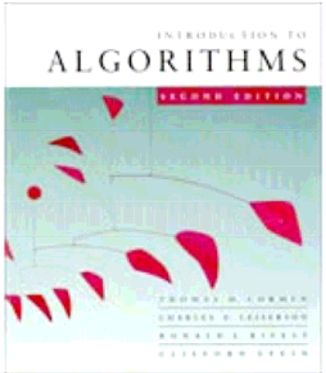
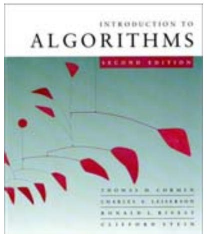


Binary Search Tree

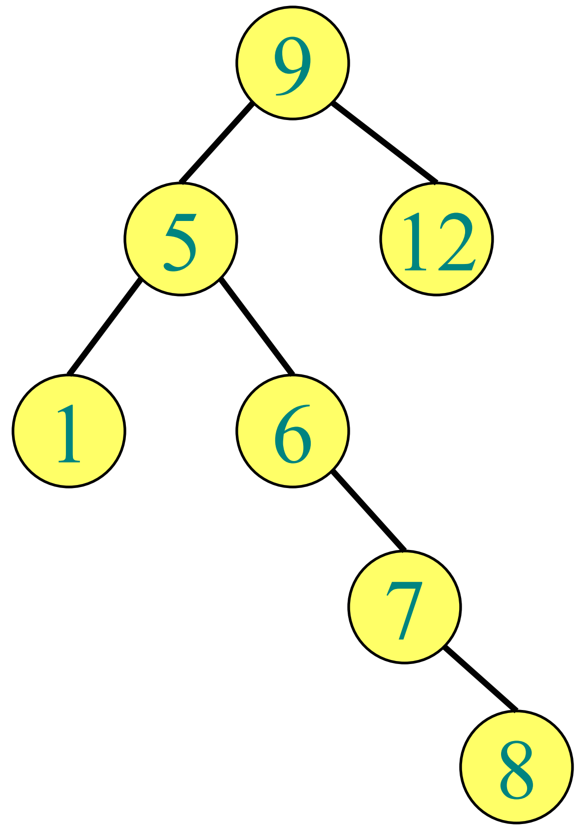
Data structures

- Previous lecture: hash tables
  - Insert, Delete, Search in (expected) constant time
  - Works for integers from $\{0\ldots m^r\text{-}1\}$
- This lecture: Binary Search Trees
  - Insert, Delete, Search (Successor)
  - Works in comparison model



Binary Search Tree

- Each node x has:
  - key[x]
  - Pointers:
    - left[x]
    - right[x]
    - p[x]



Binary Search Tree (BST)

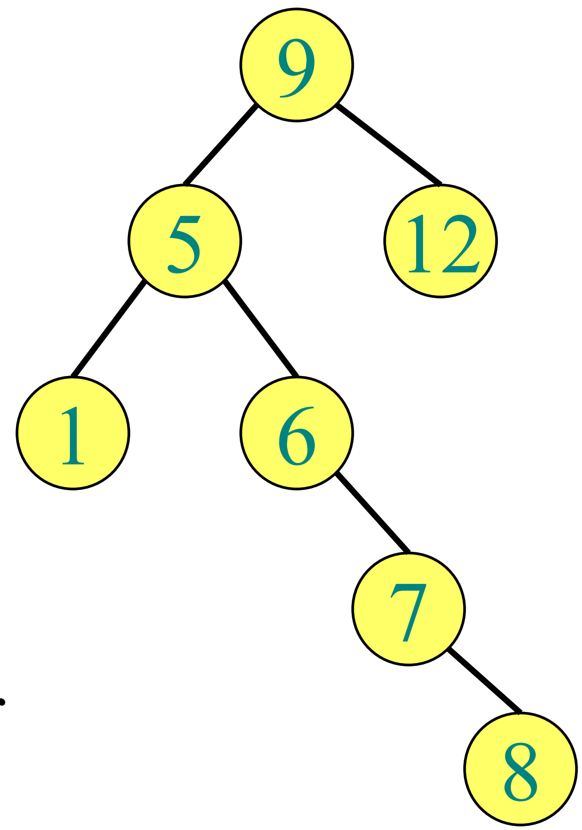
- Property: for any node x:
  - For all nodes y in the left subtree of x:

    $$key[y] \leq key[x]$$

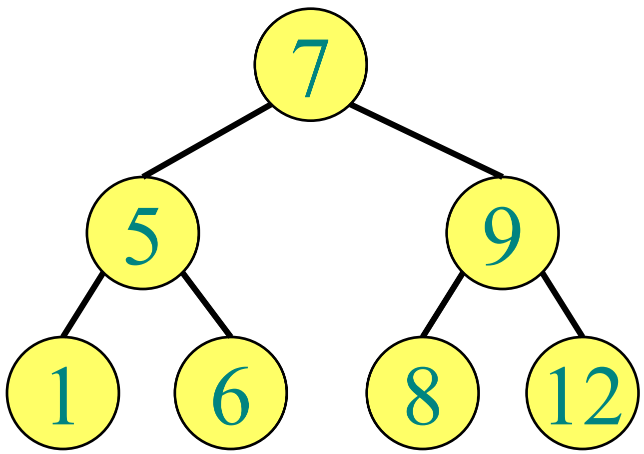
  - For all nodes y in the right subtree of x:

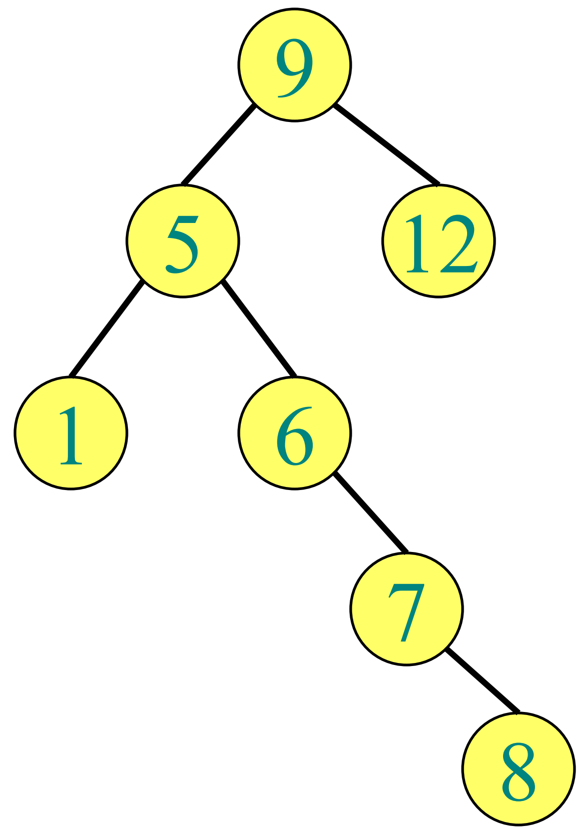
    $$key[y] \geq key[x]$$

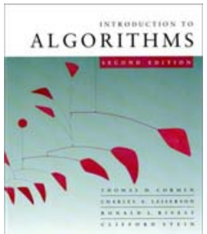
- Given a set of keys, is BST for those keys unique?



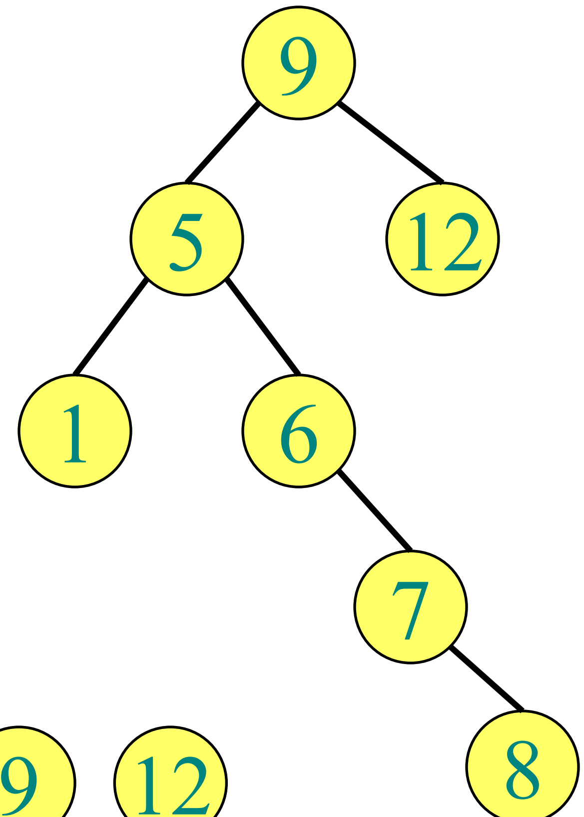
No uniqueness





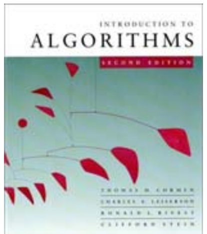


What can we do given BST ?



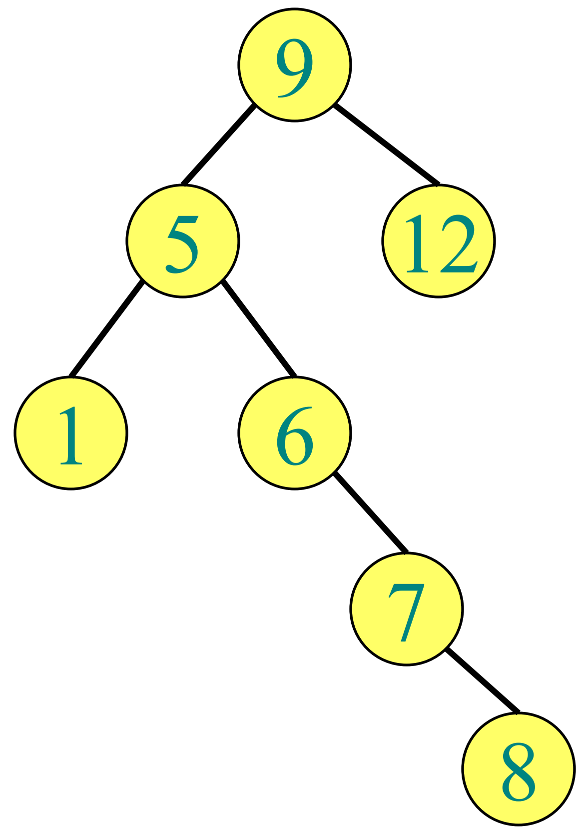
- Sort !
- Inorder-Walk(x):

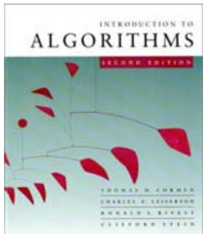
  If x≠NIL then
  - Inorder-Walk(left[x])
  - print key[x]
  - Inorder-Walk(right[x])
- Output: 1 5 6 7 8 9 12



Sorting, ctd.

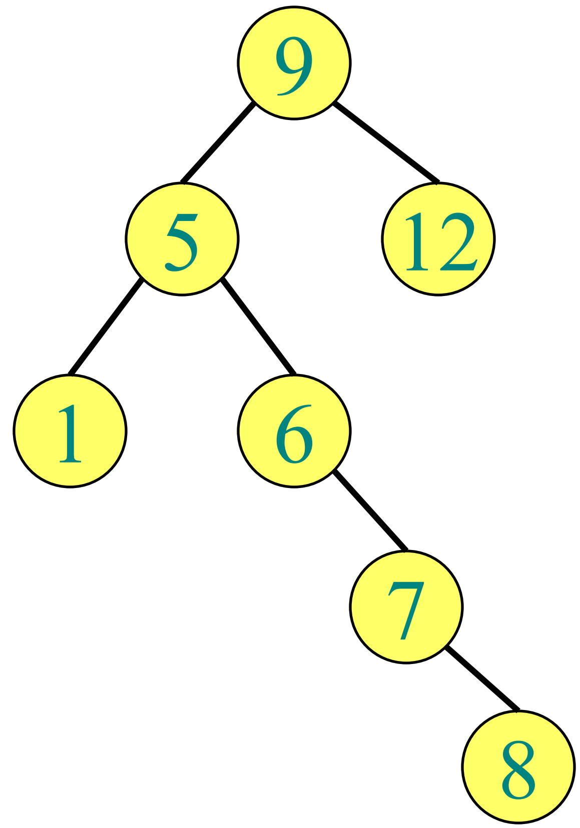
- What is the running time of Inorder-Walk?
- It is O(n)
- Because:
  – Each link is traversed twice
  – There are O(n) links

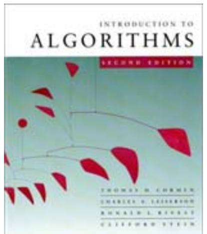




Sorting, ctd.

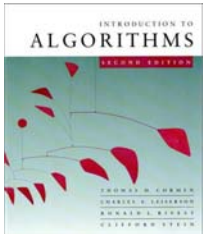
- Does it mean that we can sort n keys in $O(n)$ time ?
- No
- It just means that building a BST takes $\Omega(n \log n)$ time
(in the comparison model)





BST as a data structure

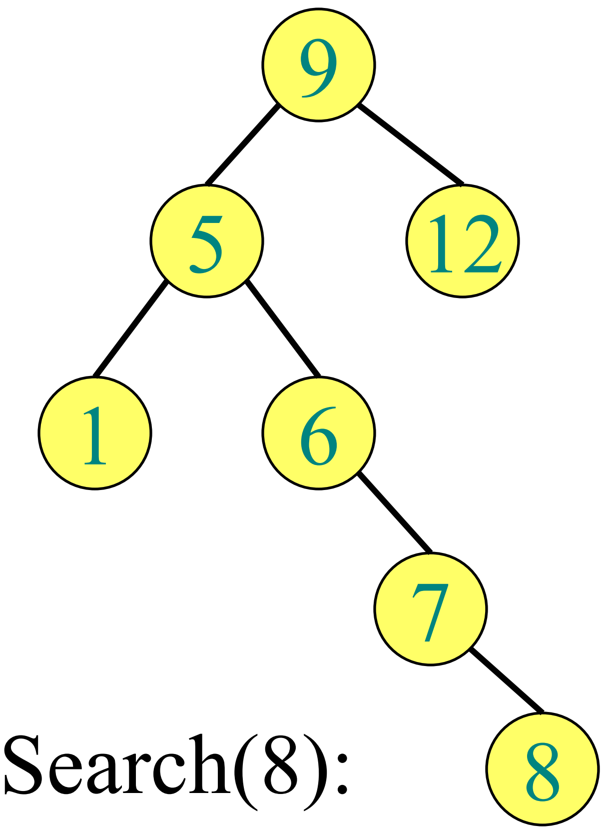
- Operations:
  - Insert(x)
  - Delete(x)
  - Search(k)



Search

Search(x):

- If x≠NIL then
  - If key[x] = k then return x
  - If k < key[x] then return Search(left[x])
  - If k > key[x] then return Search(right[x])
- Else return NIL

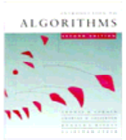


Search(8):

Search(8.5):

Predecessor/Successor

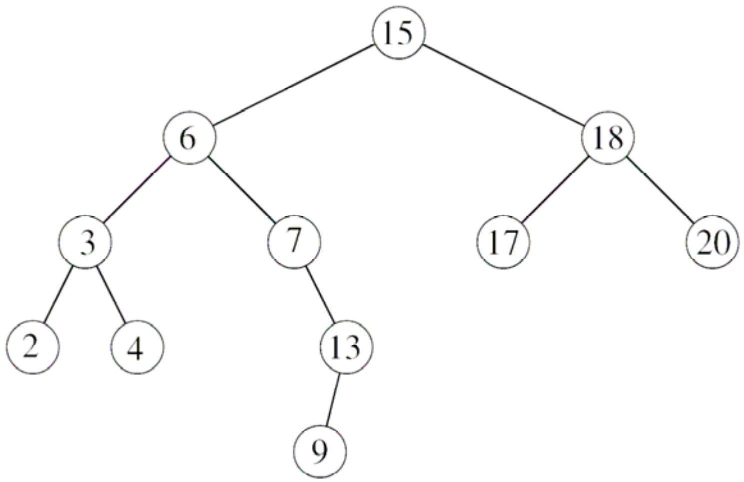
- Can modify Search (into Search') such that, if k is not stored in BST, we get x such that:
  - Either it has the largest key[x]<k, or
  - It has the smallest key[x]>k
- Useful when k prone to errors
- What if we always want a successor of k ?
  - x=Search'(k)
  - If key[x]<k, then return Successor(x)
  - Else return x



Binary Search Tree (BST)

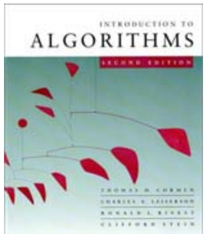
TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR.

Example:



- Find the successor of the node with key value 15. (Answer: Key value 17)
- Find the successor of the node with key value 6. (Answer: Key value 7)
- Find the successor of the node with key value 4. (Answer: Key value 6)
- Find the predecessor of the node with key value 6. (Answer: Key value 4)

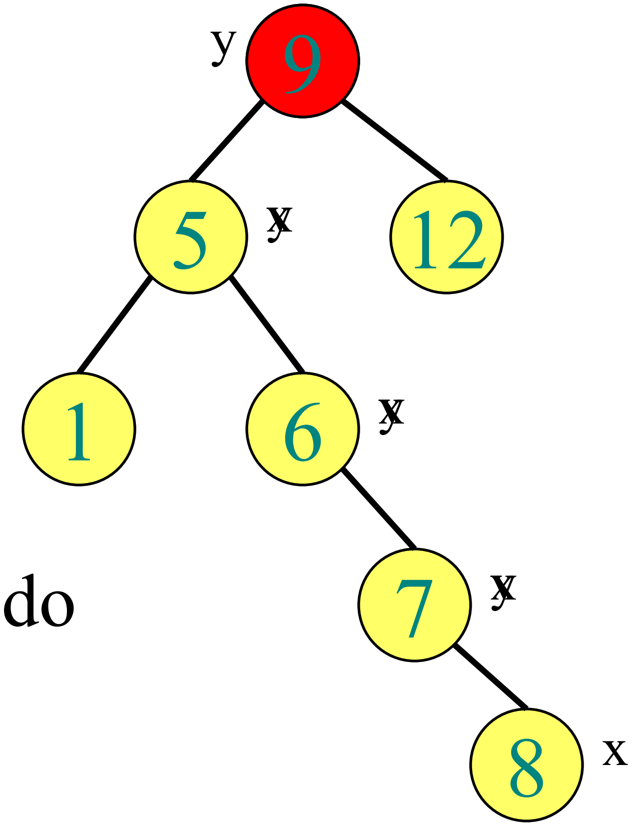
Time: For both the TREE-SUCCESSOR and TREE-PREDECESSOR procedures, in both cases, we visit nodes on a path down the tree or up the tree. Thus, running time is ***$O(h)$,*** where ***h*** is the height of the tree.

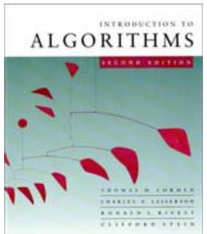


Successor

Successor(x):

- If right[x] ≠ NIL then return Minimum(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

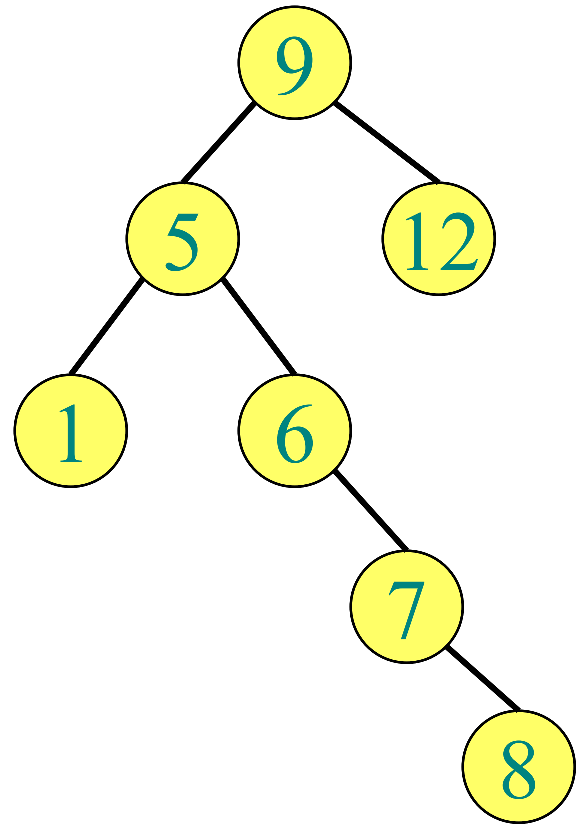


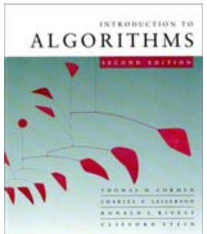


Minimum

Minimum(x)

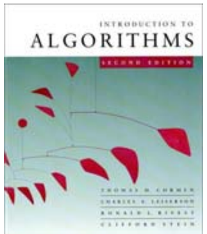
- While left[x]≠NIL do
  - x ← left[x]
- Return x





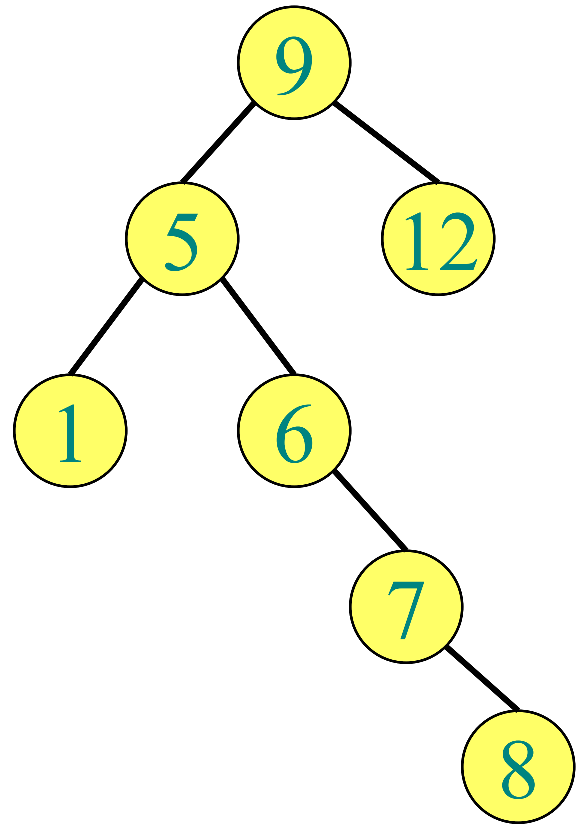
Nearest Neighbor

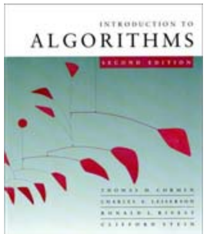
- Assuming keys are numbers
- For a key k, can we find x such that |k-key[x]| is minimal ?
- Yes:
  - key[x] must be either a predecessor or successor of k
  - y=Search'(k) //y is either succ or pred of k
  - y' =Successor(y)
  - y''=Predecessor(y)
  - Report the closest of key[y], key[y'], key[y'']



Analysis

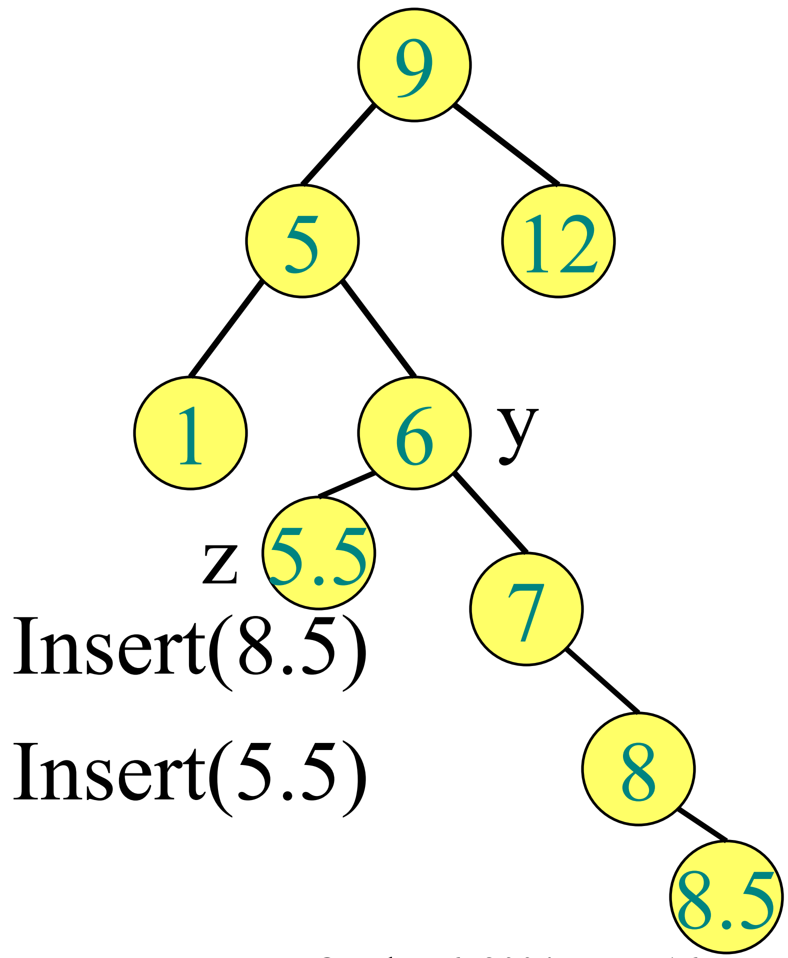
- How much time does all of this take ?
- Worst case: O(height)
- Height really important
- Tree better be balanced





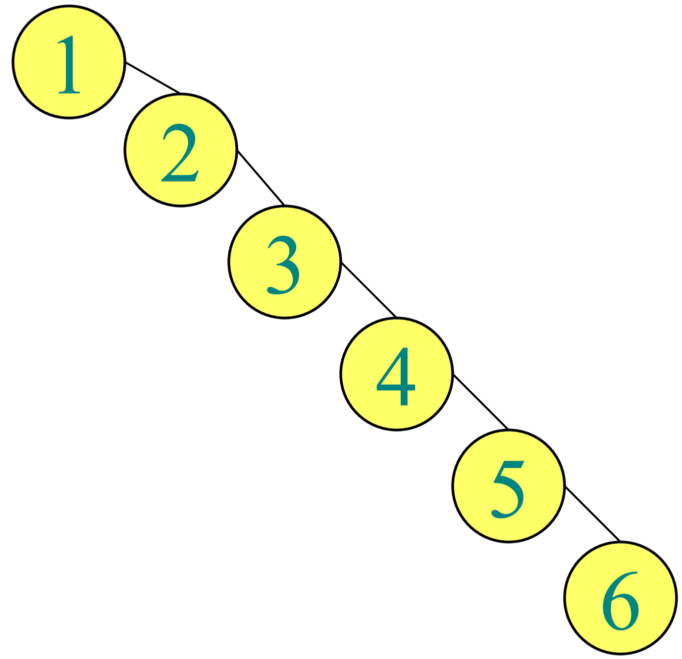
Constructing BST

Insert(z):
- y ← NIL
- x ← root
- While x ≠ NIL do
  - y ← x
  - If key[z] < key[x]
    then x ← left[x]
    else x ← right[x]
- p[z] ← y
- If key[z] < key[y]
  then left[y] ← z
  else right[y] ← z

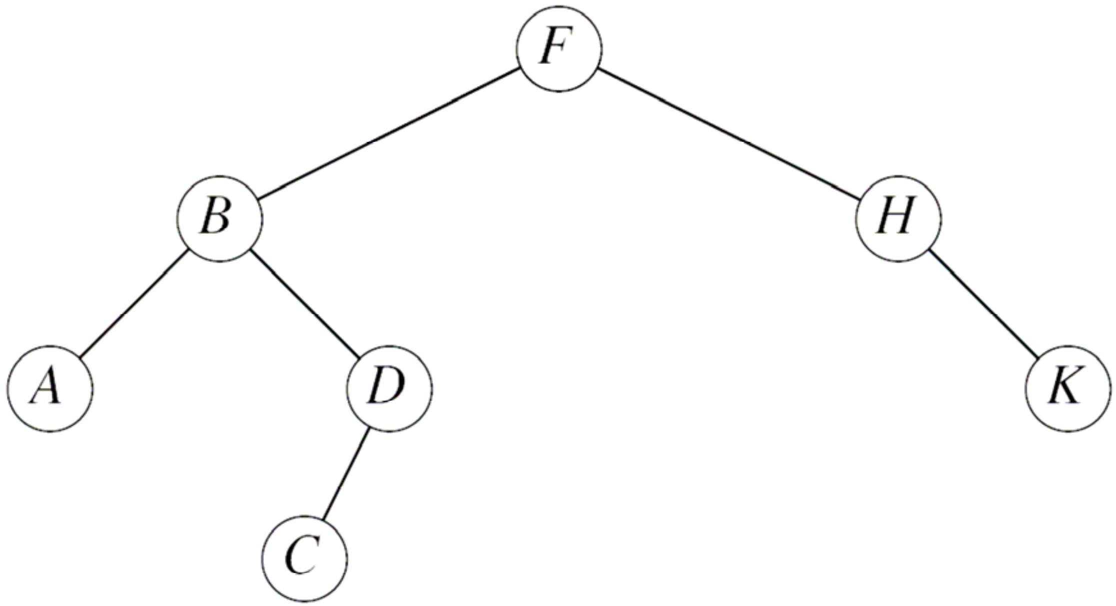


Analysis

- After we insert n elements, what is the worst possible BST height ?
- Pretty bad: n-1



BST deletion



Example: We can demonstrate on the above sample tree.

- For Case 1, delete *K*.
- For Case 2, delete *H*.
- For Case 3, delete *B*, swapping it with *C*.

Time: $O(h)$, on a tree of height h.

BST deletion

Deletion

TREE-DELETE is broken into three cases.

Case 1: *z* has no children.

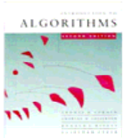
- Delete *z* by making the parent of *z* point to NIL, instead of to *z*.

Case 2: *z* has one child.

- Delete *z* by making the parent of *z* point to *z*'s child, instead of to *z*.

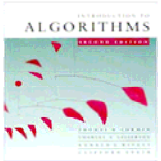
Case 3: *z* has two children.

- *z*'s successor *y* has either no children or one child. (*y* is the minimum node—with no left child—in *z*'s right subtree.)
- Delete *y* from the tree (via Case 1 or 2).
- Replace *z*'s key and satellite data with *y*'s.

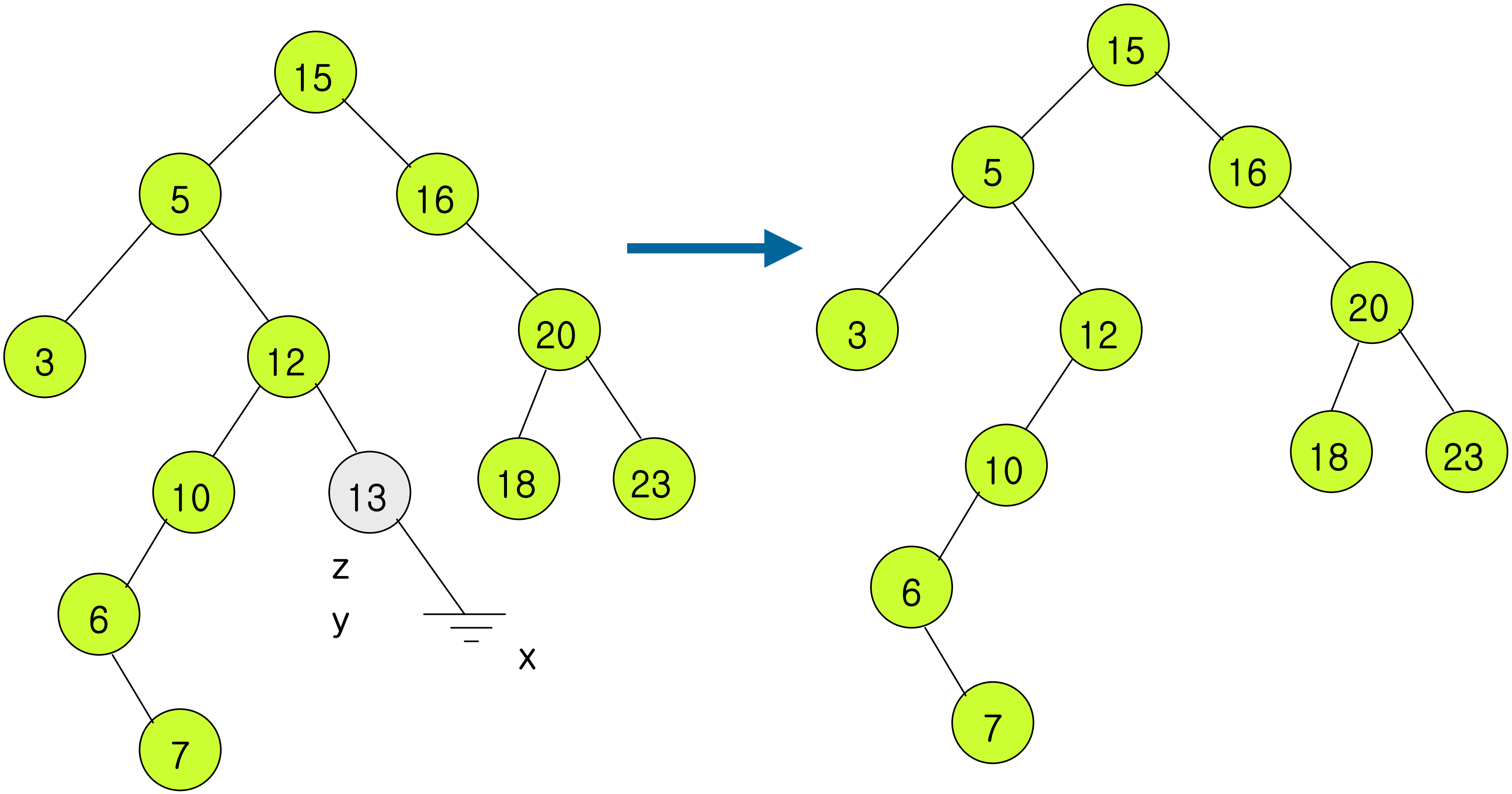


BST deletion

```
TREE-DELETE(T, z)
▷ Determine which node y to splice out: either z or z's successor.
if left[z] = NIL or right[z] = NIL
   then y ← z
   else y ← TREE-SUCCESSOR(z)
▷ x is set to a non-NIL child of y, or to NIL if y has no children.
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
▷ y is removed from the tree by manipulating pointers of p[y] and x.
if x ≠ NIL
   then p[x] ← p[y]
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
▷ If it was z's successor that was spliced out, copy its data into z.
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

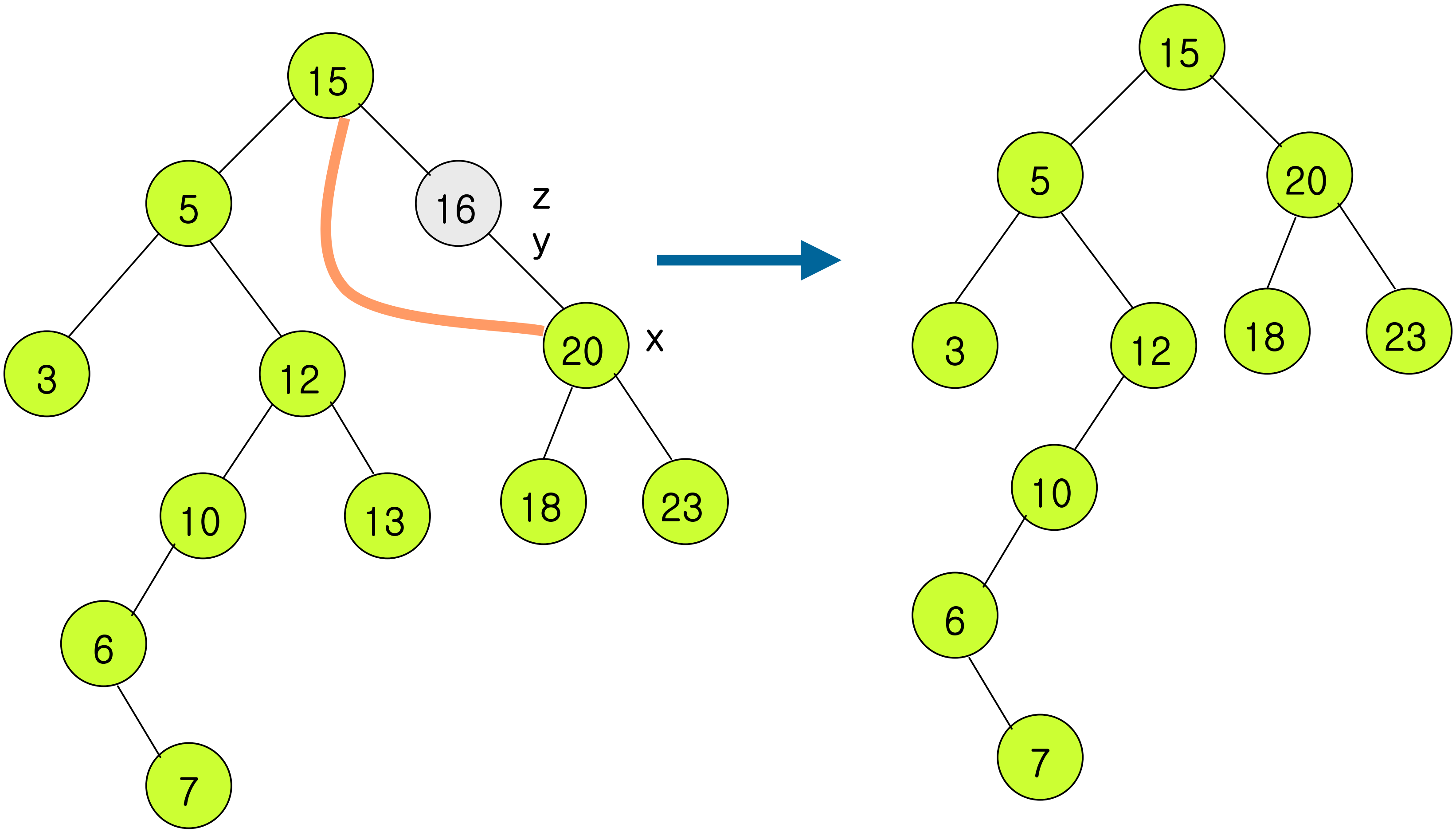


BST deletion

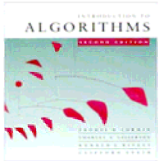


Case 1: z has no children

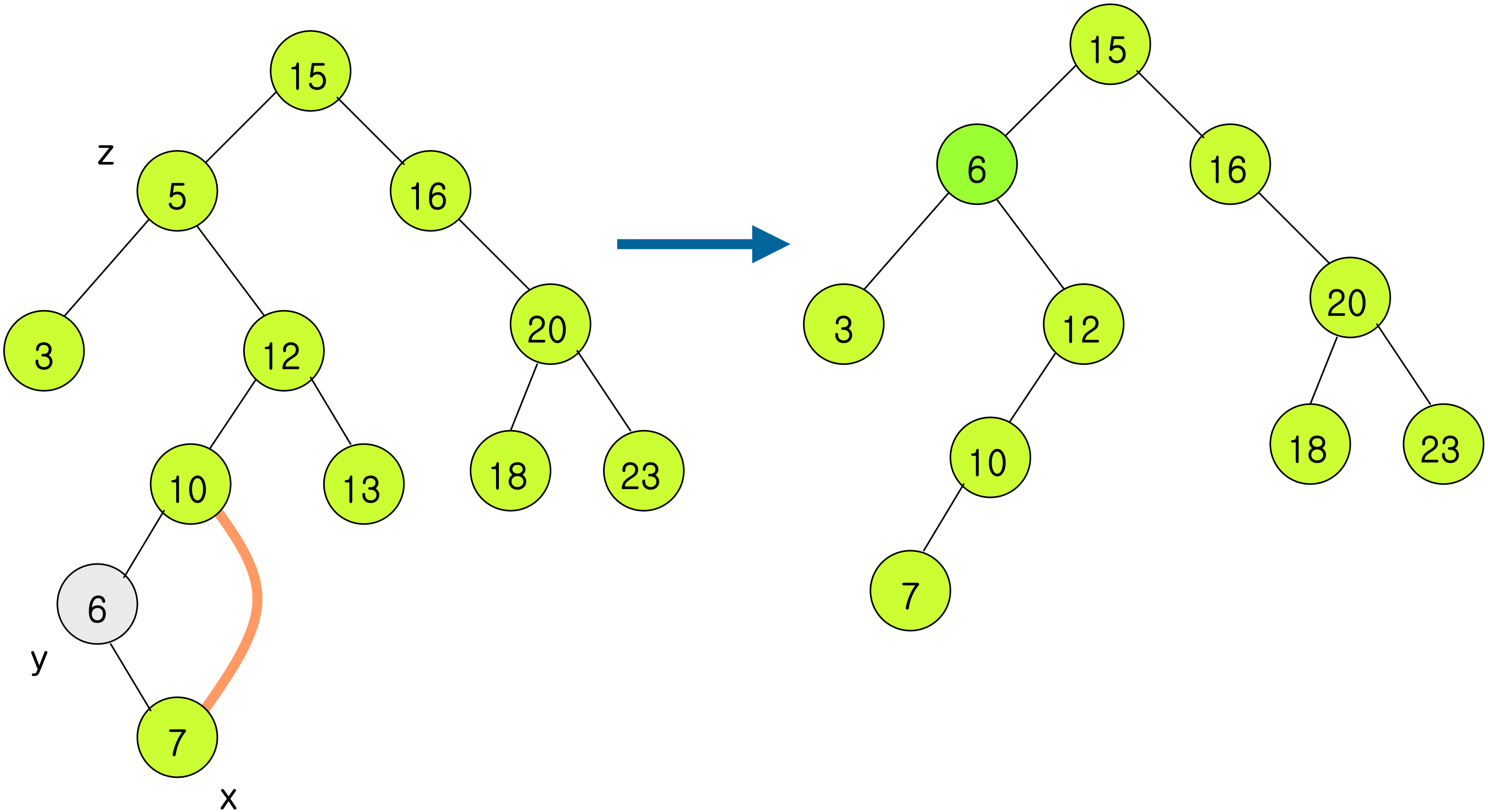
BST deletion



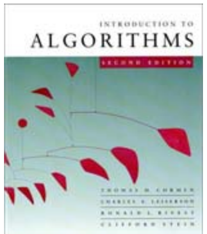
Case 2: z has one child



BST deletion



Case 3: z has two children



Average case analysis

- Consider keys 1,2,…,n, in a random order
- Each permutation equally likely
- For each key perform Insert
- What is the likely height of the tree ?
- It is O(log n)

Summing up

- We have seen BSTs
- Support Search, Successor, Nearest Neighbor etc, as well as Insert
- Worst case: O(n)
- But O(log n) on average
- Next week: O(log n) worst case