Design and Analysis of Algorithms I

Data Structures

Binary Search Tree Basics

BST are designed so that you can search easily. Heaps are designed so that you can find min/max easily
That's exactly the "invariant" each node in a BST/heap aims to enable
You can think of your own DS trying to achieve some objective motivated by this

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))
MIN/MAX		O(log(n))
PRED/SUCC	given pointer to element, find the next or previous	O(log(n))
RANK	how many keys are > given value. Like binary search	O(log(n))
OUTPUT IN SORTED ORDER		O(n)
INSERT	Insert and delete are O(n) time in sorted array	O(log(n))
DELETE		O(log(n))

Up from O(1)

new

Also supported by hash tables

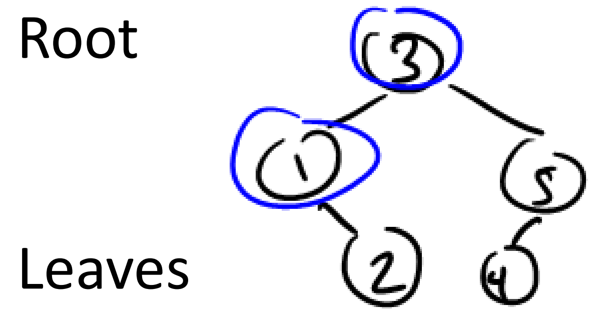
Also supported by heaps

heap has better constant factors in time and space complexity wrt BST. also heap only does min or max but not both

hashing has better constant factors wrt BST. if only insert, delete, search needed, BST is overkill; use hash (similar argument for heap above)

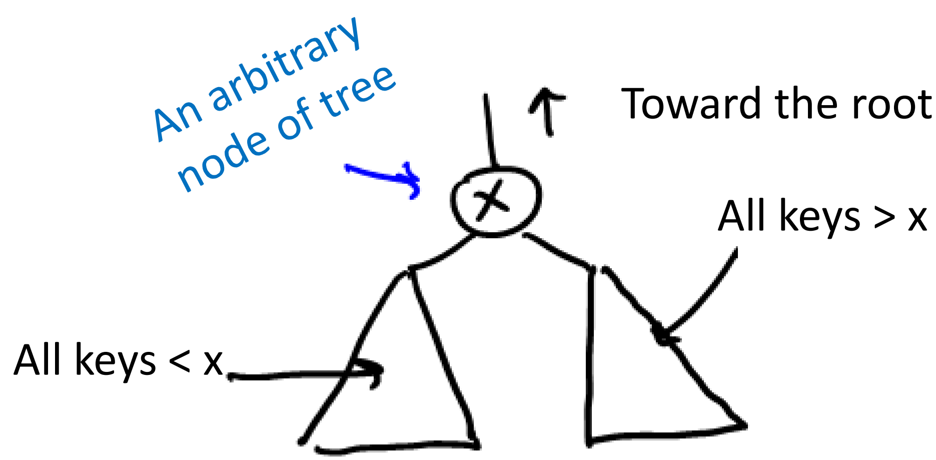
Binary Search Tree Structure

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



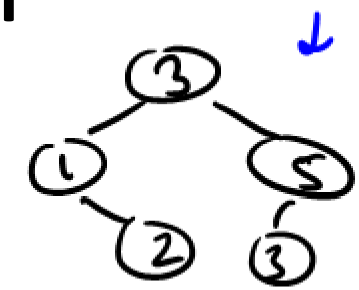
SEARCH TREE PROPERTY :
(should hold at every node of the search tree)

The above assumes all keys are distinct
To handle duplicates, slightly modify your convention.
Let the left subtree hold “all keys <=x“.
Everything that follows shall be true for it



The Height of a BST

Height = 2



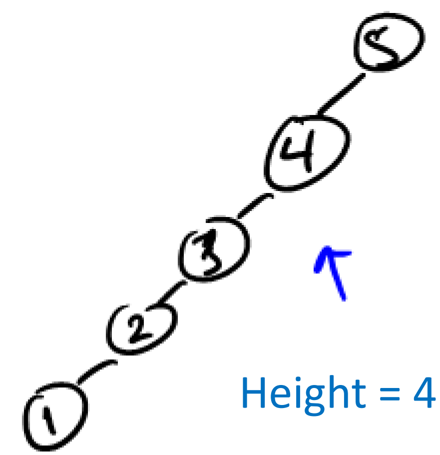
Note : many possible trees for a set of keys.

(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

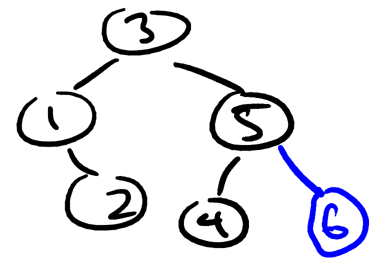
Best case, perfectly balanced



Height = 4

Tim Roughgarden

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!

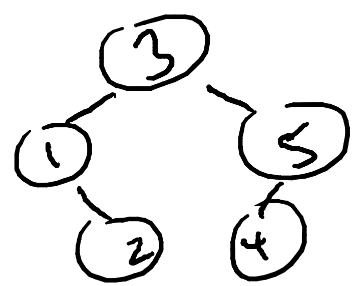
-- search for k (unsuccessfully)

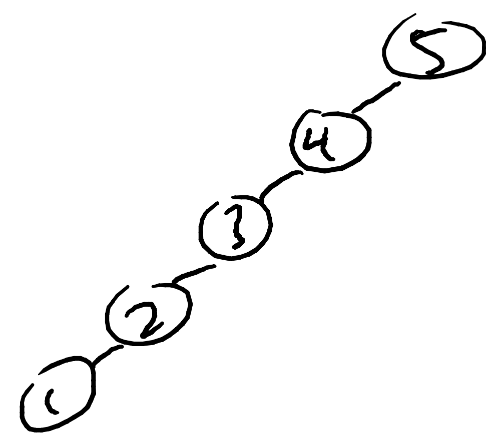
-- rewire final NULL ptr to point to new node with key k

Tim Roughgarden

can write some additional case that takes care of duplicates

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



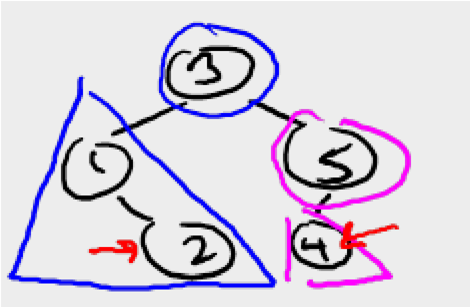


- $\theta(1)$
- $\theta(\log_2 n)$
- $\theta(height)$
- $\theta(n)$

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.

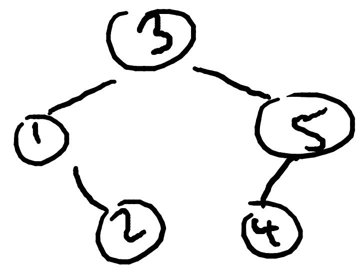
Happens first time you "turn left"

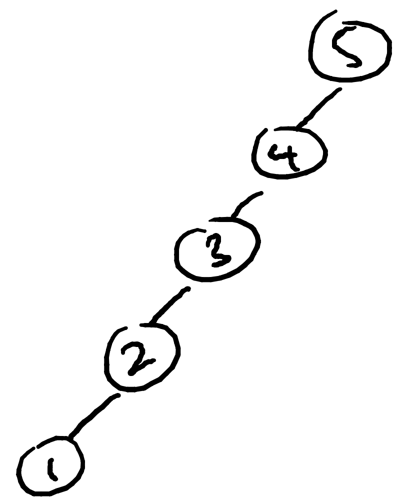
Exercise : prove this works

You will get to key less than k the first time you take left turn
Think: How to get to Pred of 4?
Based on this, what's pred of 1? Parent of 1 never turns left. Thats a property of min element of BST
Based on this, what the Succ of 5? Parent of 5 never turns right. Thats a property of max element of BST

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

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





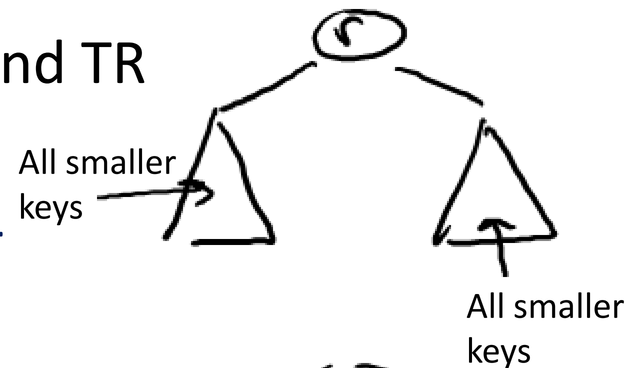
In-Order Traversal

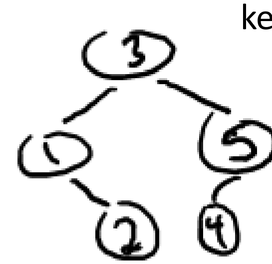
TO PRINT OUT KEYS IN INCREASING ORDER

-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

The magic here is , that the recursive calls are called in the order of “in order traversal” we want





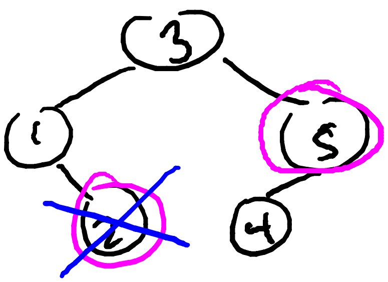
In general, whatever way you modify a tree (any DS), the BST (DS) property must be maintained
Here, BST property is that for every node; everything to the left is smaller than the node, and and everything to the right is bigger than the node

Deletion

TO DELETE A KEY K FROM A SEARCH TREE 3 cases

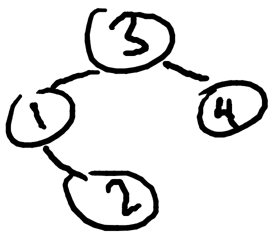
- SEARCH for k

EASY CASE (k's node has no children)



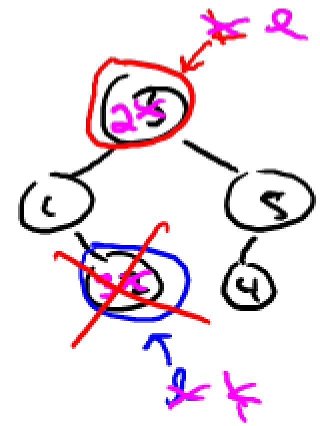
-Just delete k's node from tree, done

MEDIUM CASE (k's node has one child)
(unique child assumes position
previously held by k's node)



If you bring in 1 form left or 5 from right, it won't work. Think !
Why was it easier in heap?
Coz the BST invariant is much stronger, and need to work harder to maintain it

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

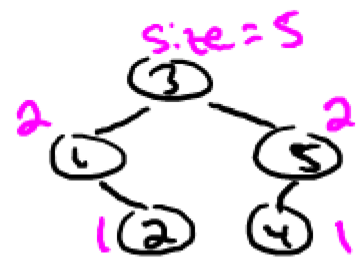
it may have a left child; but that also is the easy case we say on last slide

RUNNING TIME : θ(height)

Exercise : at end, have a valid search tree !

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 = size (x)

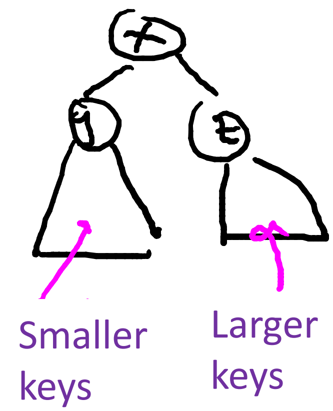
Population in left subtree   Right subtree   x itself

Also : easy to keep sizes up-to-date during an Insertion or Deletion (you check!) → Not so in red black trees.
Keeping up to date when we modify red black tree is hard

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 a>=i
- 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 ?