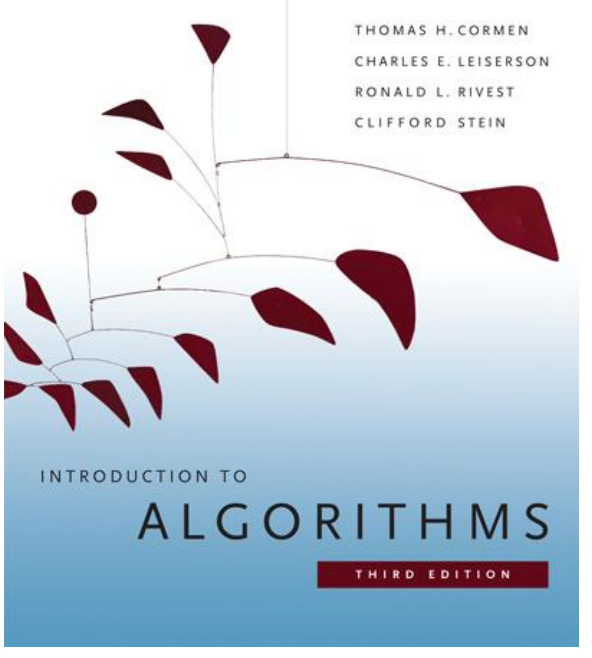
6.006- *Introduction to Algorithms*



Lecture 5

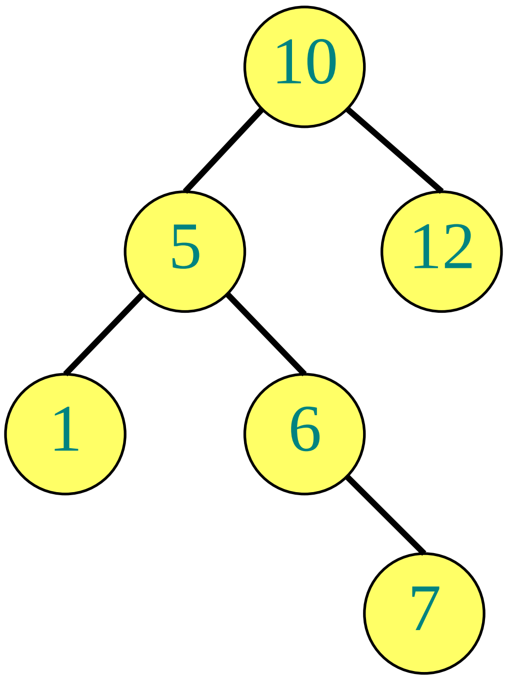
Plan

- Last lecture: Heap
  - Maintains the max element in a dynamically changing data set
- Today: Binary Search Tree
  - Maintains complete ordering of the data
  - Bulding block for:

(a,b) tree, 2-3 tree, 2-3-4 tree, AA tree, AVL tree,

Binary Search Trees (BSTs)

- A tree …
- …where each node x has:
  - a key[x]
  - three pointers:
    - left[x] : points to left child
    - right[x] : points to right child
    - p[x] : points to parent
- Note: no natural array representation, need to use pointers (unlike heaps)

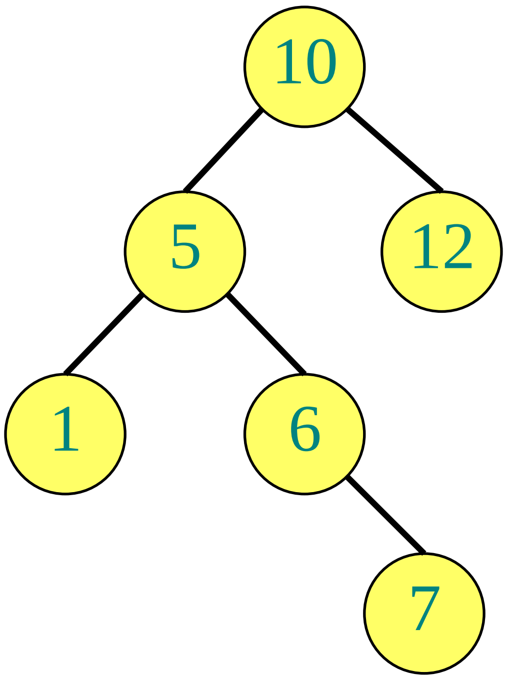


Binary Search Trees (BSTs)

- ***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:

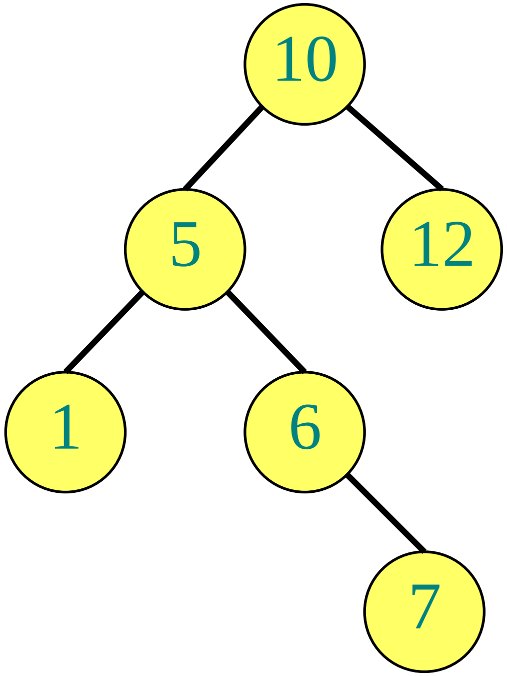
    key[y] ≤ key[x]
  - for all nodes y in the right subtree of x:

    key[y] ≥ key[x]



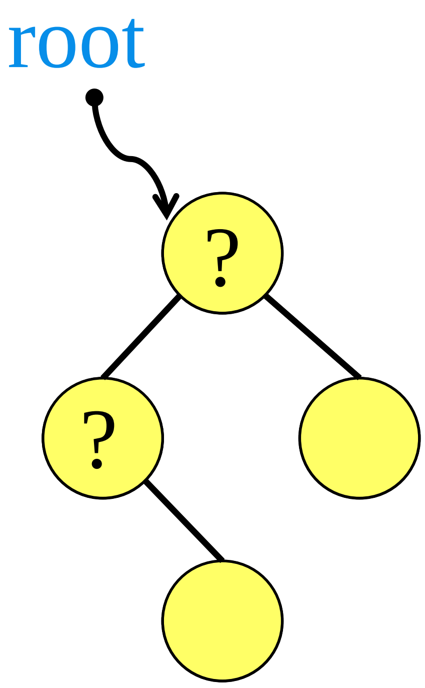
BST as a data structure

- 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]



Insertion

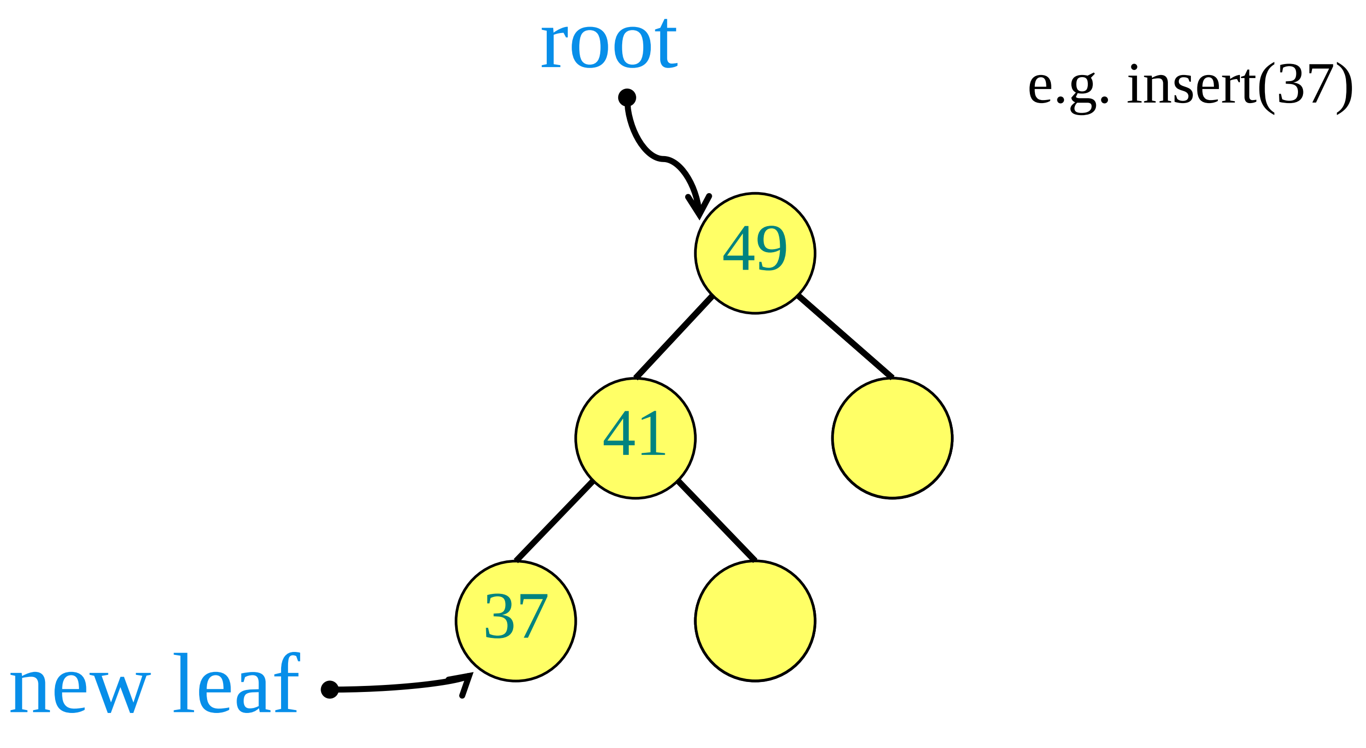
- insert(k): insert a node with key k at the appropriate location of the tree



e.g. insert(37)

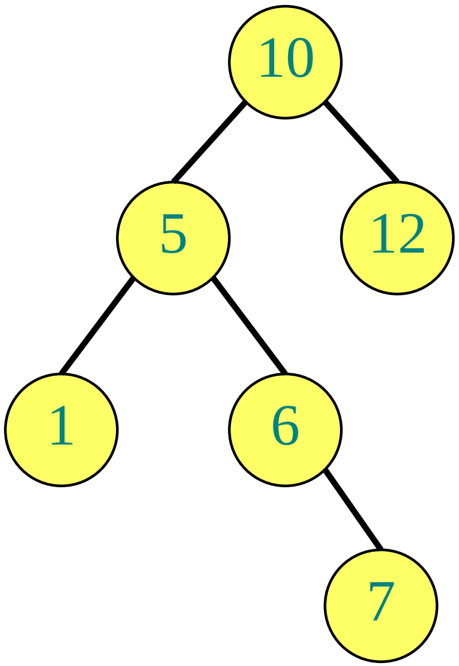
Insertion

- insert(k): insert a leaf node with key k at the unique location of the tree



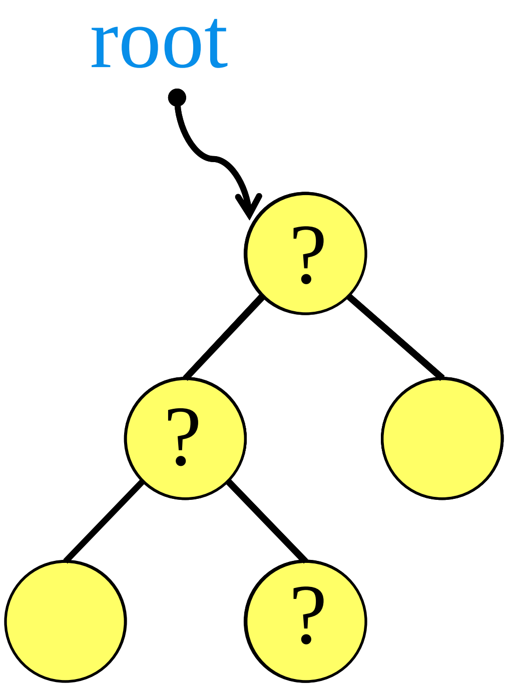
Growing BSTs

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



Find

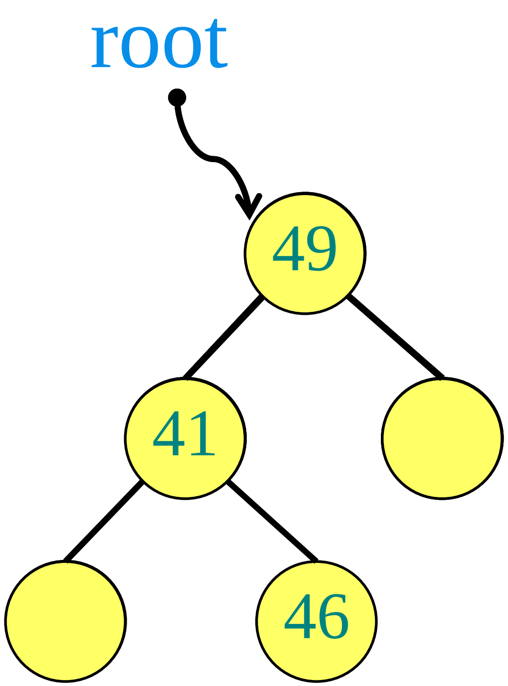
- find(k): finds the node containing key k (if it exists)



e.g. find(46)

Delete

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



e.g. delete(46)

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

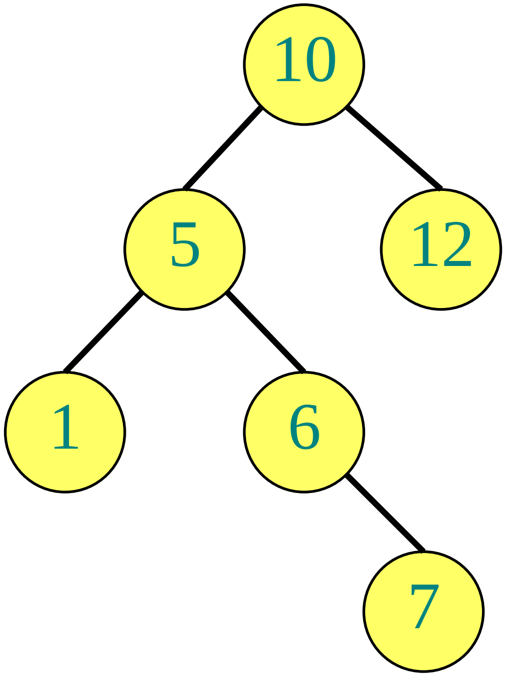
Findmin

Findmin(x)

- While left[x]≠NIL do

  x ← left[x]

- Return x



minimum(5)= 1

Next-larger

next-larger(x):

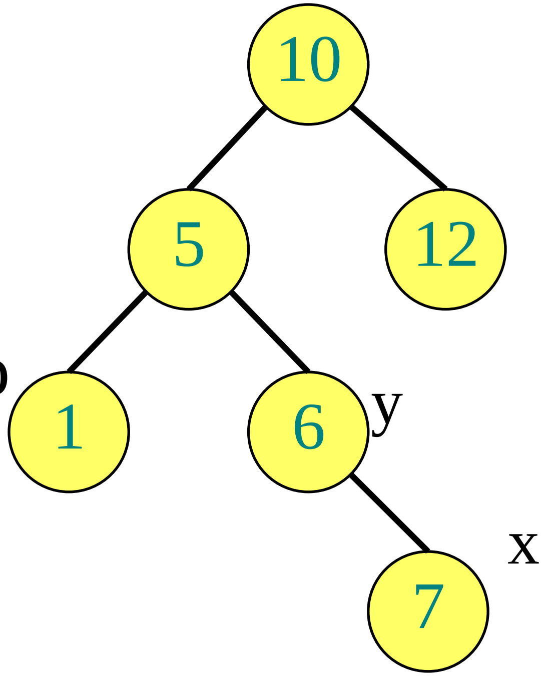
- If right[x] ≠ NIL then
  return findmin(right[x])
- Otherwise

  y ← p[x]

  While y≠NIL and x=right[y] do

  - x ← y
  - y ← p[y]

  Return y



next-larger(5) = 6

next-larger(7)

Next-larger

next-larger(x):

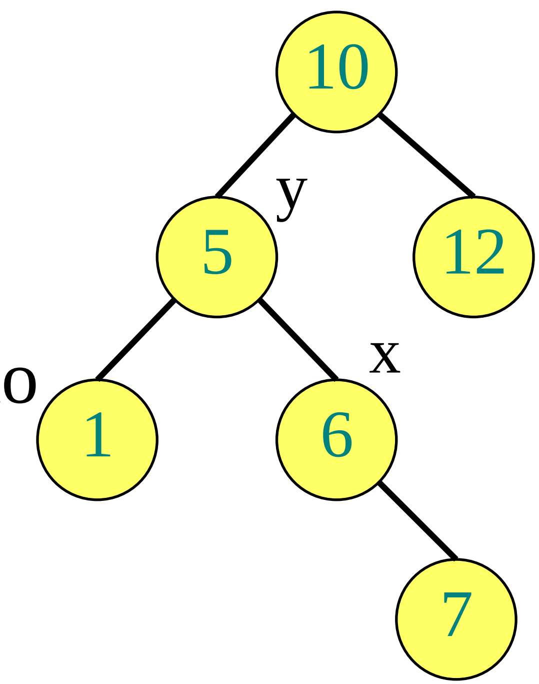
- If $right[x] \neq NIL$ then return findmin($right[x]$)
- Otherwise

  $y \leftarrow p[x]$

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

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

  Return y



next-larger(5) = 6

next-larger(7)

Next-larger

next-larger(x):

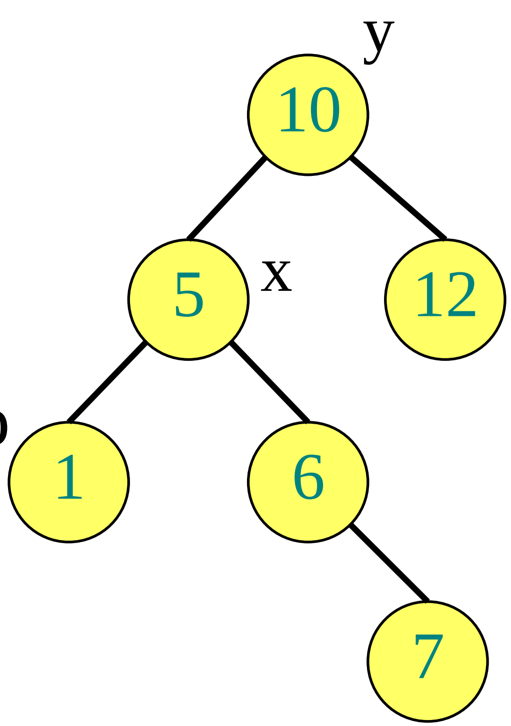
- If right[x] ≠ NIL then return findmin(right[x])
- Otherwise

  y ← p[x]

  While y≠NIL and x=right[y] do

  - x ← y
  - y ← p[y]

  Return y



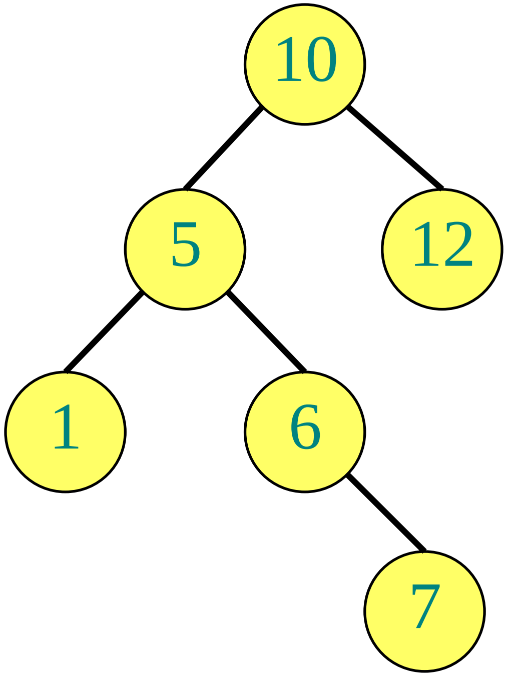
next-larger(5) = 6

next-larger(7) = 10

Analysis

- We have seen insertion, deletion, search, findmin, etc.
- How much time does any of this take ?
- Worst case: O(height)

  => height really important
- After we insert n elements, what is the worst possible BST height ?



Analysis

- n-1
- So, still O(n) per operation
- Next lecture: balanced BSTs