School of Computer Science

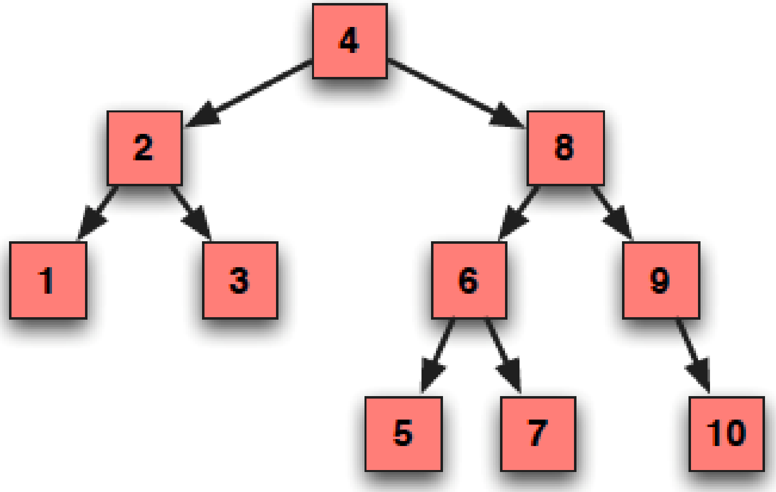
COMP SCI 1103/2103 Algorithm Design & Data Structure

Binary Trees

Review

- A graph is a collection of points where some them are connected by line segments.

- Trees are a subset of graphs with certain properties:
  - all nodes connected
  - no cycles
- Binary trees are trees with 0, 1 or 2 children



TreeItem

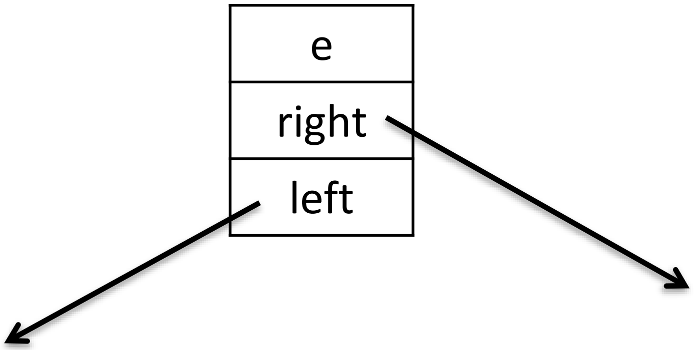
Class Handle = **Pointer to** TreeItem

Class TreeItem **of** Element

e: Element

right: Handle

left: Handle



Tree traversal

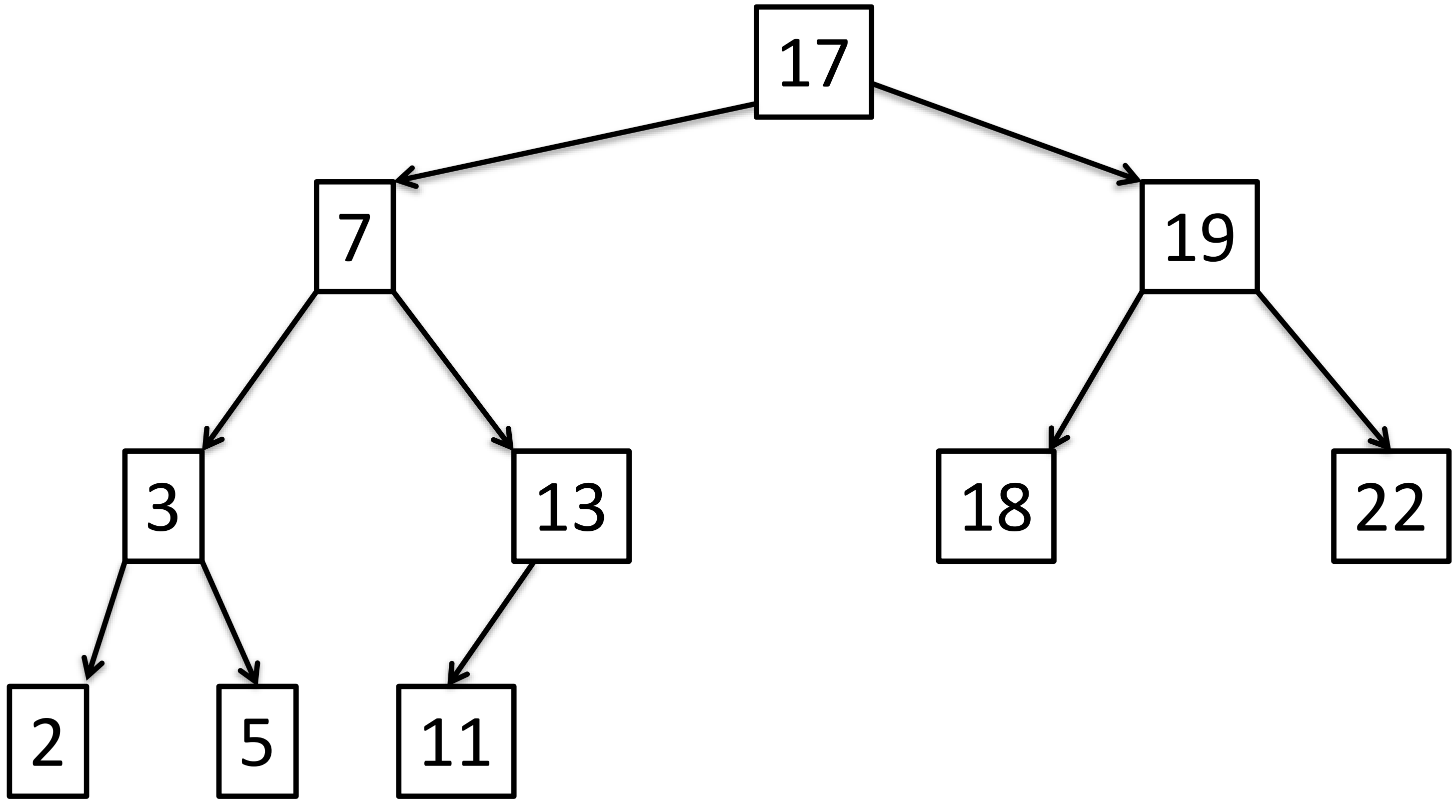
- Visit every node in the tree
  - Level-order
  - Pre-order (Node, Left, Right)
  - Post-order (Left, Right, Node)
  - In-order (Left, Node, Right)

Preorder traversal

Preorder(Tree T)

1. Visit the root (and print out the element)
2. If (T->left !=null) Preorder(T->left)
3. If (T->right !=null) Preorder(T->right)

Preorder traversal



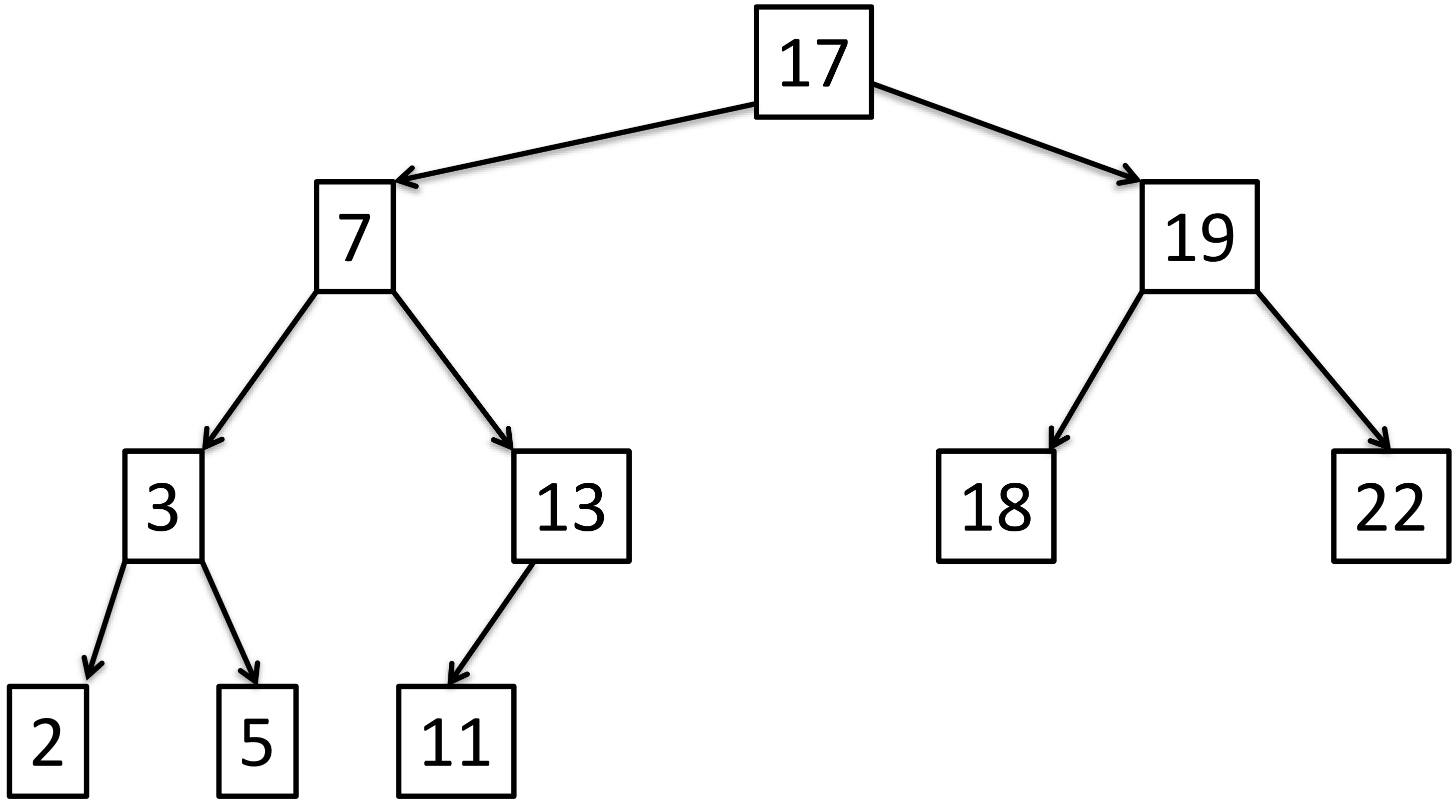
Order nodes are visited: 17, 7, 3, 2, 5, 13, 11, 19, 18, 22

Postorder traversal

Postorder(Tree T)

1. If (T->left !=null) Postorder(T->left)
2. If (T->right !=null) Postorder(T->right)
3. Visit the root (and print out the element)

Postorder traversal



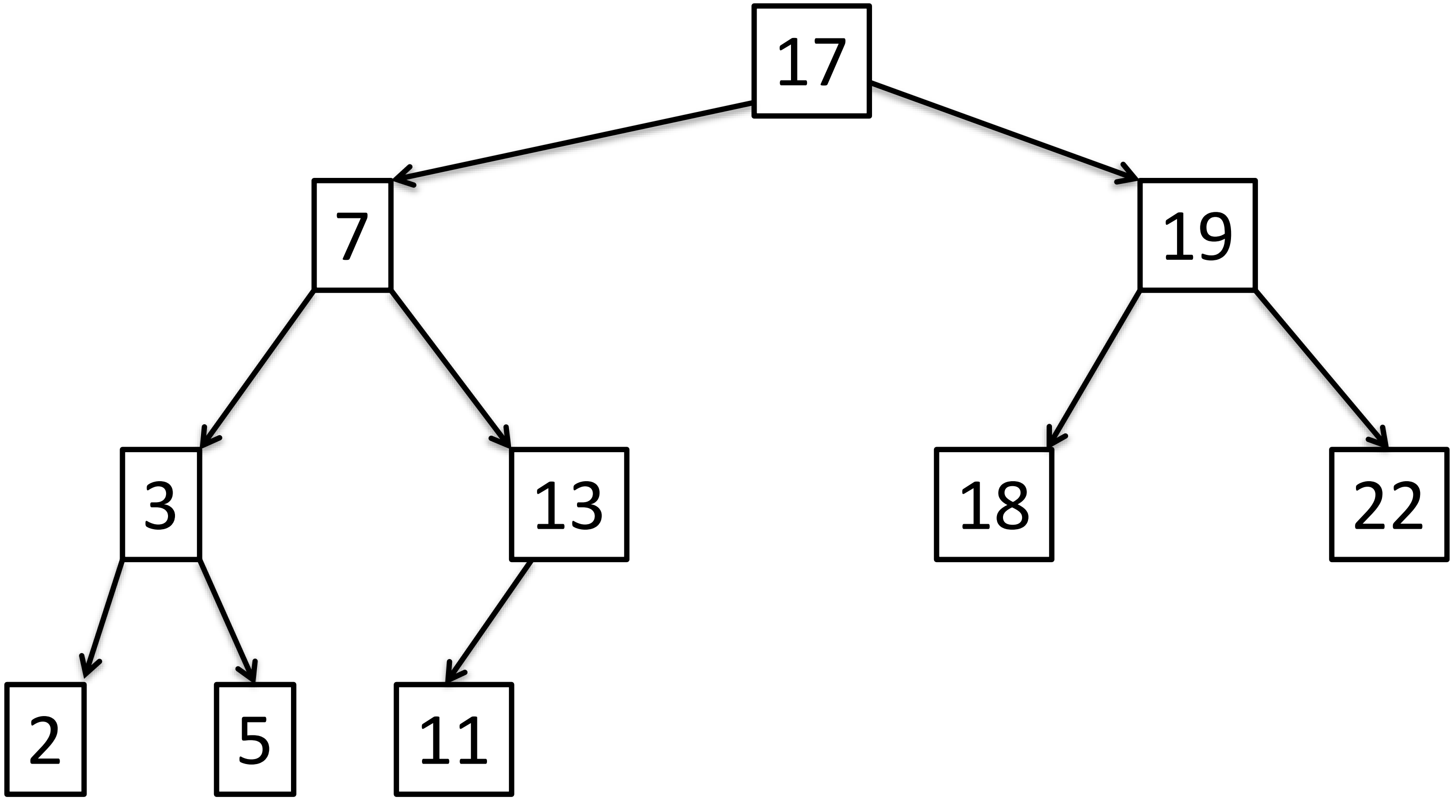
Order nodes are visited: 2, 5, 3, 11, 13, 7, 18, 22, 19, 17

Inorder traversal

Inorder(Tree T)

1. If (T->left !=null) Inorder(T->left)
2. Visit the root (and print out the element)
3. If (T->right !=null) Inorder(T->right)

Inorder traversal



Order nodes are visited: 2, 3, 5, 7, 11, 13, 17, 18, 19, 22

Observation: This sequence is sorted

Example of binary tree

- Expression Trees
  - The leaves of an expression tree are operands and other nodes contain operators.
  - The expression trees can be binary tree since most operators are unary or binary.
- We can evaluate an expression tree T by applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees.
- In-order, pre-order and post-order traverse on this tree gives us in-fix, pre-fix, and post-fix representation of arithmetic expressions

Binary search tree

- A binary search tree (BST) is a binary tree with the following properties:
  - Node values are distinct and comparable
  - The left subtree of a node contains only values that are *less than* the node's own value.
  - The right subtree of a node contains only values that are *greater than* the node's own value.

Searching

- Search whether a value exists in a dataset.
- One suitable data structure is sorted array.
  - Search takes logarithmic instead of linear time of linked list.
  - However, insertion and deletion are expensive. (Shifting array elements often takes linear time.)
- Ordered tree or binary search tree is an easy-to-implement data structure, under which searching, insertion, and deletion **all take logarithmic time on average**.
  - All are done in O(height), but height can be $\Omega(n)$ in worst case

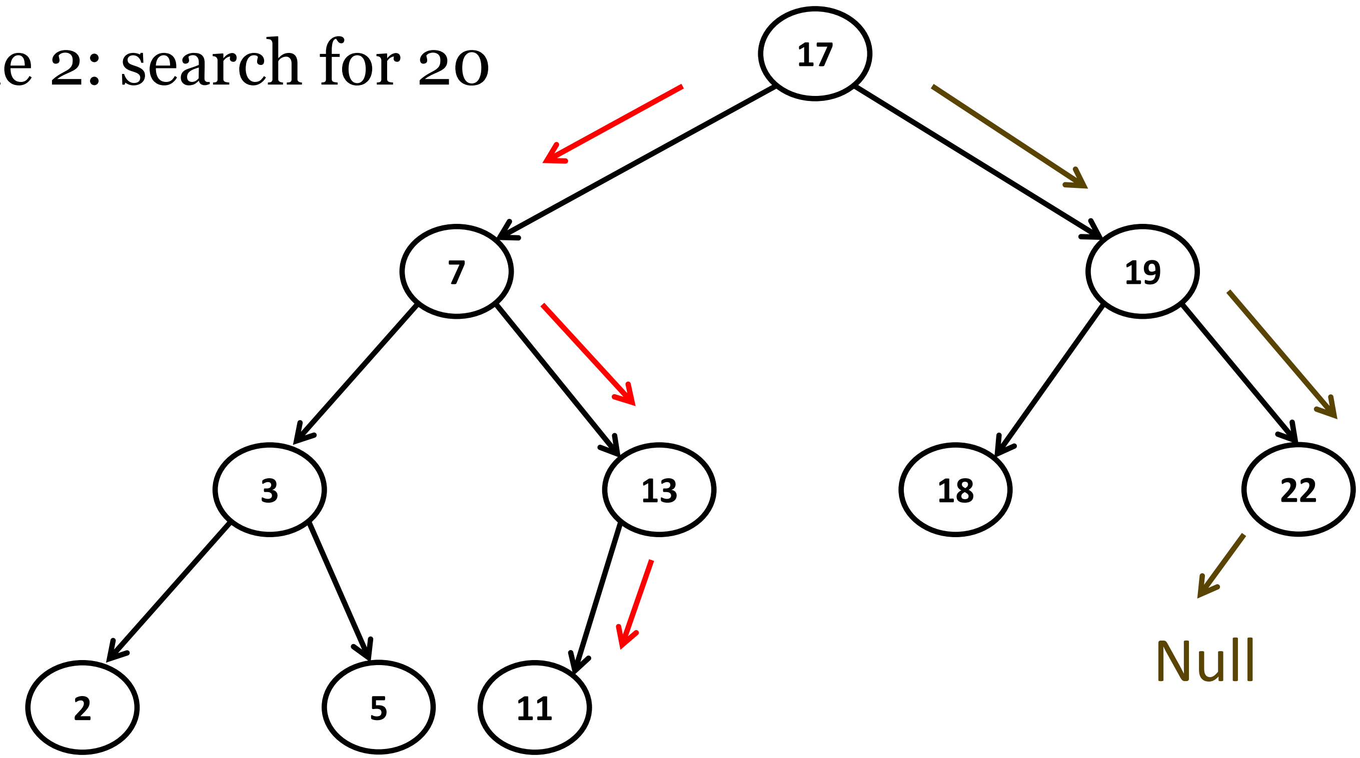
BST – searching

- This operation returns true if there is a node in tree T that has value X, or false if there is no such node.

- Start from root
- If current subtree is empty, return not found
- If target value = current value, return found
- If target value < current value, go left
- If target value > current value, go right

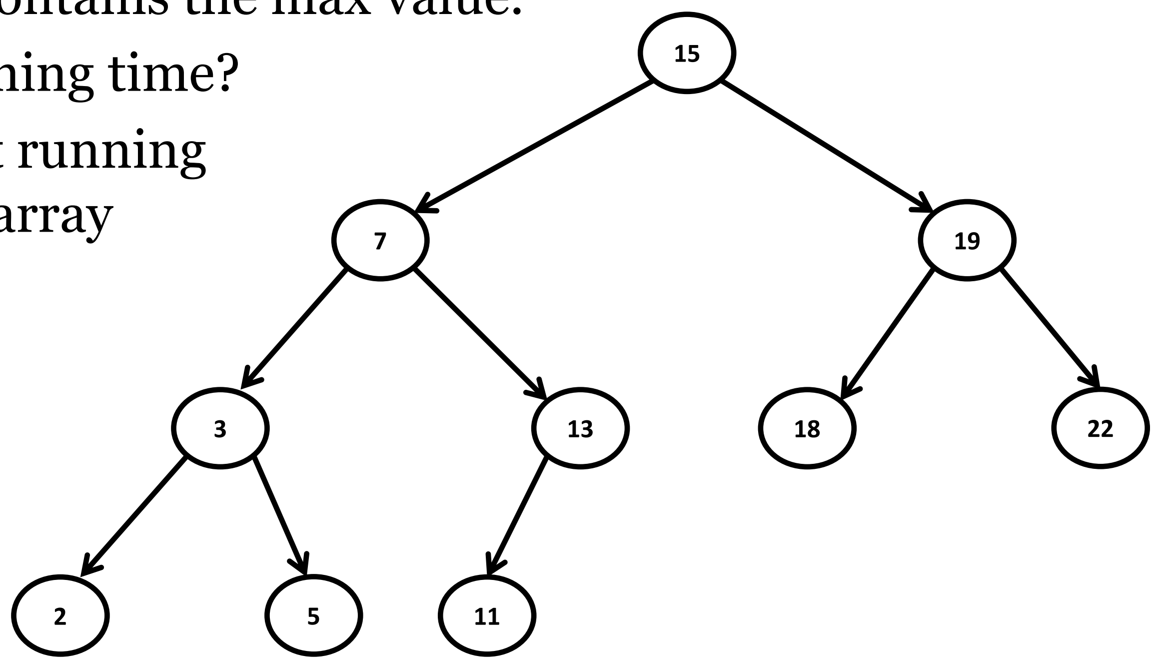
BST – searching example

- This operation returns true if there is a node in tree T that has value X, or false if there is no such node.
- Example 1: search for 11
- Example 2: search for 20



BST – min and max

- The operation returns the node containing the smallest or largest elements in the tree.
- To find the max value:
  - Start from root and go right all the way. The last node contains the max value.
  - Worst-case running time?
  - Versus constant running time for sorted array
- Similar for min.



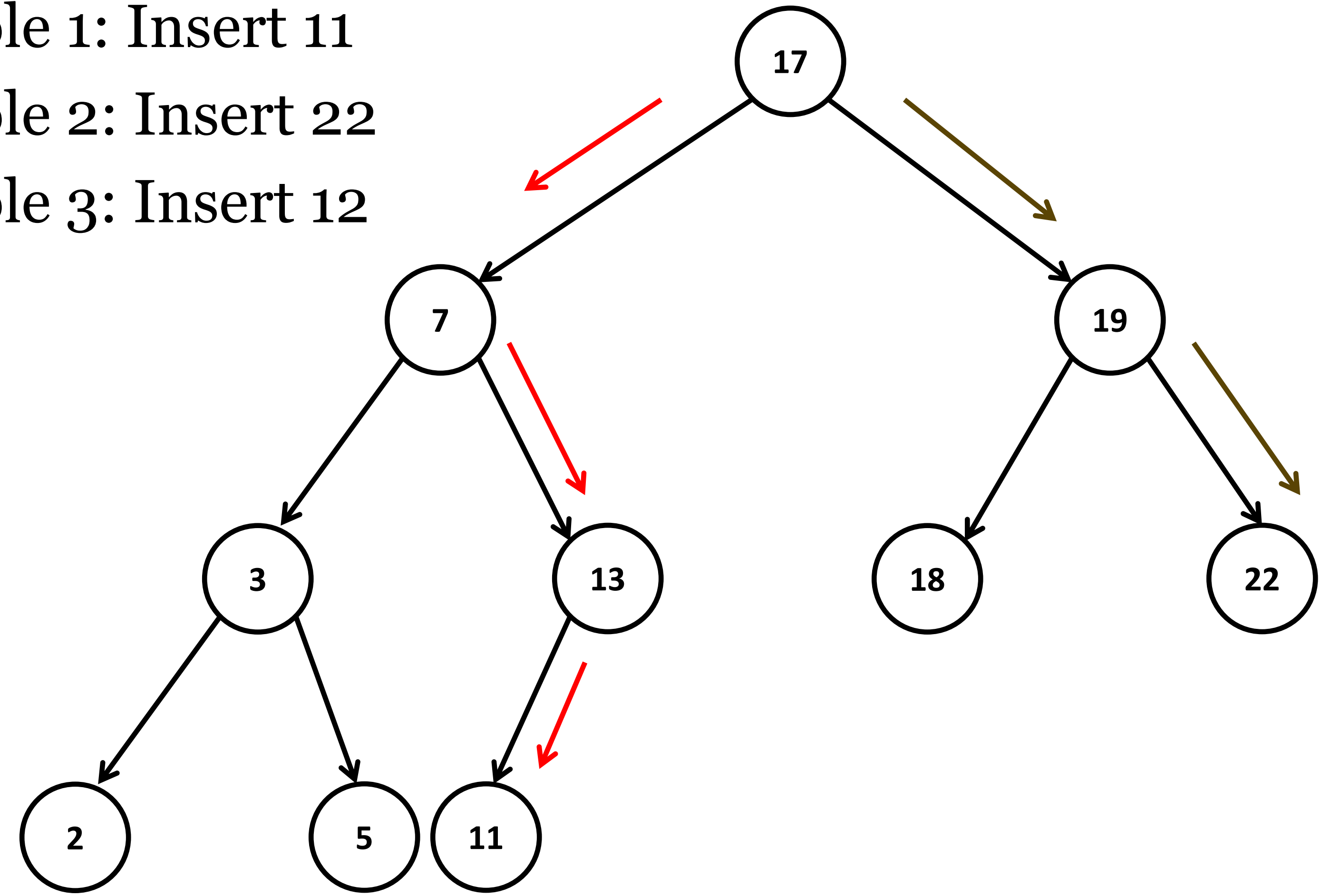
BST – insertion

- Start from root
- If current subtree is empty, create new node here.
- If target value = current value, terminate.
- If target value < current value, go left.
- If target value > current value, go right.

- What is the worst-case running time of insertion under a BST with n nodes?

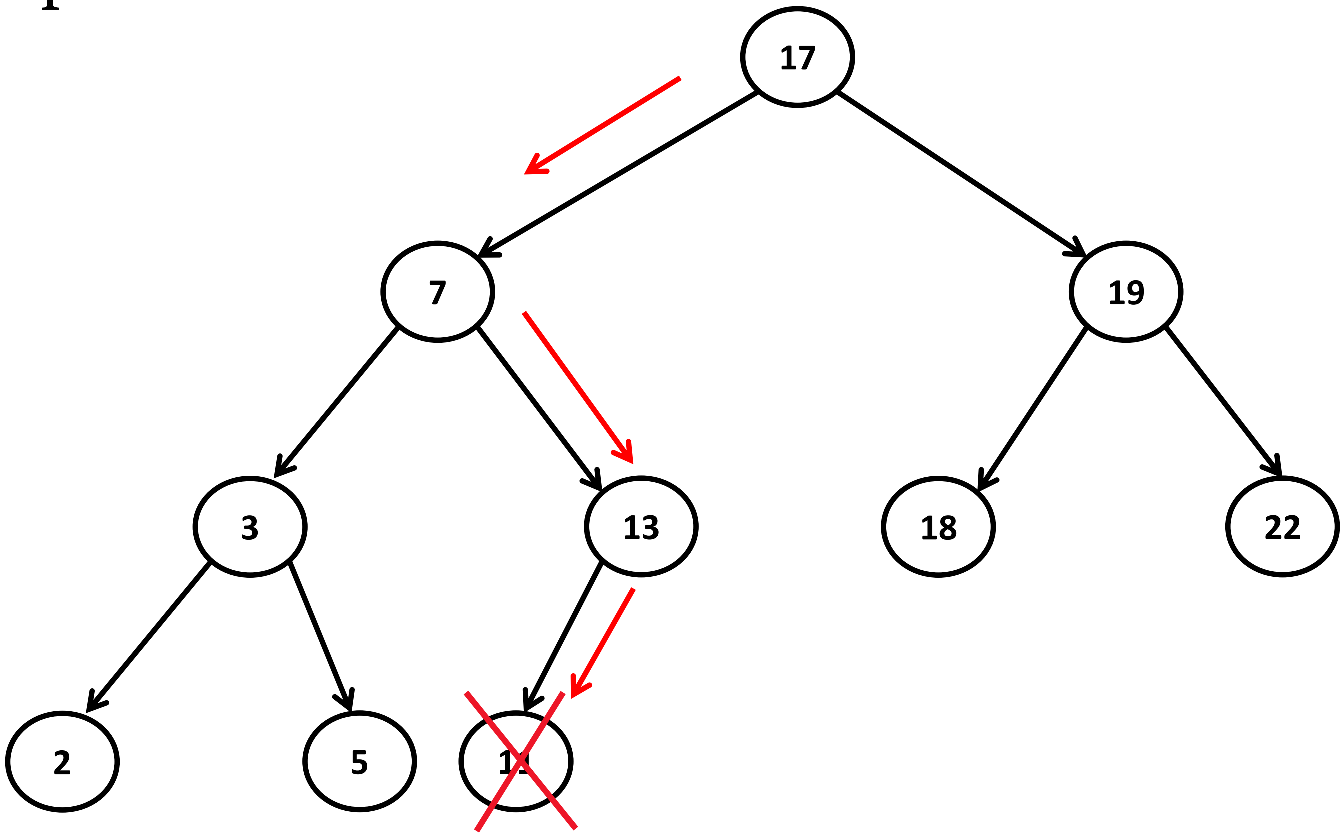
BST – insertion example

- Example 1: Insert 11
- Example 2: Insert 22
- Example 3: Insert 12



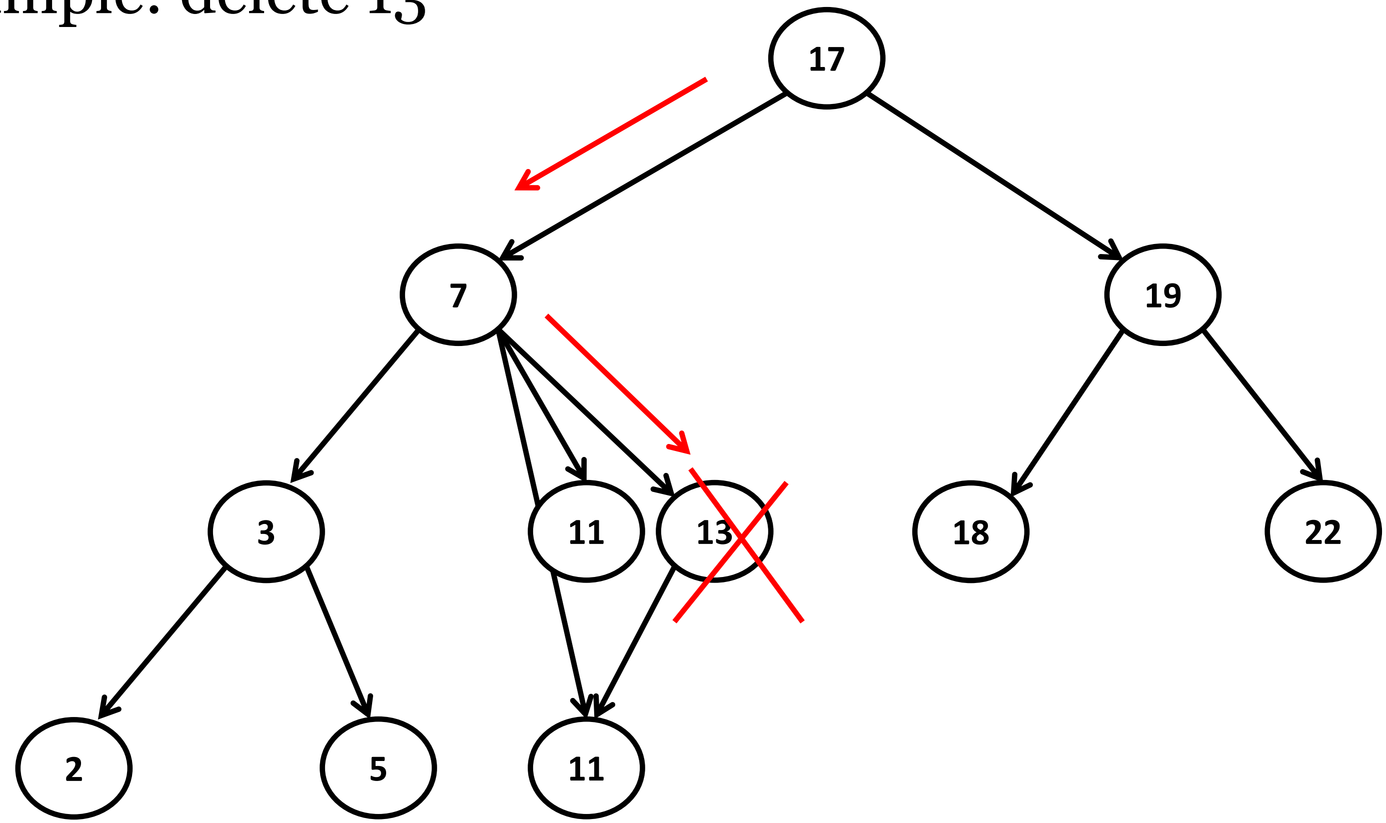
BST – deletion

- Case 1: the node to be deleted does not have any children.
- Example: delete 11



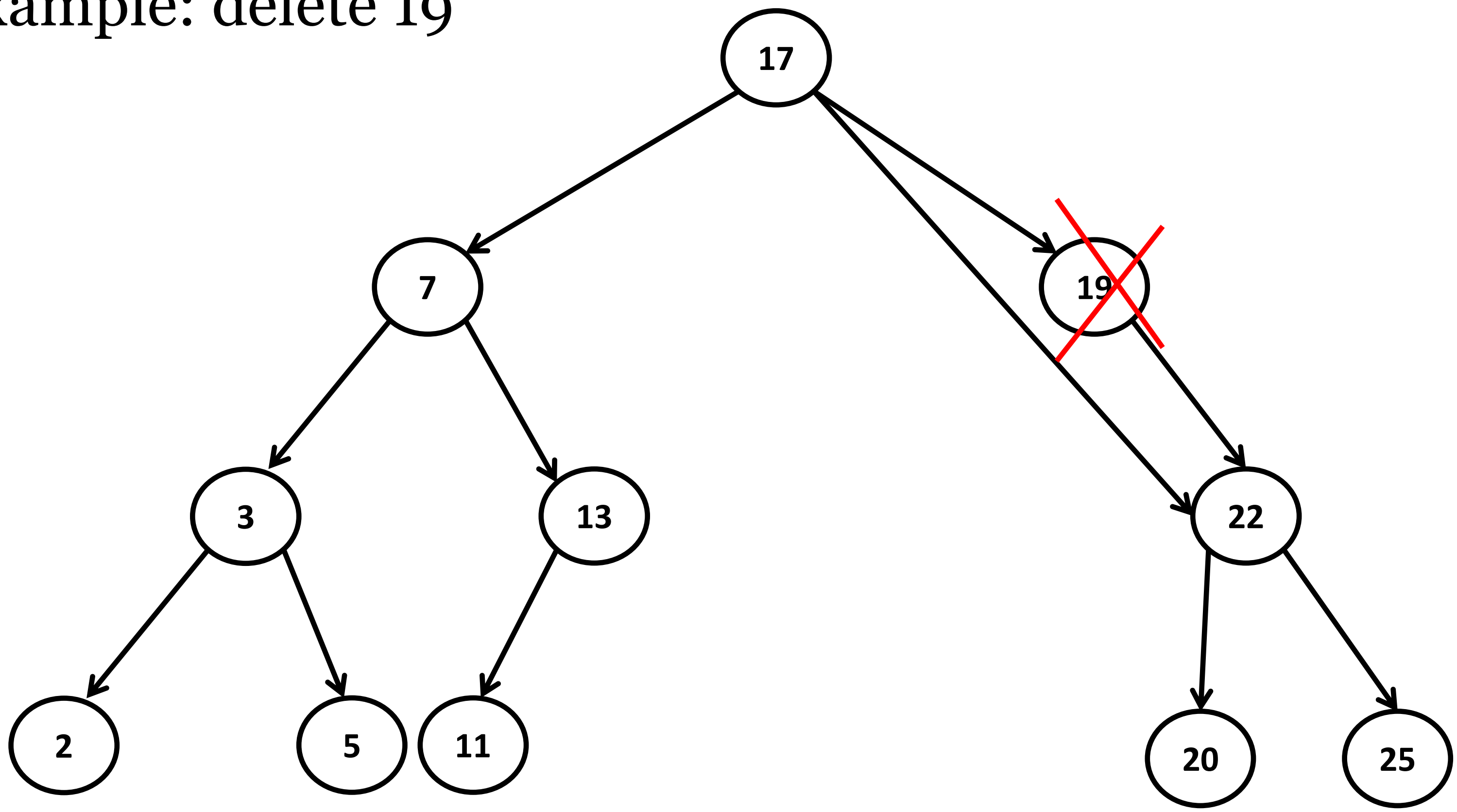
BST – deletion

- Case 2: the node to be deleted has one child.
- Example: delete 13



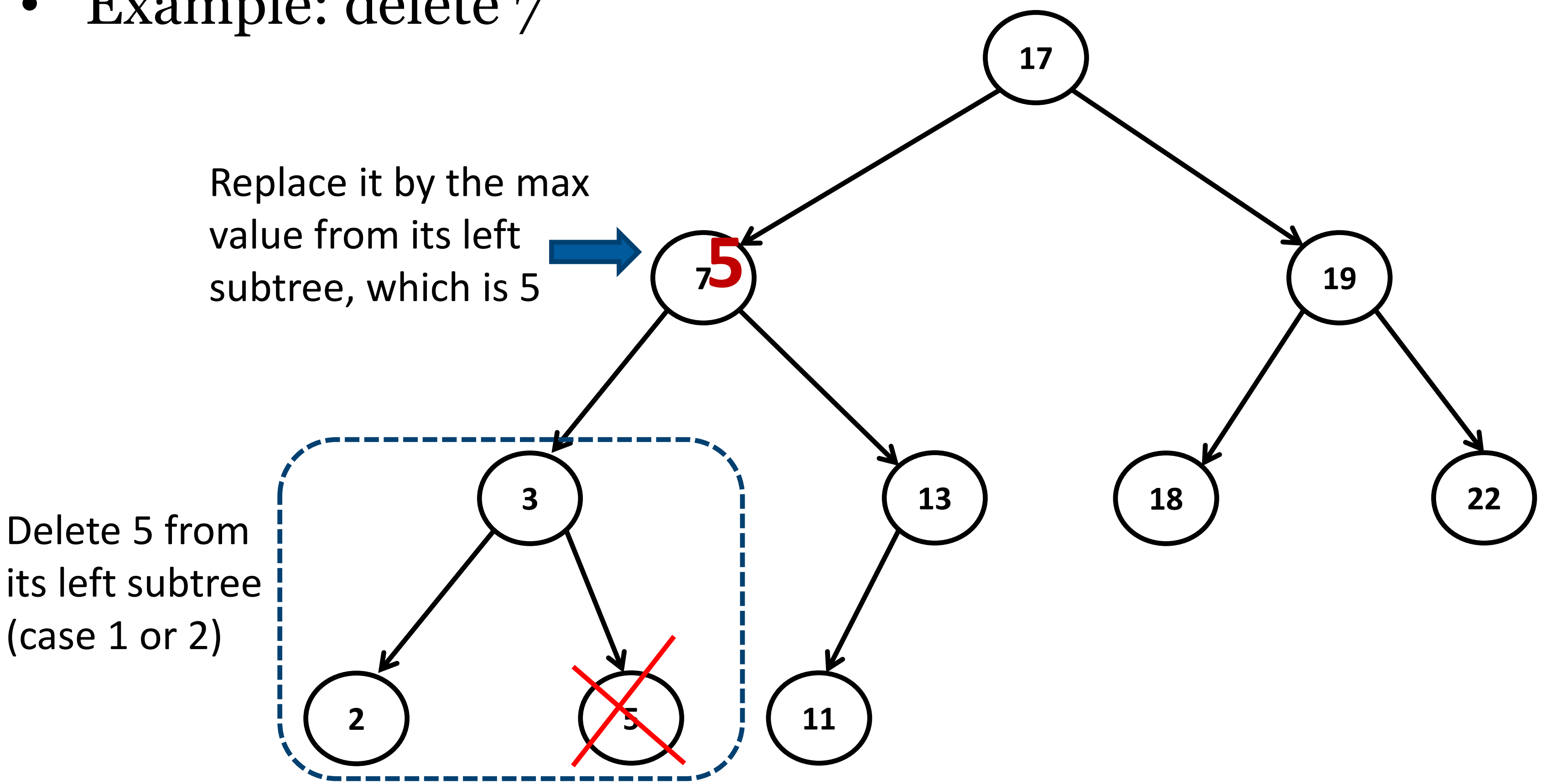
BST – deletion

- Case 2: the node to be deleted has one child.
- Example: delete 19



BST – deletion

- Case 3: the node to be deleted has both children.
- Example: delete 7



BST – performance

- Searching, insertion, and deletion all take Θ(height) time in the worst case where height is at most n-1.
- If height is k, then n is at most $1+2+\ldots+2^k = 2^{k+1}-1$.
  - $n \le 2^{k+1}-1$
  - $k \ge \log(n+1)-1$ -> height is at least logarithmic in n.
- For random insertion, average height ≤ 2.989 log(n).
- Thus, we can claim that searching, insertion, and deletion all take logarithmic time **on average**. (All three operations take linear time in the worst case).