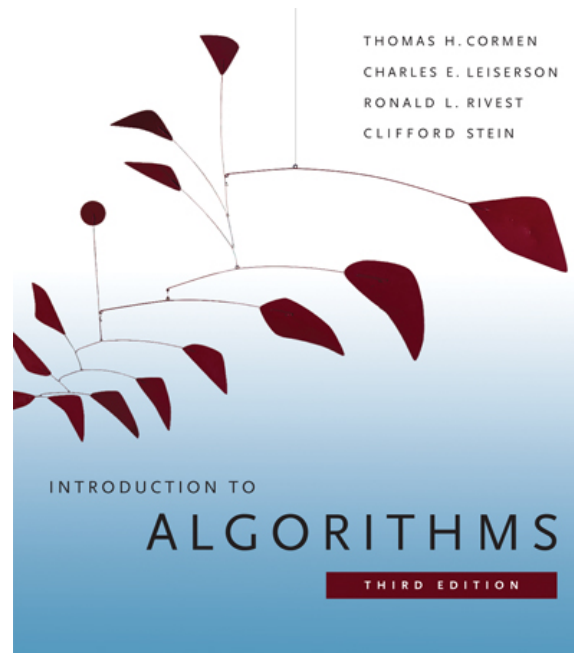


6.006- *Introduction to Algorithms*



Lecture 3

Prof. Costis Daskalakis

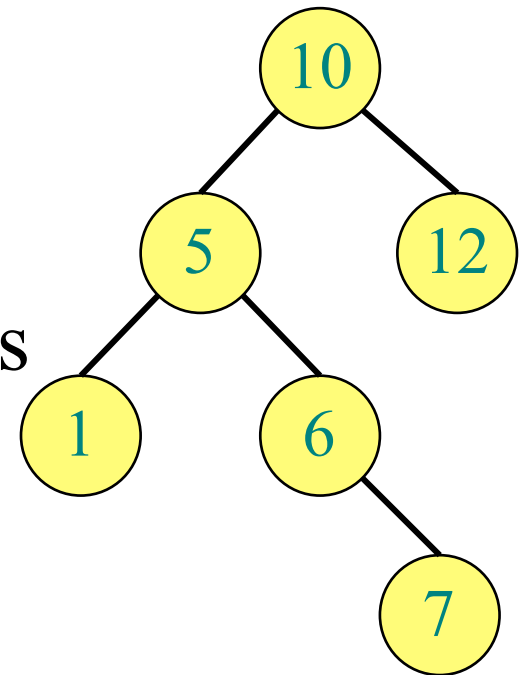
Overview

- Runway reservation system:
 - Definition
 - How to solve with linked-lists
- Binary Search Trees
 - Operations
- Next time: Balanced Search Trees

Readings: CLRS 10, 12.1-3



<http://izismile.com/tags/Gibraltar/>



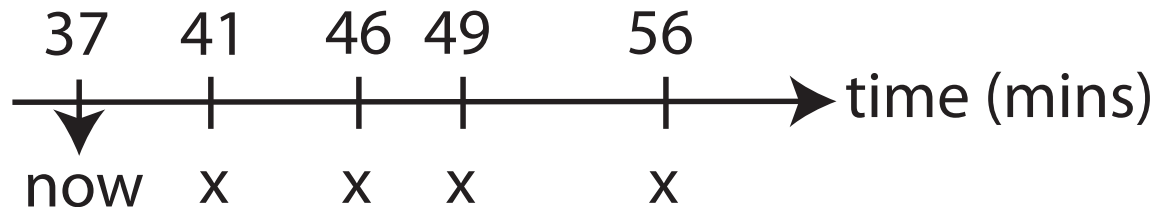
Runway reservation system

- Problem definition:
 - Single (**busy**) runway
 - Reservations for landings
 - maintain a set of future landing times
 - a new request to land at time **t**
 - add **t** to the set if no other landings are scheduled within < 3 minutes from **t**
 - when a plane lands, removed from the set



Runway reservation system

- Example



- $R = (41, 46, 49, 56)$
- requests for time:
 - 44 \Rightarrow reject (46 in R)
 - 53 \Rightarrow ok
 - 20 \Rightarrow not allowed (already past)
- Ideas for efficient implementation ?

Proposed algorithm

- (keep R as a linked-list)

init: R = []

req(**t**): if $t < \text{now}$: return "error"

for i in range (len(R)):

if $\text{abs}(t - R[i]) < 3$: return "runway busy"

R.append(t)

- Complexity?
- Can we do better?

Some other options:

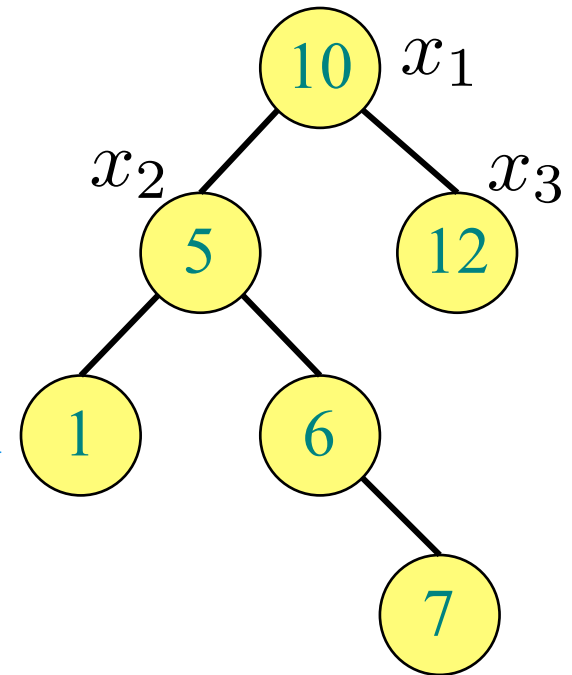
- Keep R as a sorted list:
 - on request t , it takes linear time to find the right location in the list where t needs to be inserted
 - before inserting t at found location check whether the numbers on the left and right of the location are $\leq t-3$ and $\geq t+3$ respectively
- Keep R as a sorted array:
 - takes $O(\log n)$ to find the place to insert new t
 - but still requires linear time to actually insert (requires shifting of elements)

Need best of both worlds:

fast insertion into sorted list

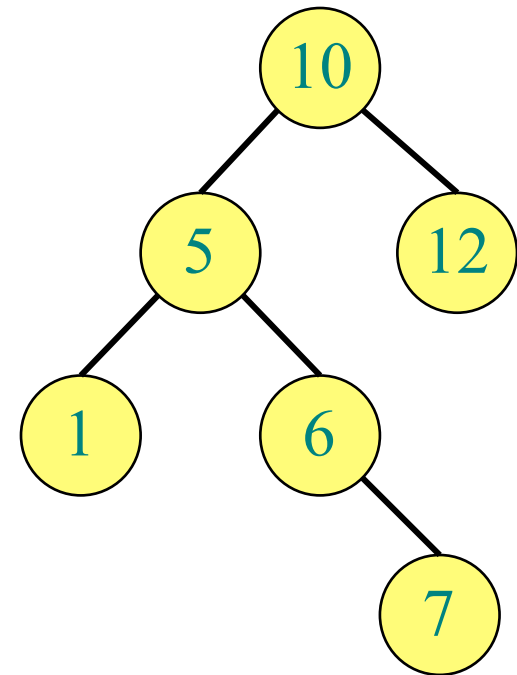
Binary Search Trees (BSTs)

- A tree ...
- ...where each node x has:
 - a $\text{key}[x]$
 - three pointers:
 - $\text{left}[x]$: points to left child
 - $\text{right}[x]$: points to right child
 - $\text{p}[x]$: points to parent
- E.g. $\text{key}[x_1]=10$
- $\text{left}[x_1]=x_2$
- $\text{p}[x_2]=x_1$
- $\text{p}[x_1]=\text{NIL}$



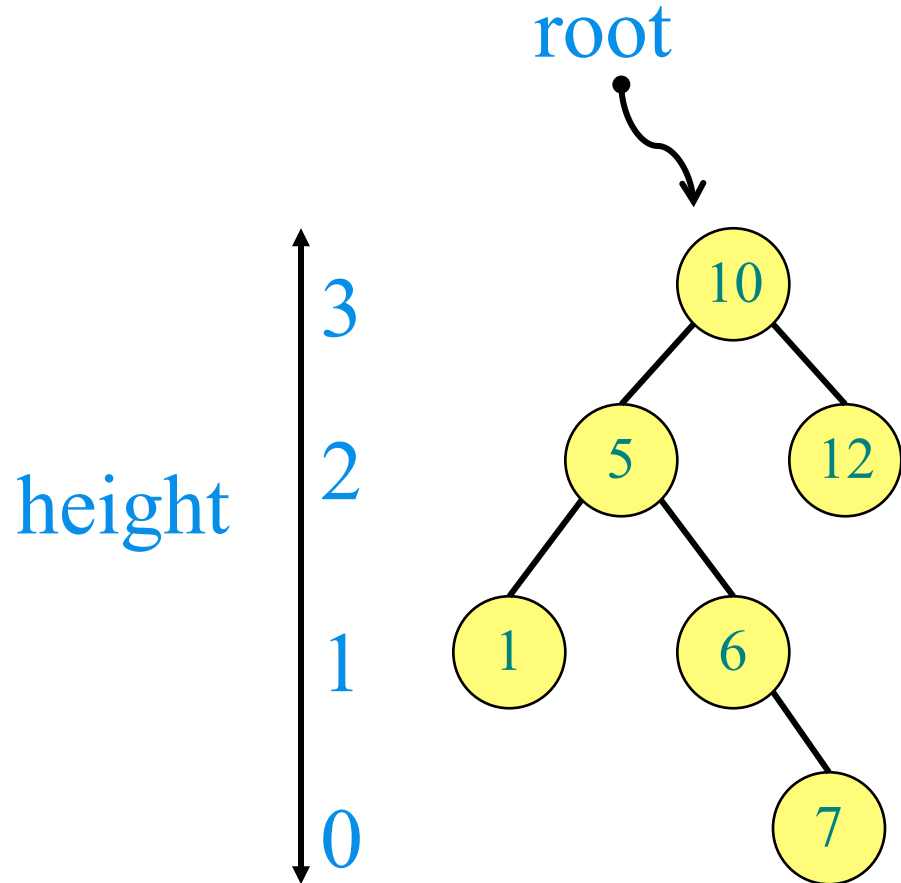
Binary Search Trees (BSTs)

- **Defining** property (i.e. what makes it a binary SEARCH tree):
- for any node x :
 - for all nodes y in the **left** subtree of x :
$$\text{key}[y] \leq \text{key}[x]$$
 - for all nodes y in the **right** subtree of x :
$$\text{key}[y] \geq \text{key}[x]$$
- How are BSTs created?



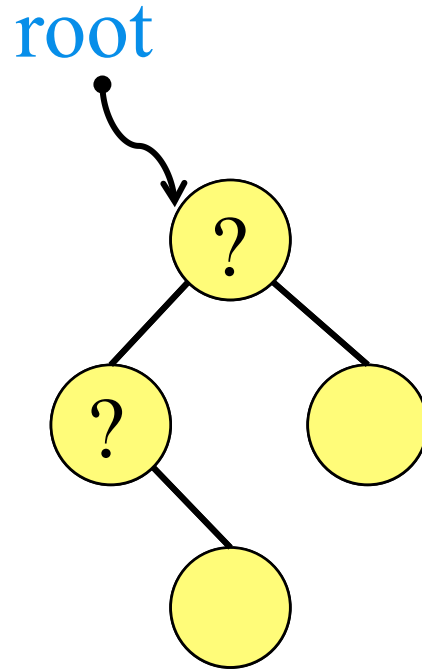
Growing BSTs

- Insert 10
- Insert 12
- Insert 5
- Insert 1
- Insert 6
- Insert 7



BST as a data structure

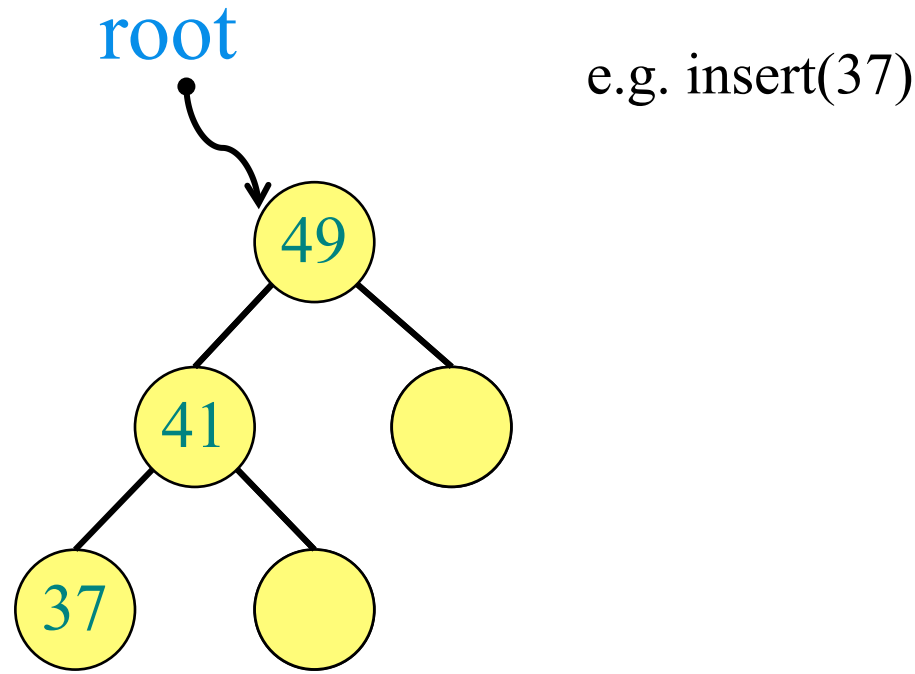
- Supported Operations:
 - insert(**k**): insert a node with key **k** at the appropriate location of the tree



e.g. insert(37)

BST as a data structure

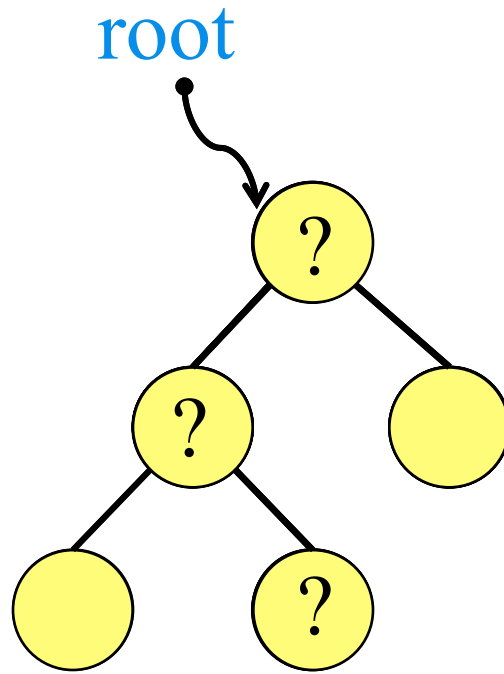
- Supported Operations:
 - insert(**k**): insert a node with key **k** at the appropriate location of the tree



Aside: Can do the “within 3” check for reservation system during insertion.

BST as a data structure

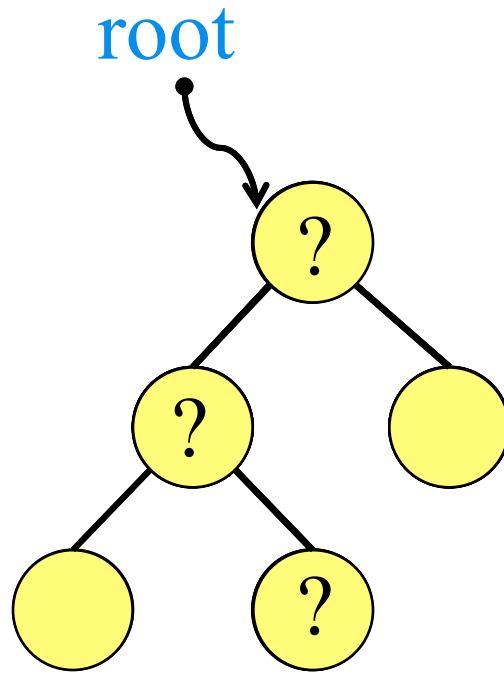
- Supported Operations:
 - find(**k**): finds the node containing key **k** (if it exists)



e.g. find(46)

BST as a data structure

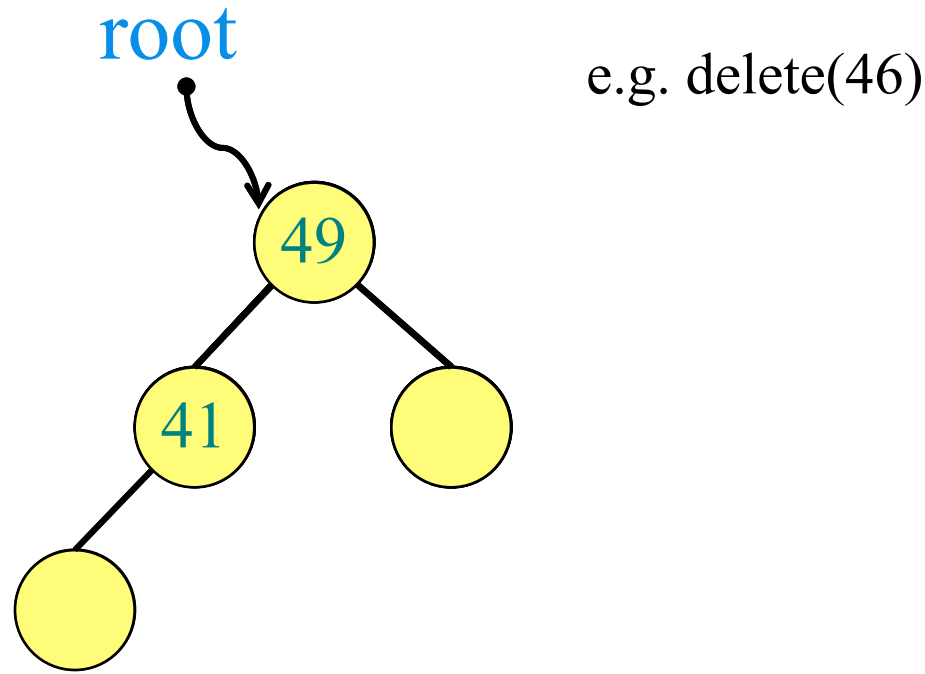
- Supported Operations:
 - delete(**k**): delete the node containing key **k**, if such a node exists



e.g. delete(46)

BST as a data structure

- Supported Operations:
 - delete(**k**): delete the node containing key **k**, if such a node exists



Question: What if we have to delete a node that is internal?
How do we fill in the hole? A: next lecture.

BST as a data structure

- Supported Operations:
 - insert(**k**): insert a node with key **k** at the appropriate location of the tree
 - find(**k**): finds the node containing key **k** (if it exists)
 - delete(**k**): delete the node containing key **k**, if such a node exists
 - findmin(**x**): finds the minimum of the tree rooted at **x**
 - deletemin(): finds the minimum of the tree and deletes it
 - next-larger(**x**): finds the node containing the key that is the immediate next of **key[x]**

Next-larger

next-larger(x):

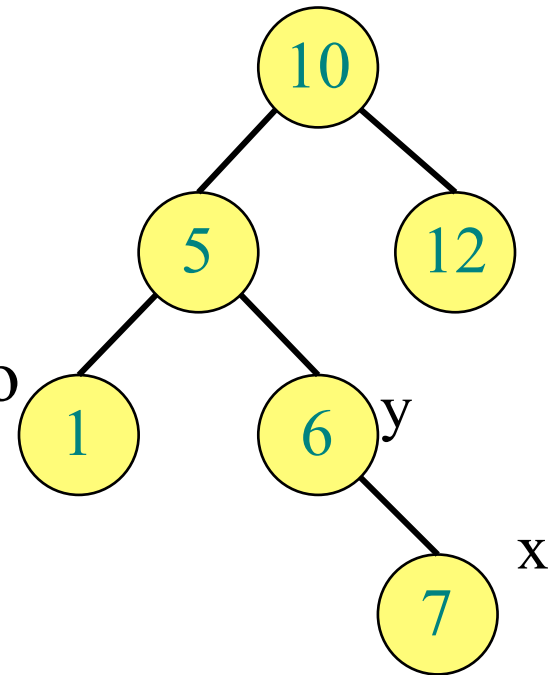
- If $\text{right}[x] \neq \text{NIL}$ then
return $\text{findmin}(\text{right}[x])$
- Otherwise

$y \leftarrow p[x]$

While $y \neq \text{NIL}$ and $x = \text{right}[y]$ do

- $x \leftarrow y$
- $y \leftarrow p[y]$

Return y



next-larger($\textcircled{5}$) = $\textcircled{6}$

next-larger($\textcircled{7}$)

Next-larger

next-larger(x):

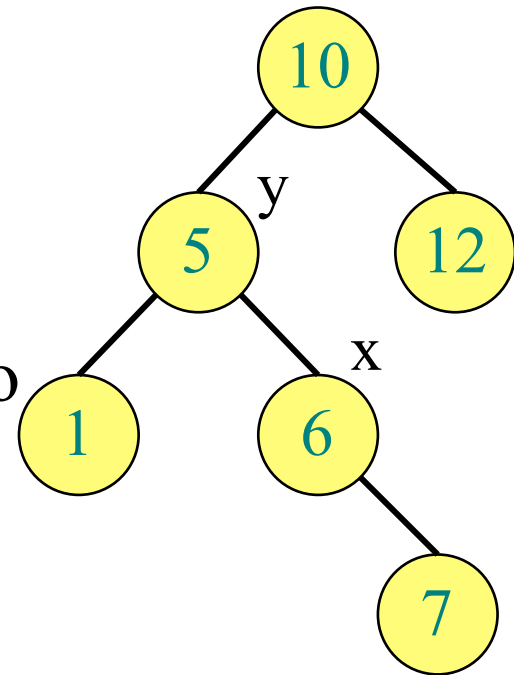
- If $\text{right}[x] \neq \text{NIL}$ then
return $\text{findmin}(\text{right}[x])$
- Otherwise

$y \leftarrow p[x]$

While $y \neq \text{NIL}$ and $x = \text{right}[y]$ do

- $x \leftarrow y$
- $y \leftarrow p[y]$

Return y



next-larger($\textcircled{5}$) = $\textcircled{6}$

next-larger($\textcircled{7}$)

Next-larger

next-larger(x):

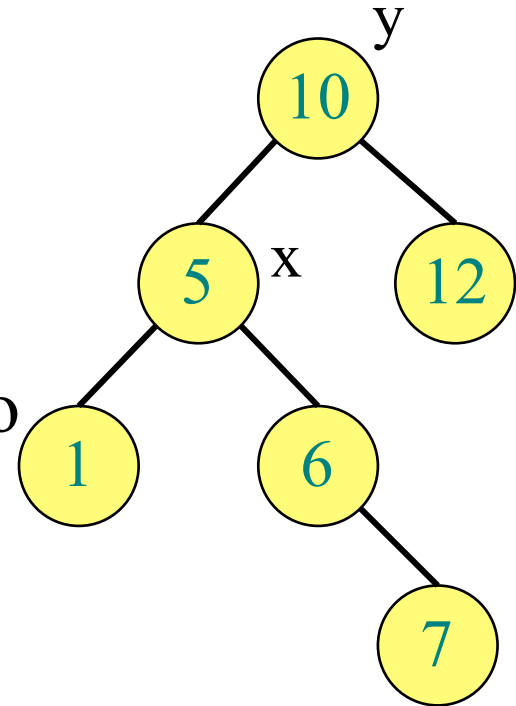
- If $\text{right}[x] \neq \text{NIL}$ then
return $\text{findmin}(\text{right}[x])$
- Otherwise

$y \leftarrow p[x]$

While $y \neq \text{NIL}$ and $x = \text{right}[y]$ do

- $x \leftarrow y$
- $y \leftarrow p[y]$

Return y



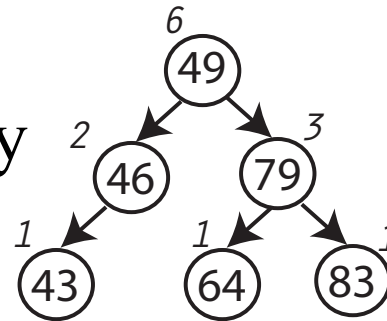
next-larger(5) = 6

next-larger(7) = 10

Back to runway reservation system

- Introducing extra requirements:
e.g. how many planes are
scheduled to land at times $\leq t$?

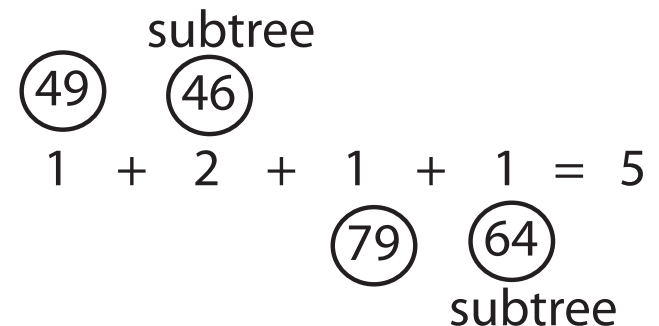
- Augment the BST structure by
keeping track of size of
subtrees rooted at all nodes



- To figure out how many planes
will land $\leq t$:

- Walk down the tree to find where
key t would have been inserted in
the tree...
- ... and for every node where you
forked to the right:
 - add 1 + size of subtree on the
left of that node

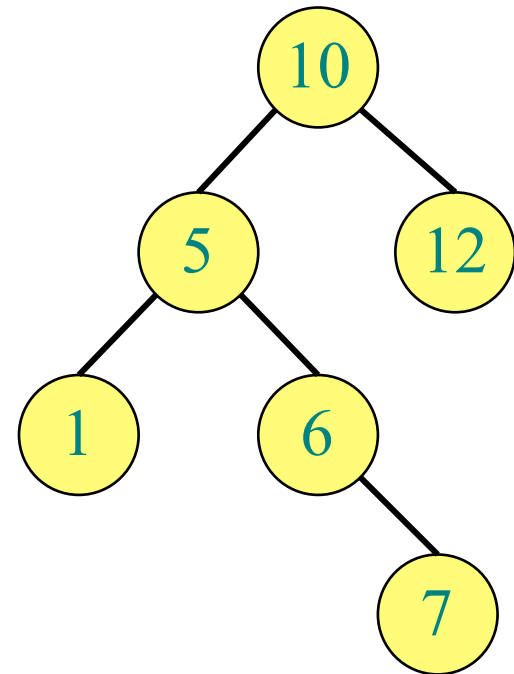
e.g. #planes land ≤ 80 ?



e.g. #planes land ≤ 75 ? A: 4

Analysis

- We have seen insertion, deletion, search, findmin, etc.
- How much time does any of this take ?
- Worst case: $O(\text{height})$
=> height really important
- After we insert n elements, what is the worst possible BST height ?



Analysis

- $n-1$
- so, still $O(n)$ for the runway reservation system operations
- Next lecture: balanced BSTs
- Readings: CLRS 13.1-2
- Hw: notice correction in question 4: a ' $>$ ' was turned to a ' \geq '

