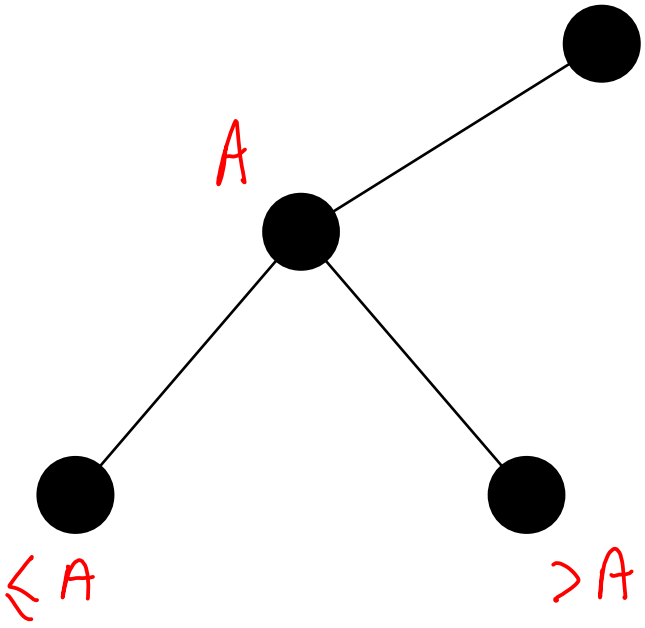
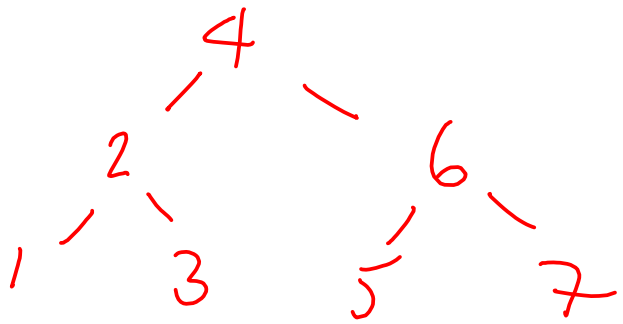
Binary Search Trees (BSTs)

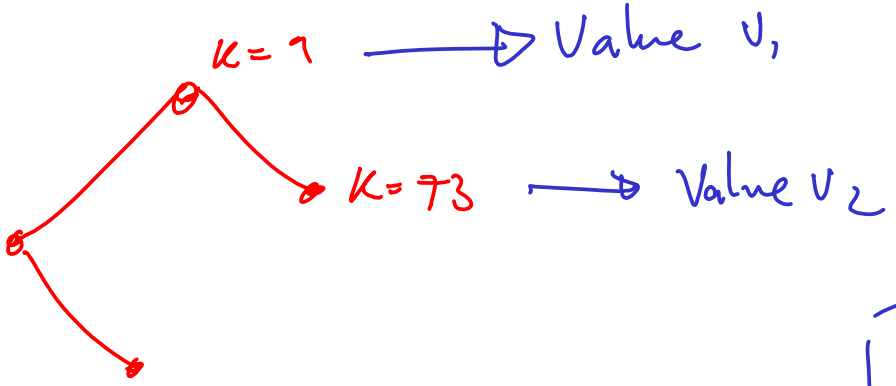
- Binary tree where each node has a value X, a left branch (<X), a right branch (>X) and a parent





Relation to Tables

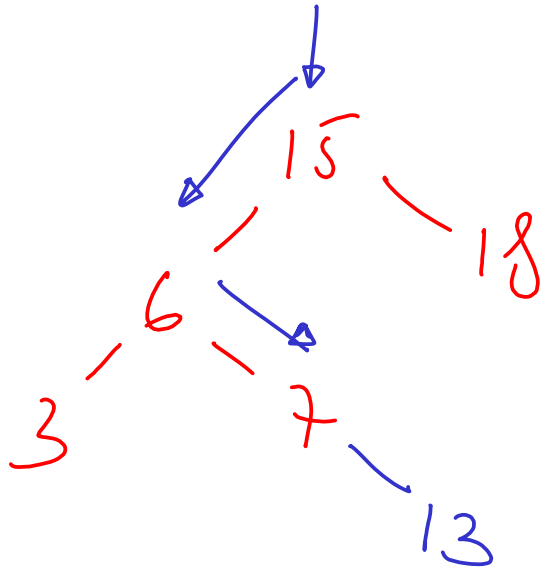
- BSTs are a good choice for implementing the Table ADT.
- The tree is constructed using the keys as node values
- They are good because they're easy to search and do operations on...



Key	Value
9	V_1
73	V_2

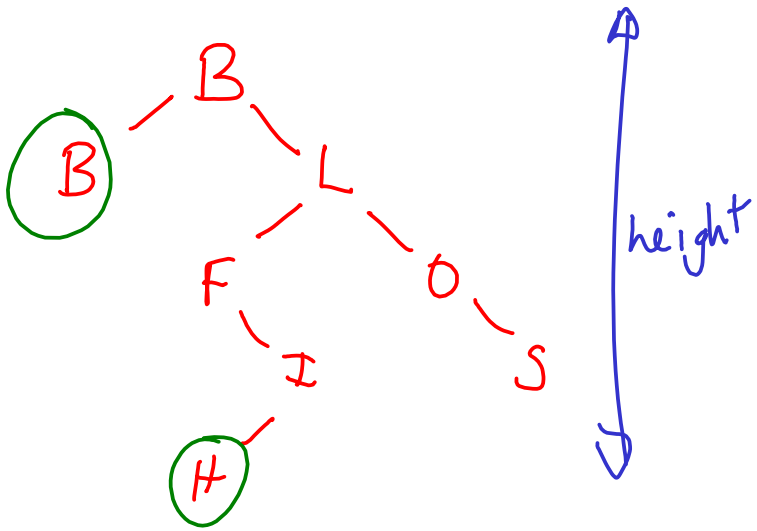
Building a BST: Insertion

1. follow tree down until you hit a null
2. Insert at null

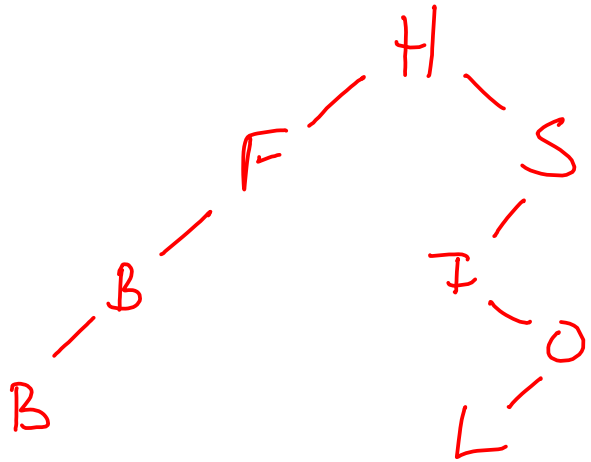


Insertion Example

BLOBFISH



HSIFBOLB

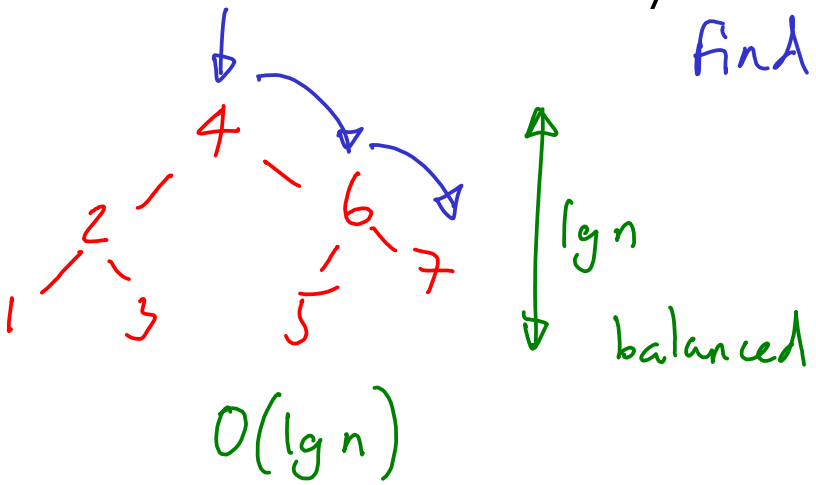


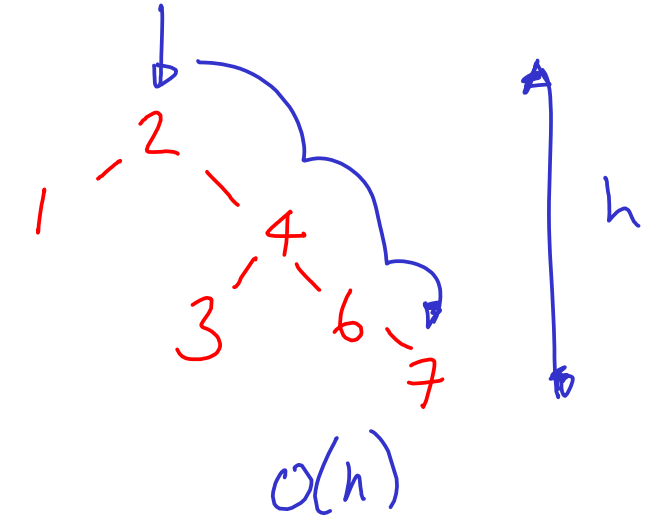
- Insertion order matters
- We only add at leaves

$O(h)$

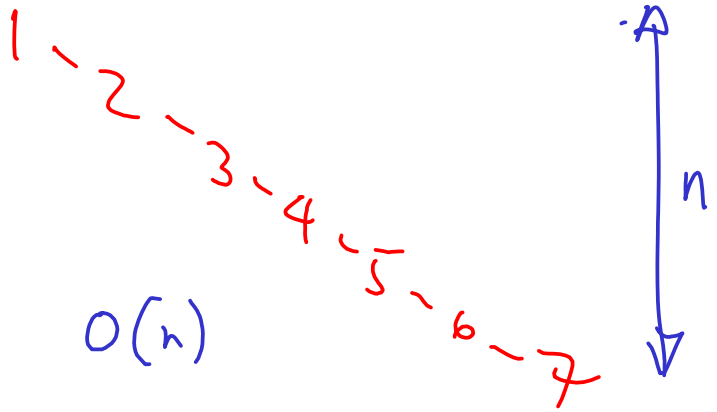
Search Costs

- We have to be careful to balance the tree:
- We will look at how to auto-balance next lecture
- How do we manually balance?

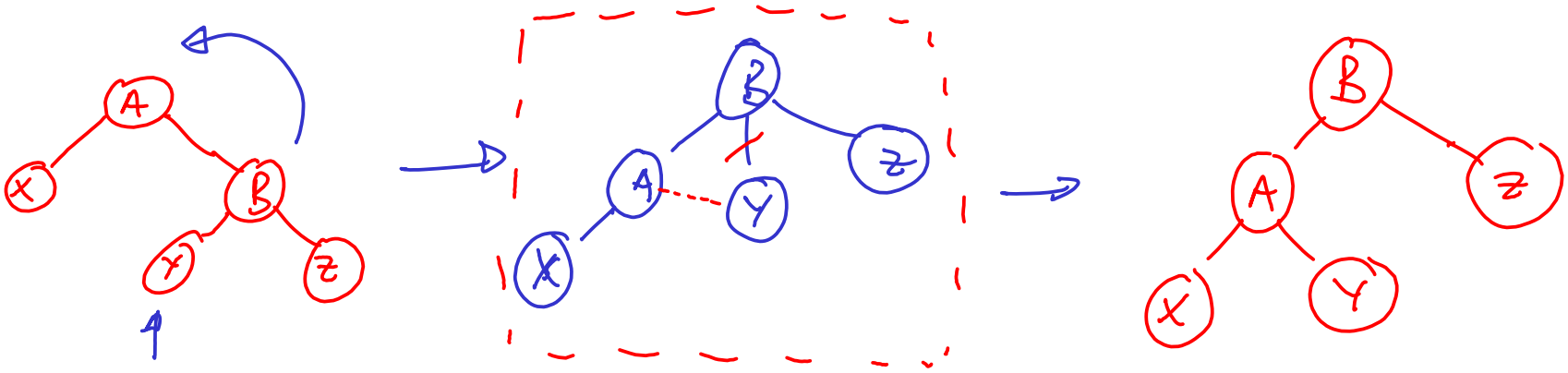


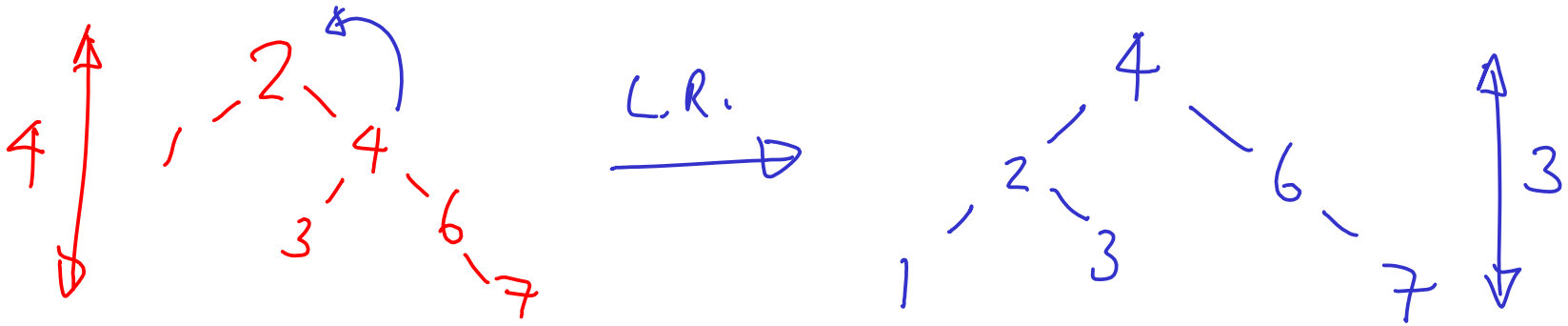


→ Need balanced trees



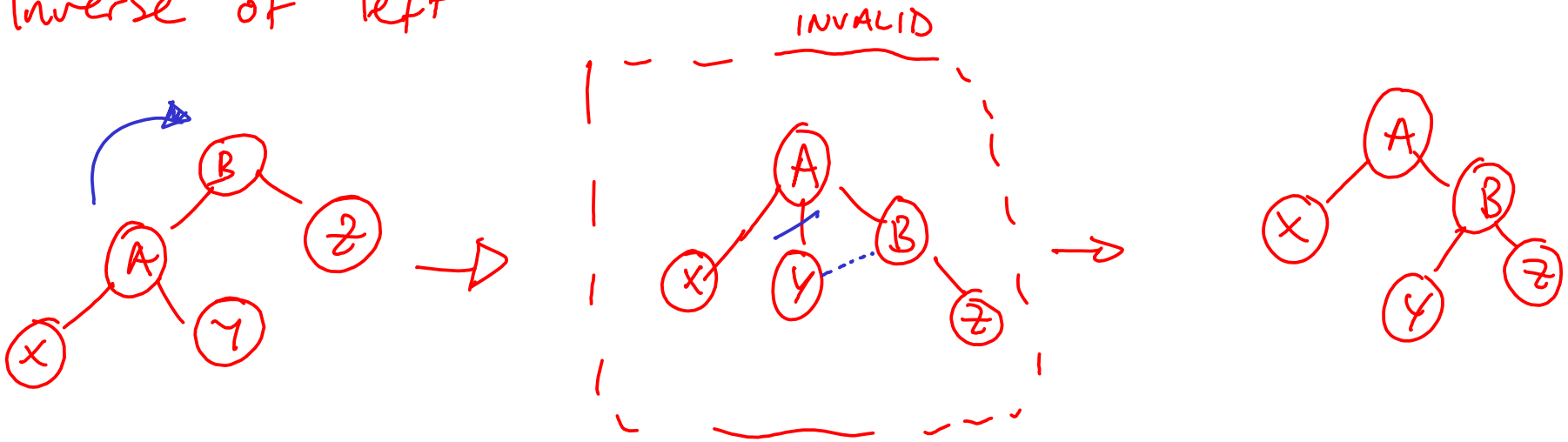
Left Rotation





Right Rotation

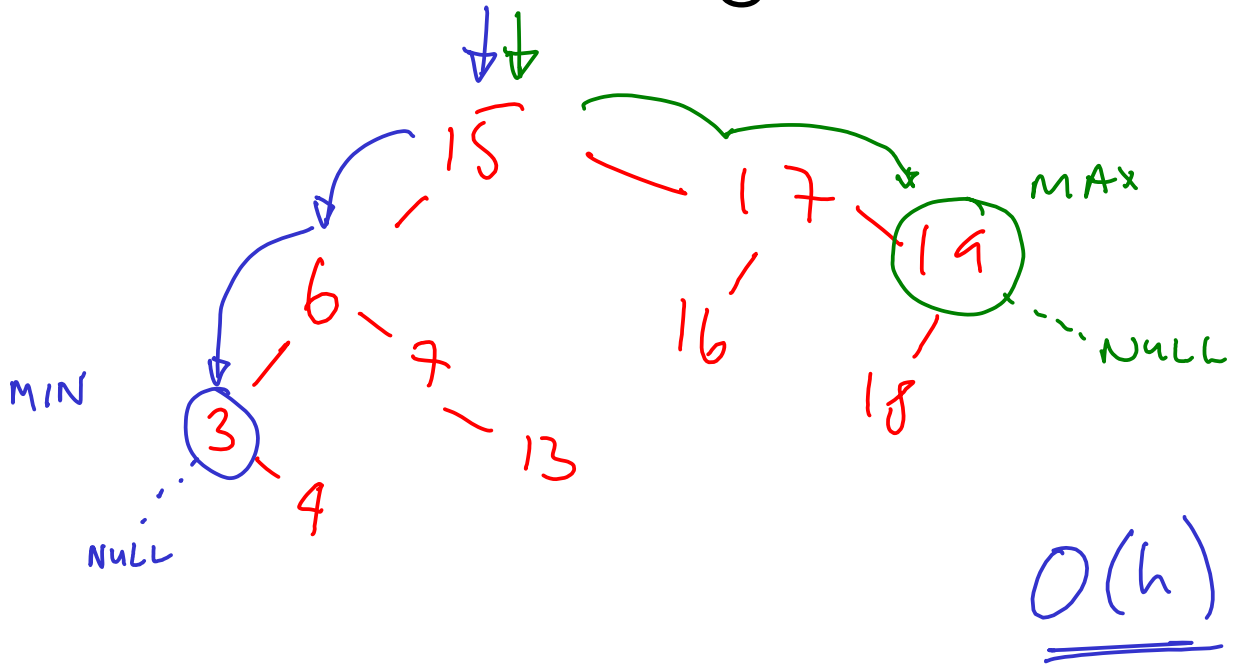
Inverse of left



Every rotation → one side lengthened by 1
" " shortened by 1

Min and Max

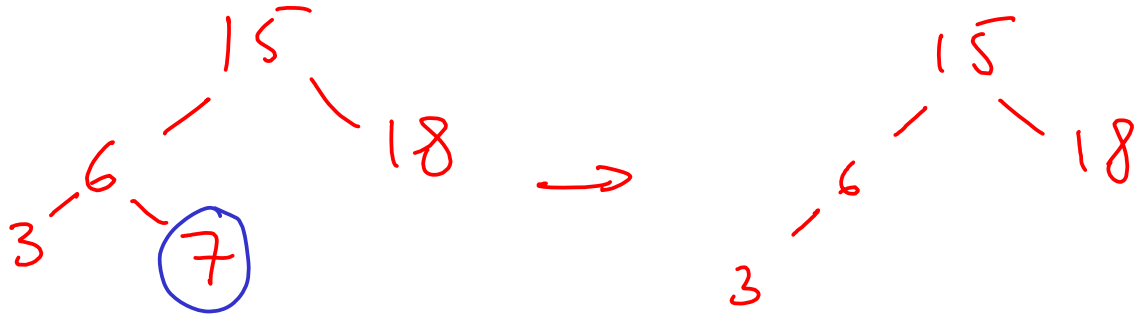
- Minimum of a BST can be found by walking down all of the left branches
- Maximum: all of the right branches



Deletion

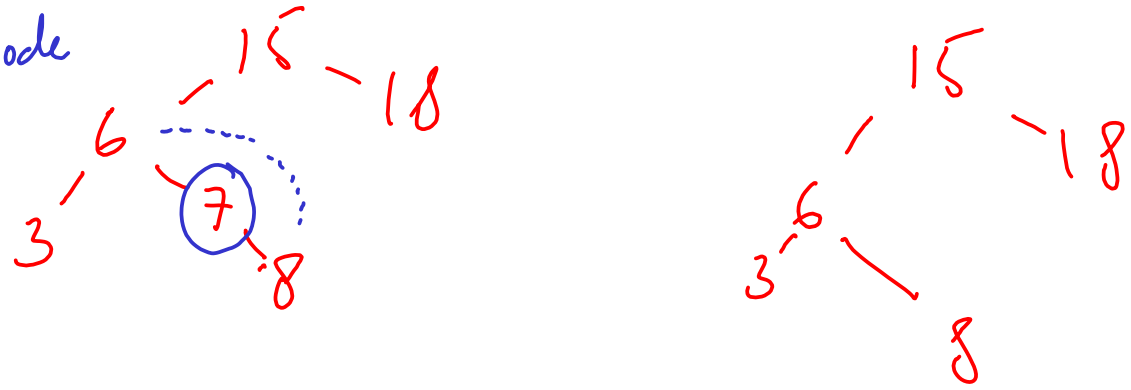
Case 1 Delete leaf

⇒ Delete node



Case 2 Deleted node has 1 child

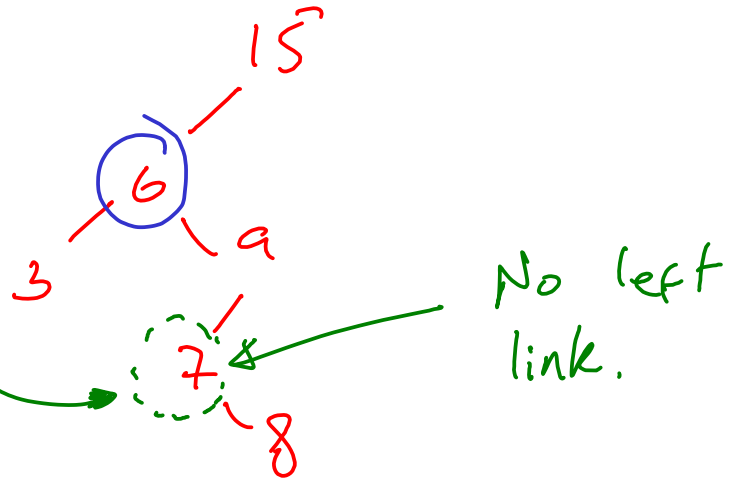
→ splice out the node

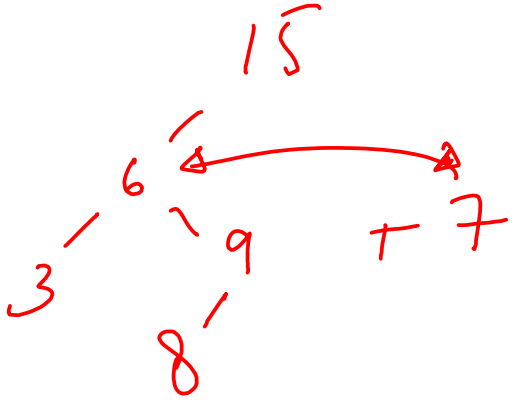


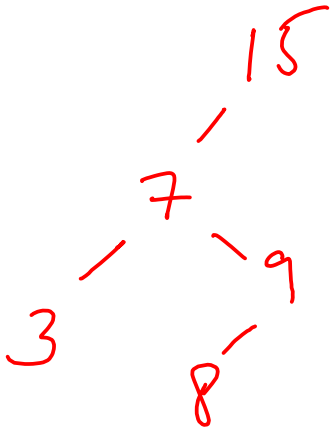
Deletion

Case 3: 2 children

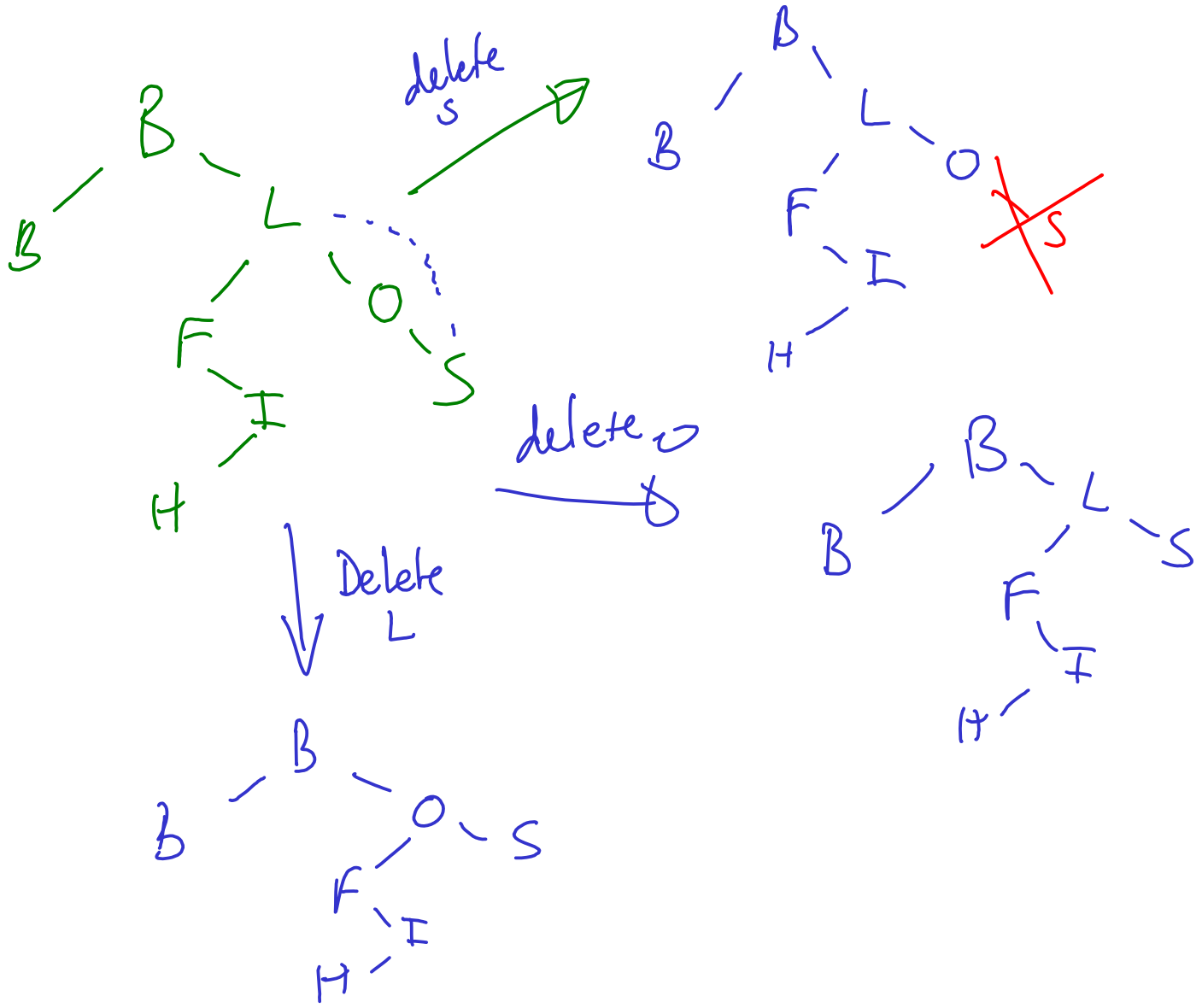
1. Find successor
2. Splice out successor
3. Replace deleted node with successor



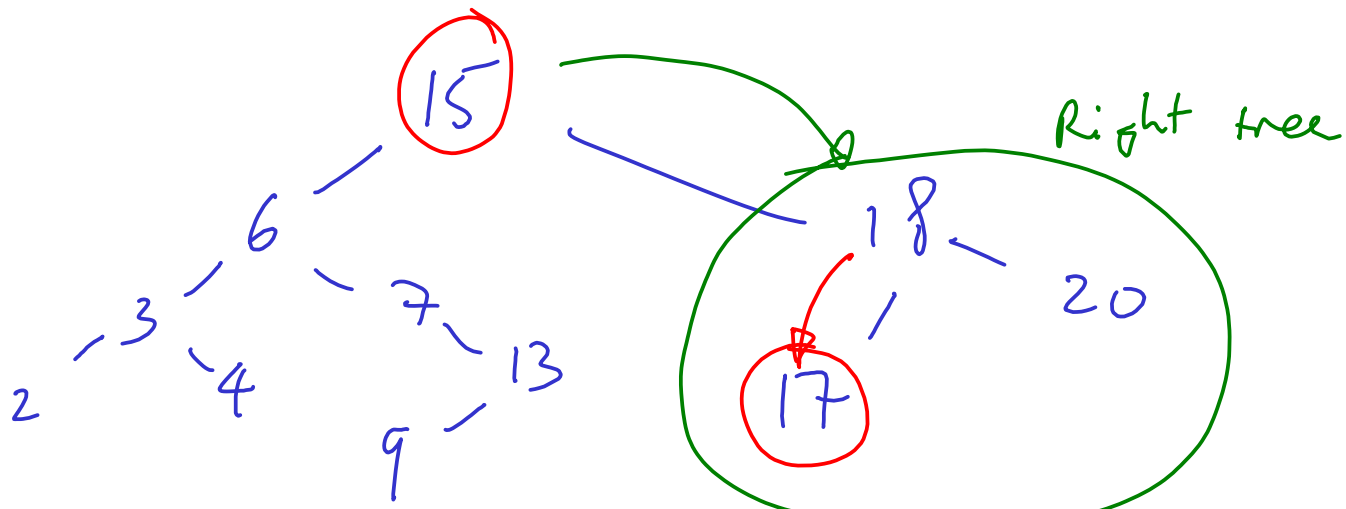




Examples



Find the Successor



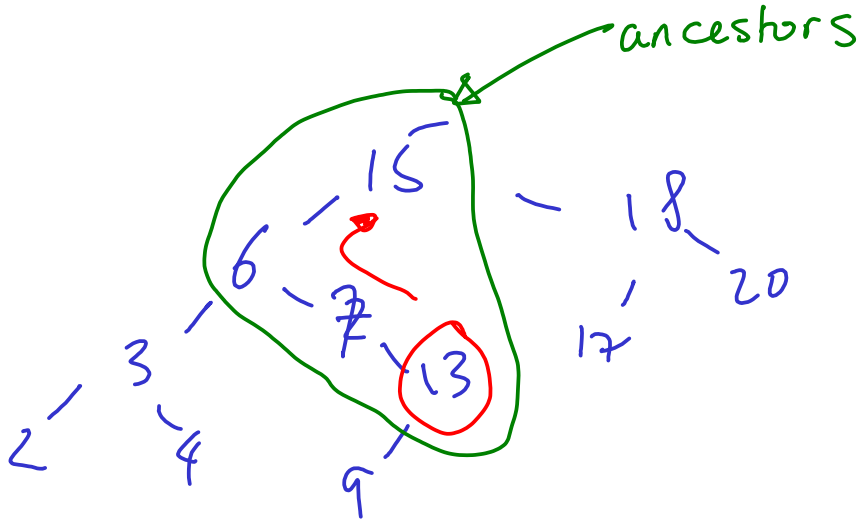
15, 17

Case 1 has a right tree

⇒ Successor in right subtree

⇒ Min. right subtree

Find the Successor



13, 15

Case 2 No right subtree

⇒ Move up the tree until a node that

i) is an ancestor of node

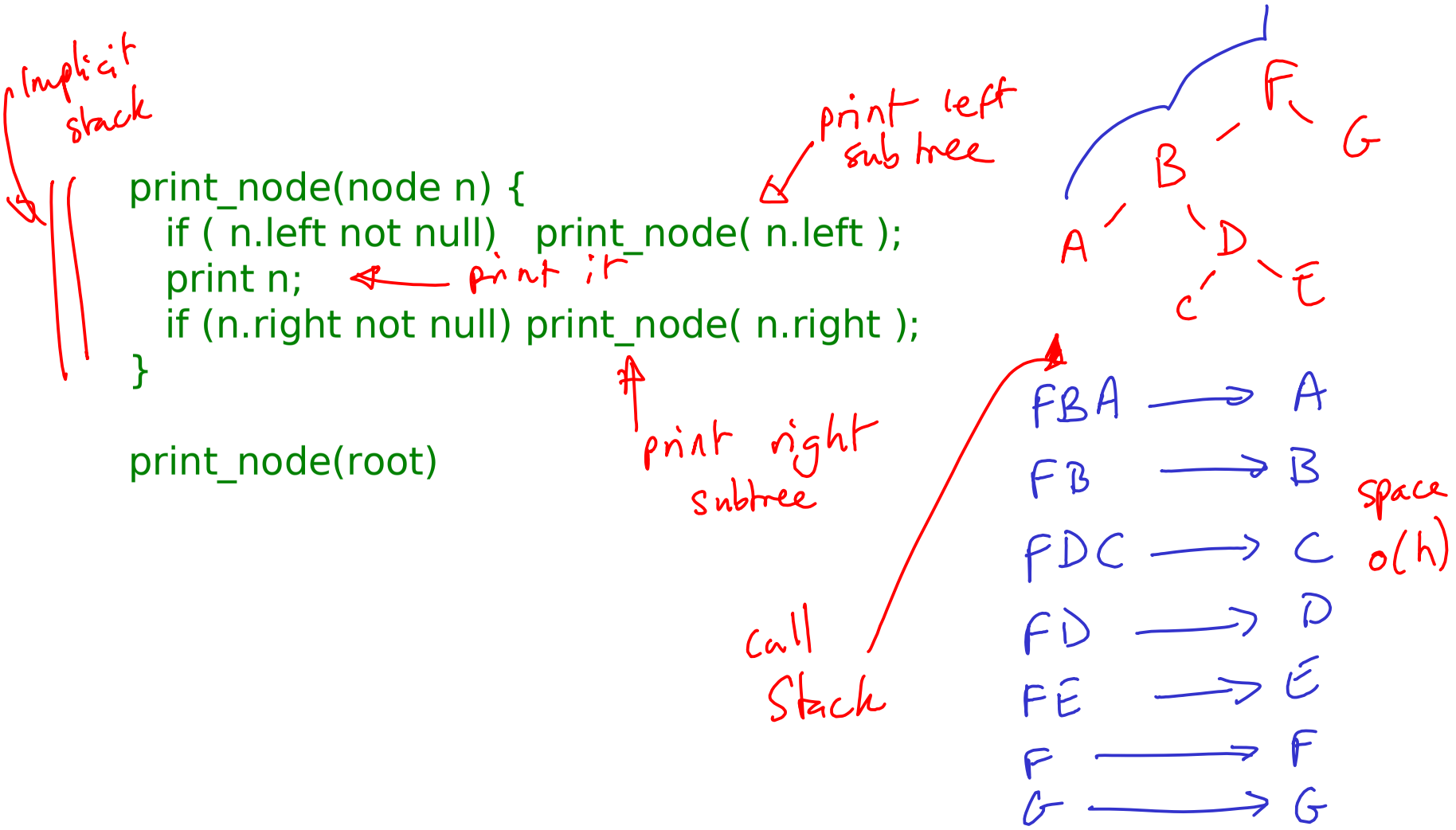
ii) has left node that is also an ancestor

Move up until you go ↗ rather than ↖

O(h)

Inorder Traversal

- How do we implement a function to print the keys in order?
- Start with the first node and look for successors recursively?



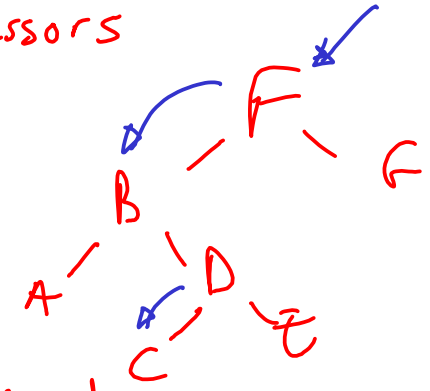
Non-recursive Inorder Traversal

explicit

- Use a stack!

Essentially: find min then find successors

1. while n not null
   stack.push(n)
   n=n.left
2. if stack not empty
   n=stack.pop()
   print n
   n=n.right
3. if ((n not null) OR
   (stack not empty))
   GOTO 1

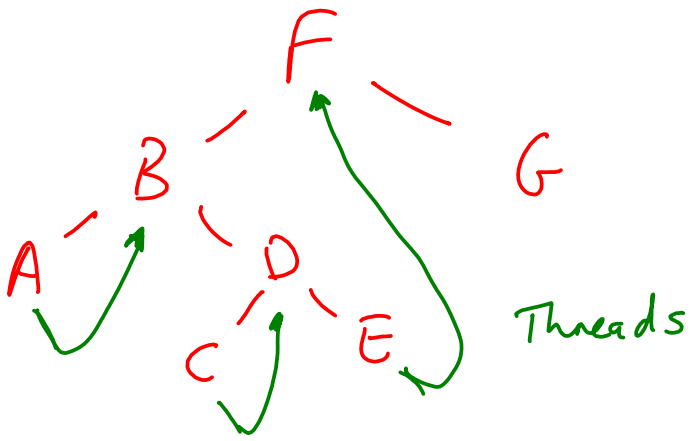


Stack	n	Printed
—	F	—
FBA	—	—
FB	—	A
F	D	B
FDC	—	—

Threaded Inorder Traversal

- No stack!

Threads point to successors



1. while n.left not null
   n=n.left
2. print n
3. if (n.right is a thread)
   n=n.right

   else

   n=n.right
   while (n not null) AND (n.left not null)
   n=n.left
4. if (n not null) GOTO 2

n	output
F	—
B	—
A	A
B	B
D	—
C	C
D	D
E	E

BST So Far

- Good complexity for operations
- Easy to represent
- But falls down if it is unbalanced

- Let's see how we can auto-balance...