERT a new key k into a tree T

Exercise: preserves search tree property!

- search for k (unsuccessfully)
- rewire final NULL ptr to point to new node with key k

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



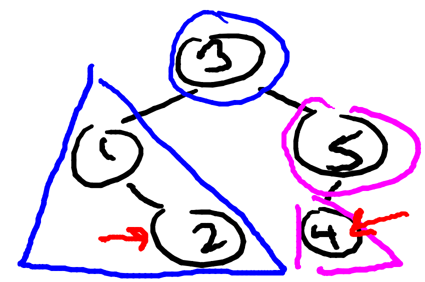


- ○ $\theta(1)$
- ○ $\theta(\log_2 n)$
- ○ $\theta(height)$
- ○ $\theta(n)$

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) until you can't anymore (return last key found)

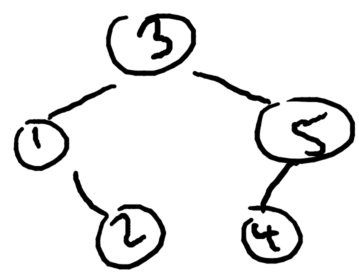


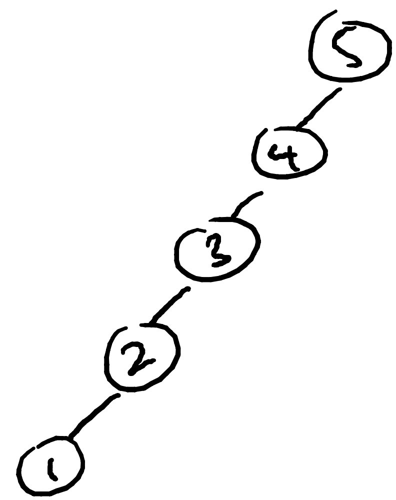
TO COMPUTE THE PREDECESSOR OF KEY k

- 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 ("turn left" happens first time you)

exercise: prove this works!

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



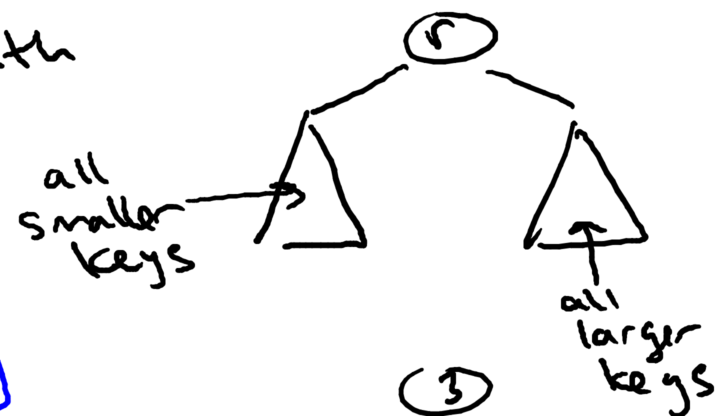


- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

In-Order Traversal

TO PRINT OUT KEYS IN INCREASING ORDER

- let r = root of search tree, with subtrees T_L and T_R



- recurse on T_L

  [by recursion/induction, prints out keys of T_L in increasing order]

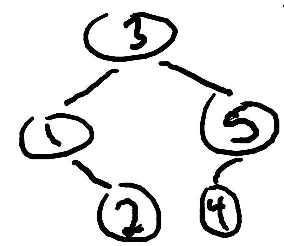
- print out r's key

- recurse on T_R

  [prints out keys of T_R in increasing order]

RUNNING TIME

$O(1)$ time, n recursive calls $\Rightarrow$ $O(n)$ total

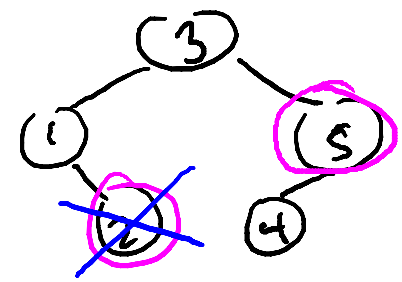


Tim Roughgarden

Deletion

TO DELETE A KEY k FROM A SEARCH TREE

- SEARCH for k

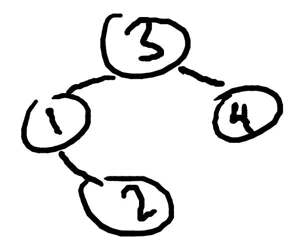


EASY CASE (k's node has no children)

- just delete k's node from tree, done

MEDIUM CASE (k's node has one child)

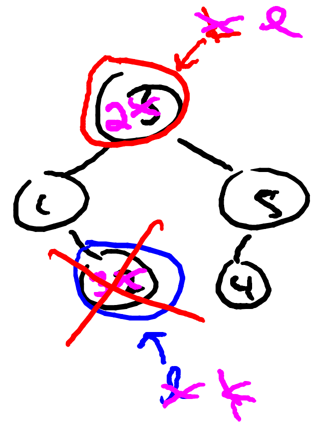
- just "splice out" k's node
  (unique child assumes position previously held by k's node)



Tim Roughgarden

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 its new position, k has no right child!
⇒ easy to delete or splice out k's new node

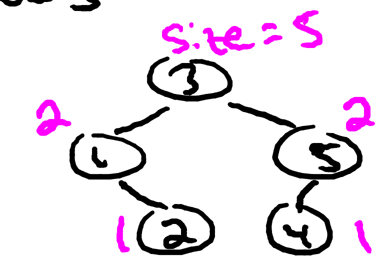
Exercise: at end, have a valid search tree!

RUNNING TIME: $\Theta(\text{height})$

Select and Rank

Idea: store a little bit of extra info at each tree node <u>about the tree itself</u> (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 $\underbrace{\text{size}(y)}_{\text{population in left subtree}} + \underbrace{\text{size}(z)}_{\text{right subtree}} + \underbrace{1}_{x \text{ itself}}$

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

Tim Roughgarden

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?]