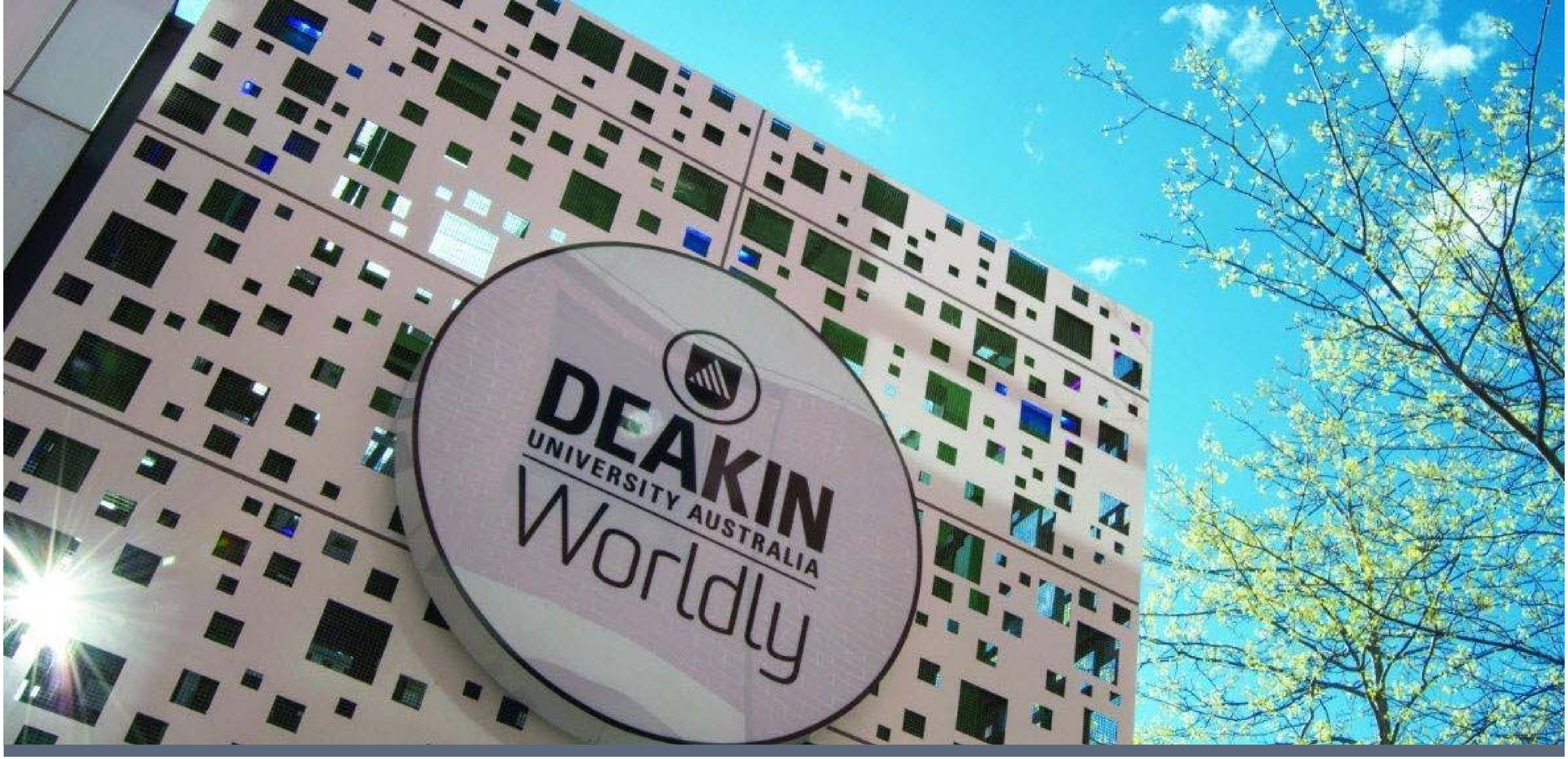
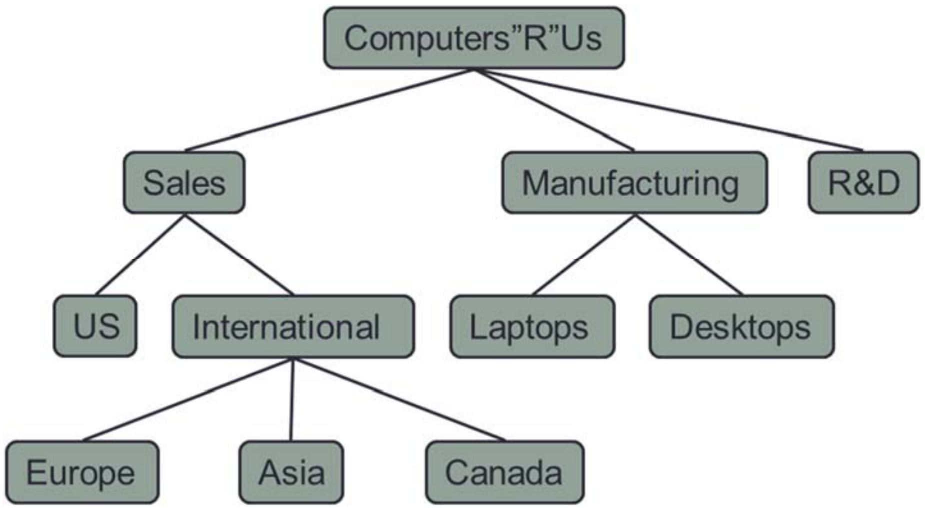



# Lecture 6. Binary Search Trees. Binary Heaps

SIT221 Data Structures and Algorithms

# Tree: Definition

- In computer science, a tree is an abstract model of a hierarchical structure.
- A tree consists of nodes with a parent-child relation (inspired by family trees).
- Every node except one has a unique parent.
- Applications:
  - Organization charts
  - File systems
  - Programming environments

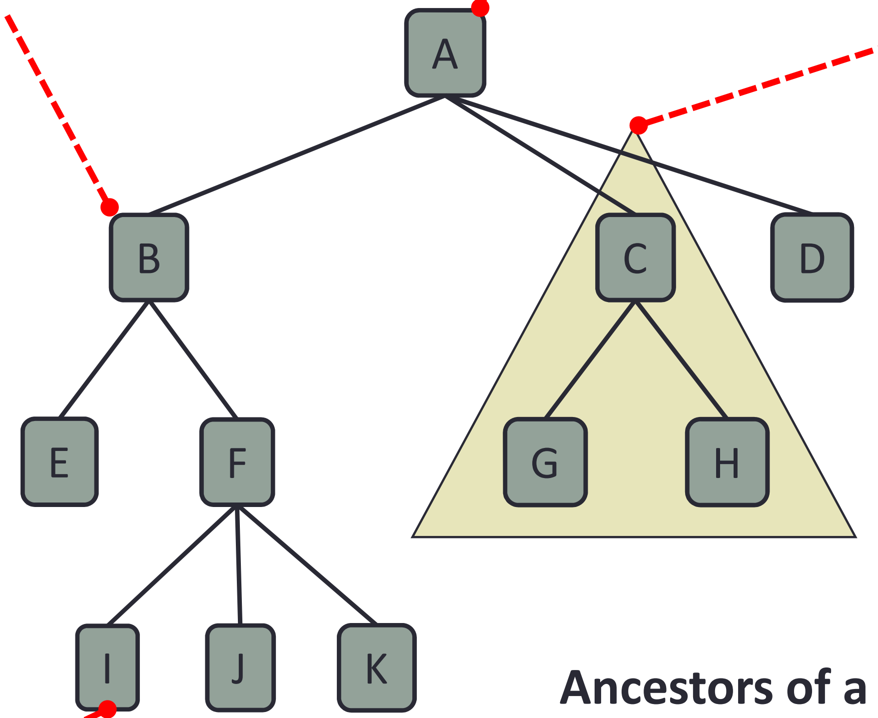


# Tree: Terminology

**Root:** unique node without a parent

**Internal node:** node with at least one child (A, B, C, F)

**Subtree:** tree consisting of a node and its descendants




External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)

**Ancestors of a node:** parent, grandparent, etc.

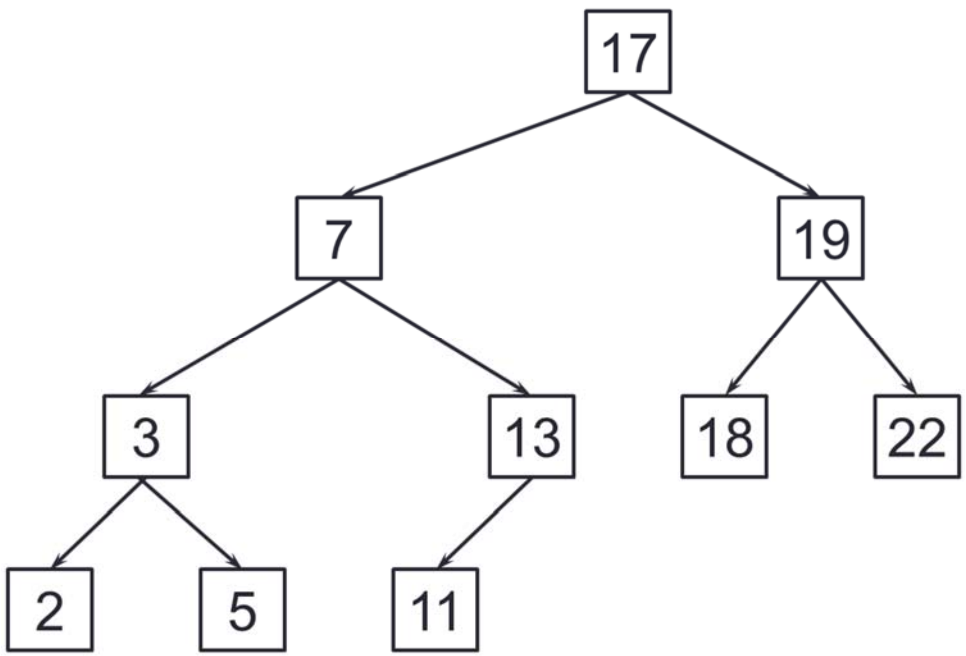
**Depth of a node:** number of ancestors

**Descendant of a node:** child, grandchild, etc.

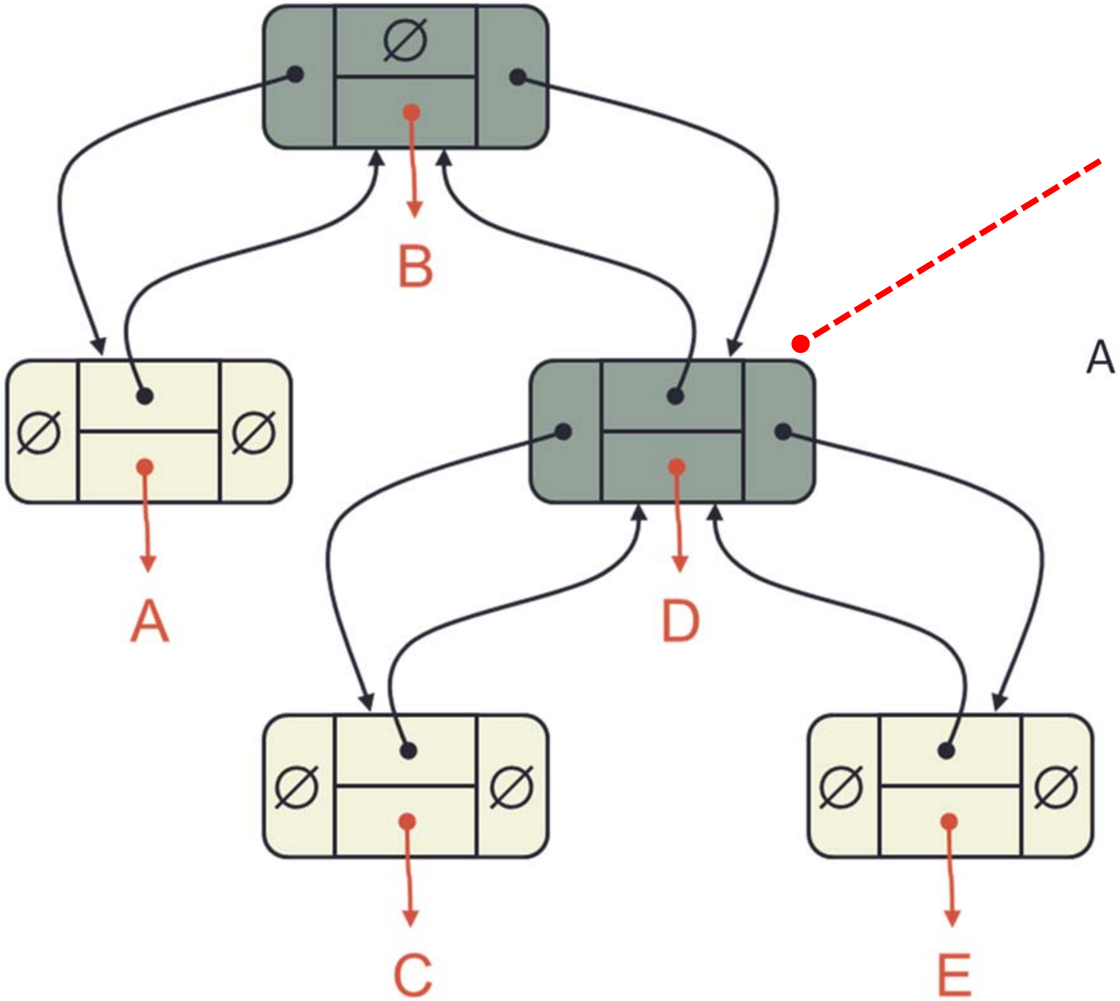
**Height of a tree:** maximum depth of any node (3)

# Binary Search Trees: Idea

- We want to have a pointer-based representation of trees
- Each node
  - stores an element
  - has a pointer to the left subtree (might be null)
  - has a point to the right subtree (might be null)



# Binary Search Trees: Linked Structure




**NODE**

<table>
<tr><td colspan="3">link (parent)</td></tr>
<tr><td>link<br>(left child)</td><td>data</td><td>link<br>(right child)</td></tr>
</table>

A node is represented by an object storing

- Element
- Parent node
- Left child node
- Right child node

# Binary Search Trees

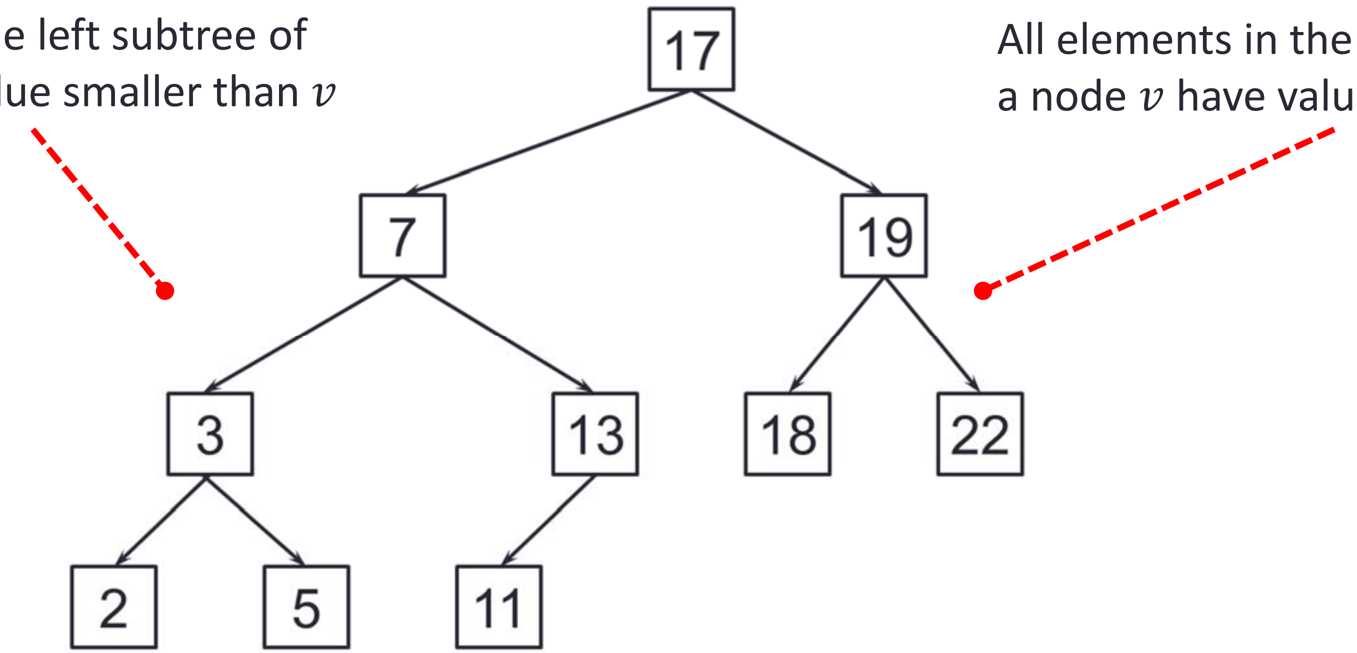
A ***binary search tree*** is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

Let $u$, $v$, and $w$ be three nodes such that $u$ is in the left subtree of $v$ and $w$ is in the right subtree of $v$. Then we have

$$key(u) \leq key(v) \leq key(w)$$

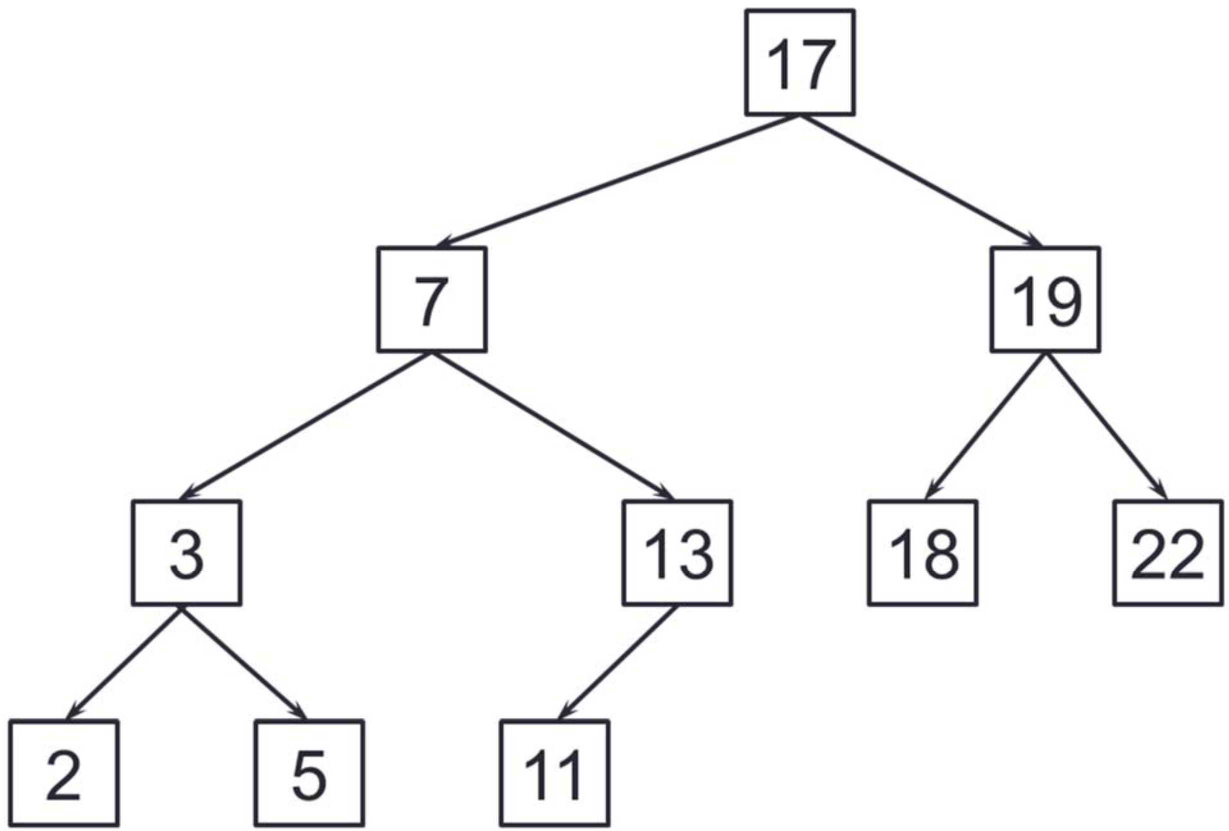
All elements in the left subtree of a node $v$ have value smaller than $v$

All elements in the right subtree of a node $v$ have value larger than $v$



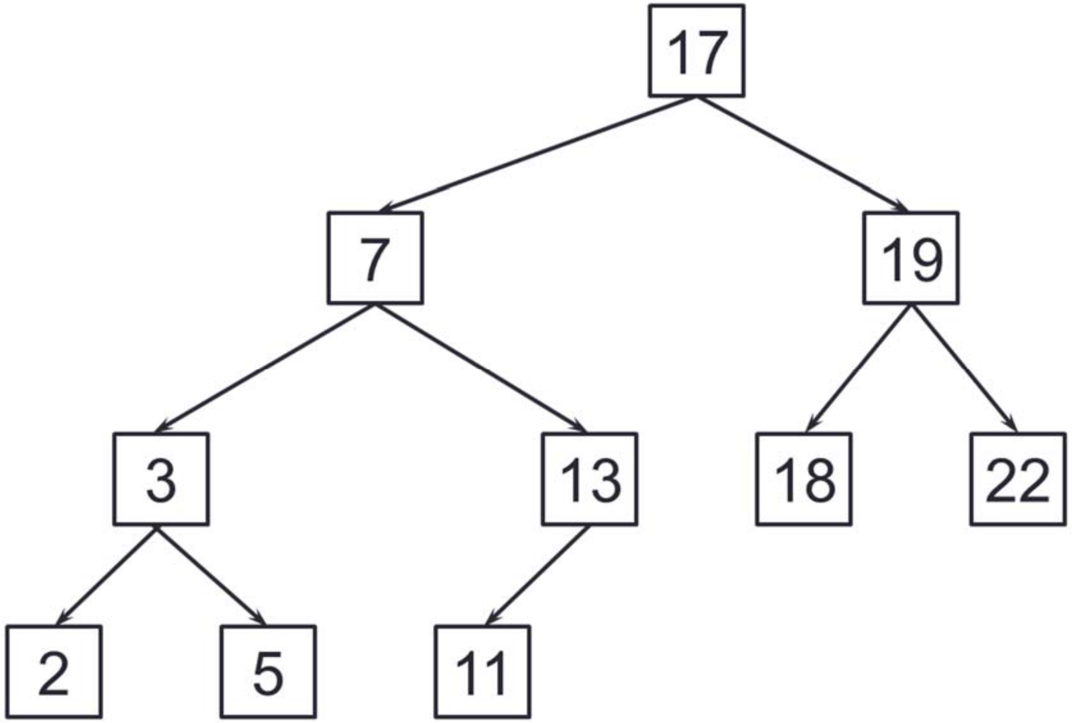
# Binary Search Trees: Tree Traversal

- Want to visit every node in the tree (and print out the elements).
- Want to have a recursive formulation for tree traversal.



# Binary Search Trees: Preorder Traversal

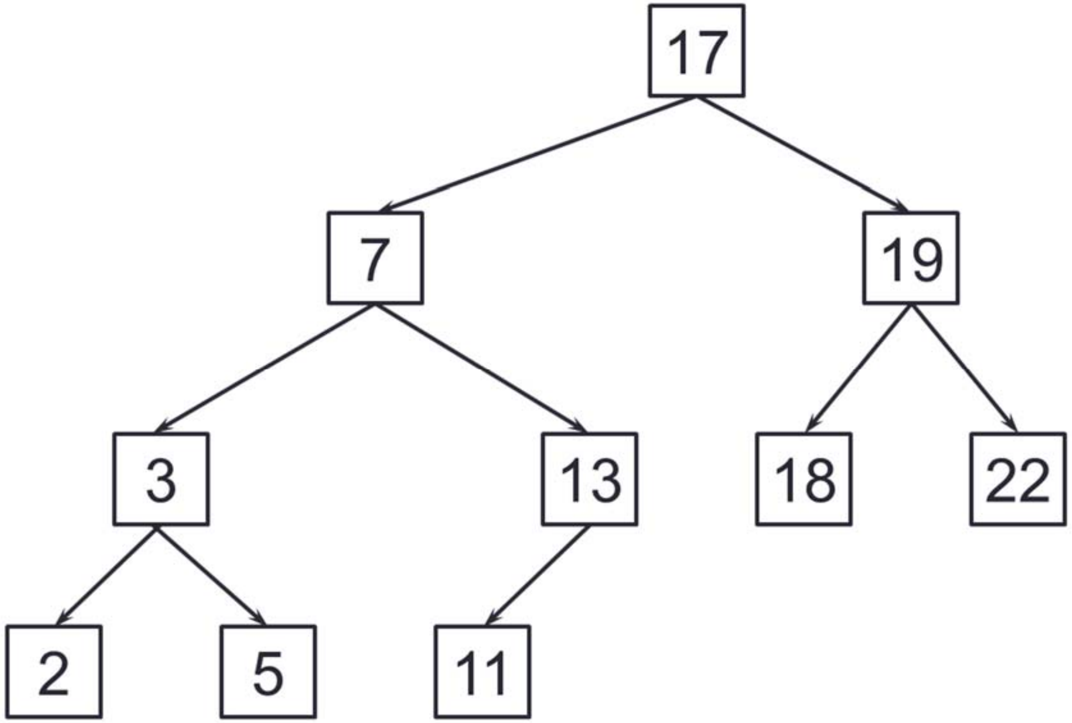
1. Visit the root (and print out the element)
2. If (node.left !=null) Preorder(node.left)
3. If (node.right !=null) Preorder(node.right)




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

# Binary Search Trees: Postorder Traversal

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

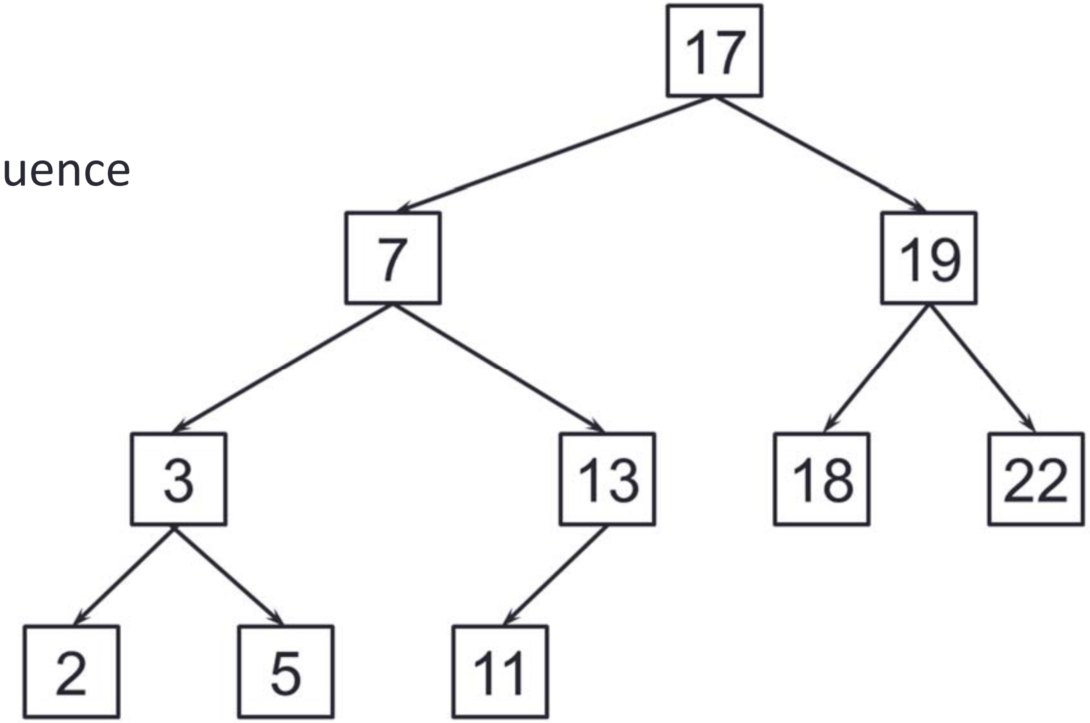



Order nodes are visited: 2, 5, 3, 11, 13, 7, 18, 22, 19, 17

# Binary Search Trees: Inorder Traversal

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

Observation:
The resulting sequence is sorted




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

# Binary Search Trees: Basic Operations

- Find an element $e$ in the binary search tree
- Insert an element $e$ into the binary search tree
- Delete an element $e$ from the binary search tree

Want to have all these operations implemented in time $O(\log \mathrm{n})$.

# Binary Search Trees: Find Operation

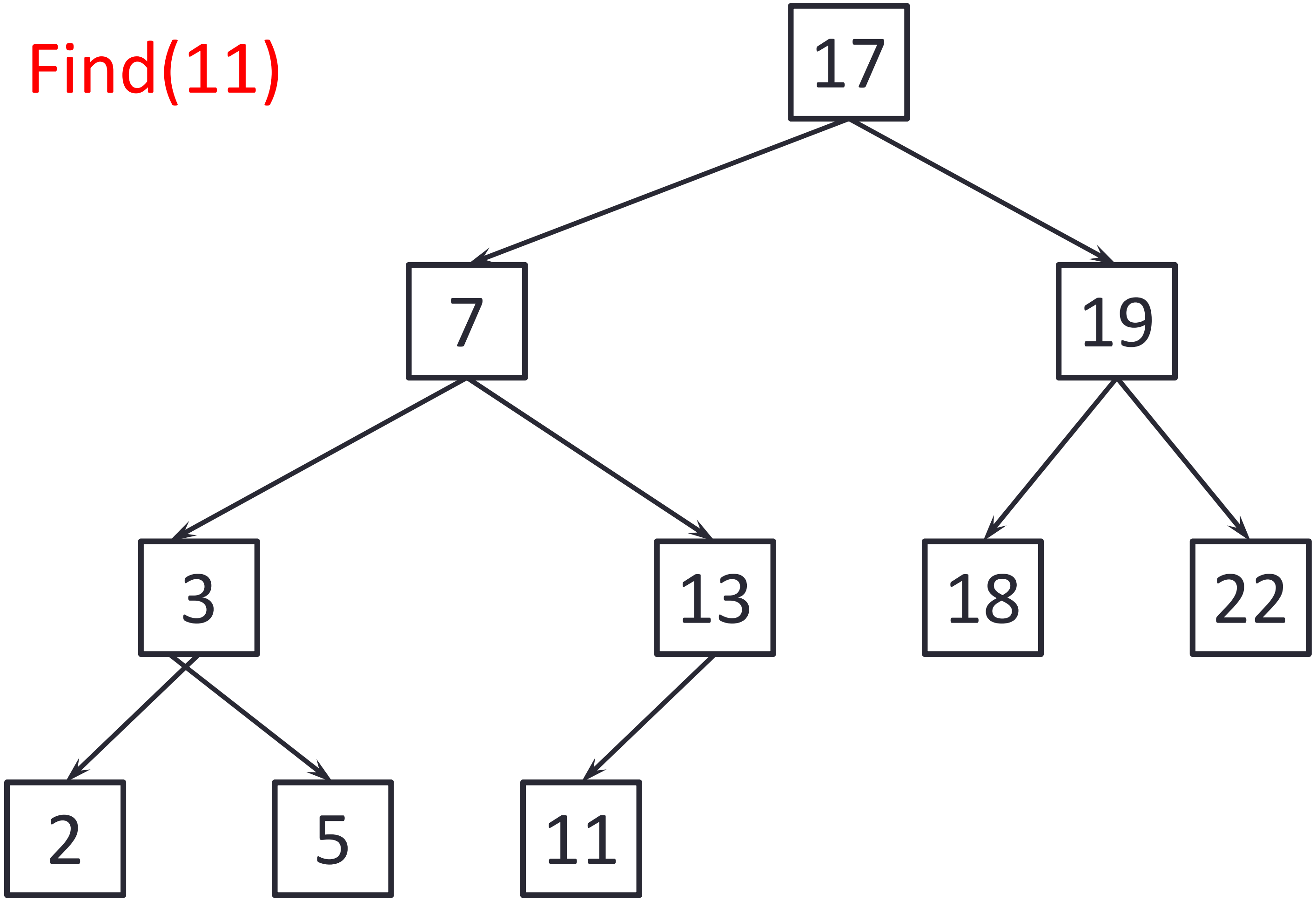
- To search for a key $k$, we trace a downward path starting at the root.
- The next node visited depends on the outcome of the comparison of $k$ with the key of the current node.
- If we reach a leaf, the key is not found and we return null.

## Find($k$)

1. Start at the root.
2. At a node $x$, compare $x$ and $k$.
   - If $k = x$, then found
   - If $k < x$, search in the left subtree of $x$.
     If subtree does not exist return not found.
   - If $k > x$, search in the right subtree of $x$.
     If subtree does not exist, return not found

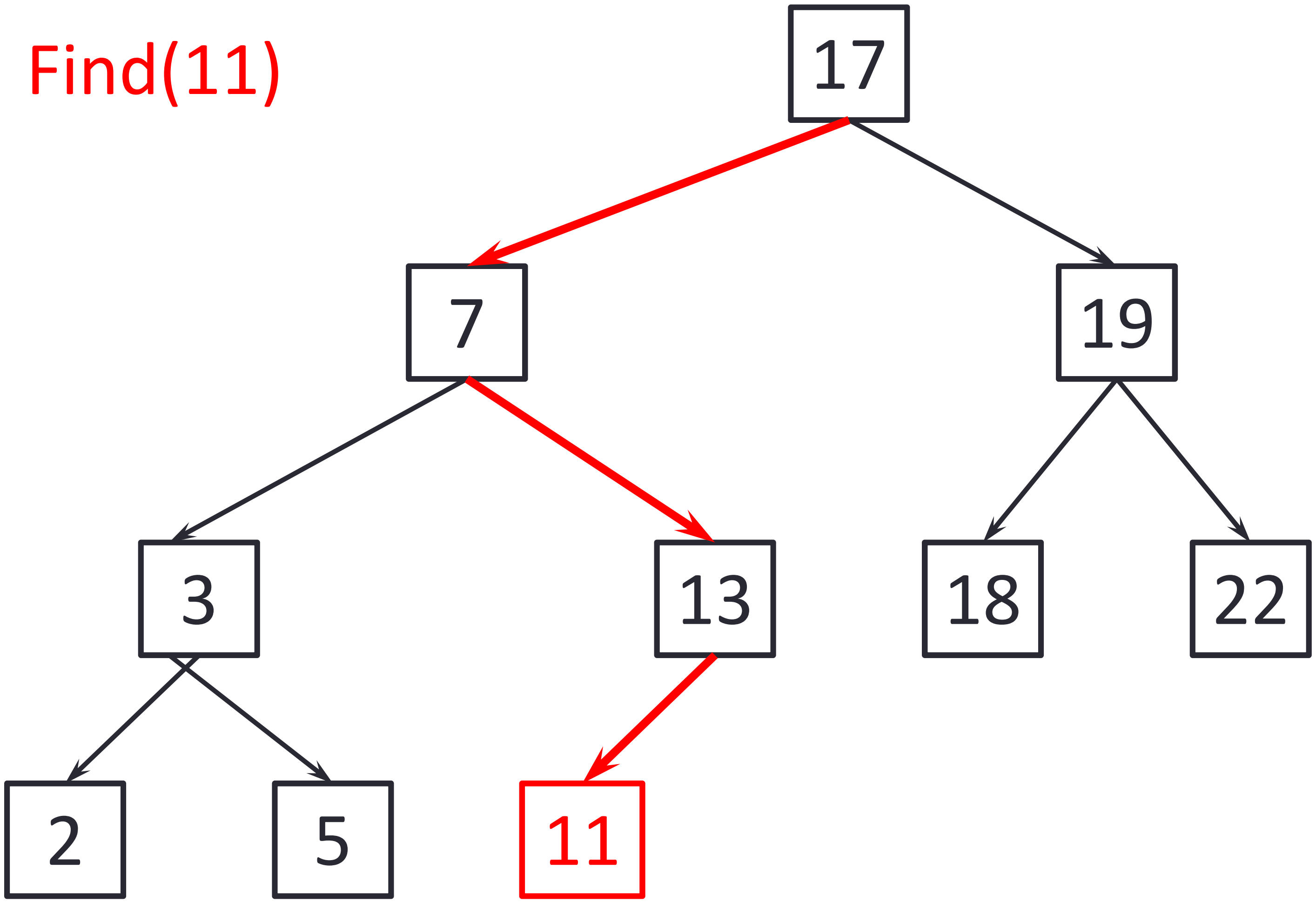
# Binary Search Trees: Find Operation

Find(11)



# Binary Search Trees: Find Operation

Find(11)



# Binary Search Trees: Insertion

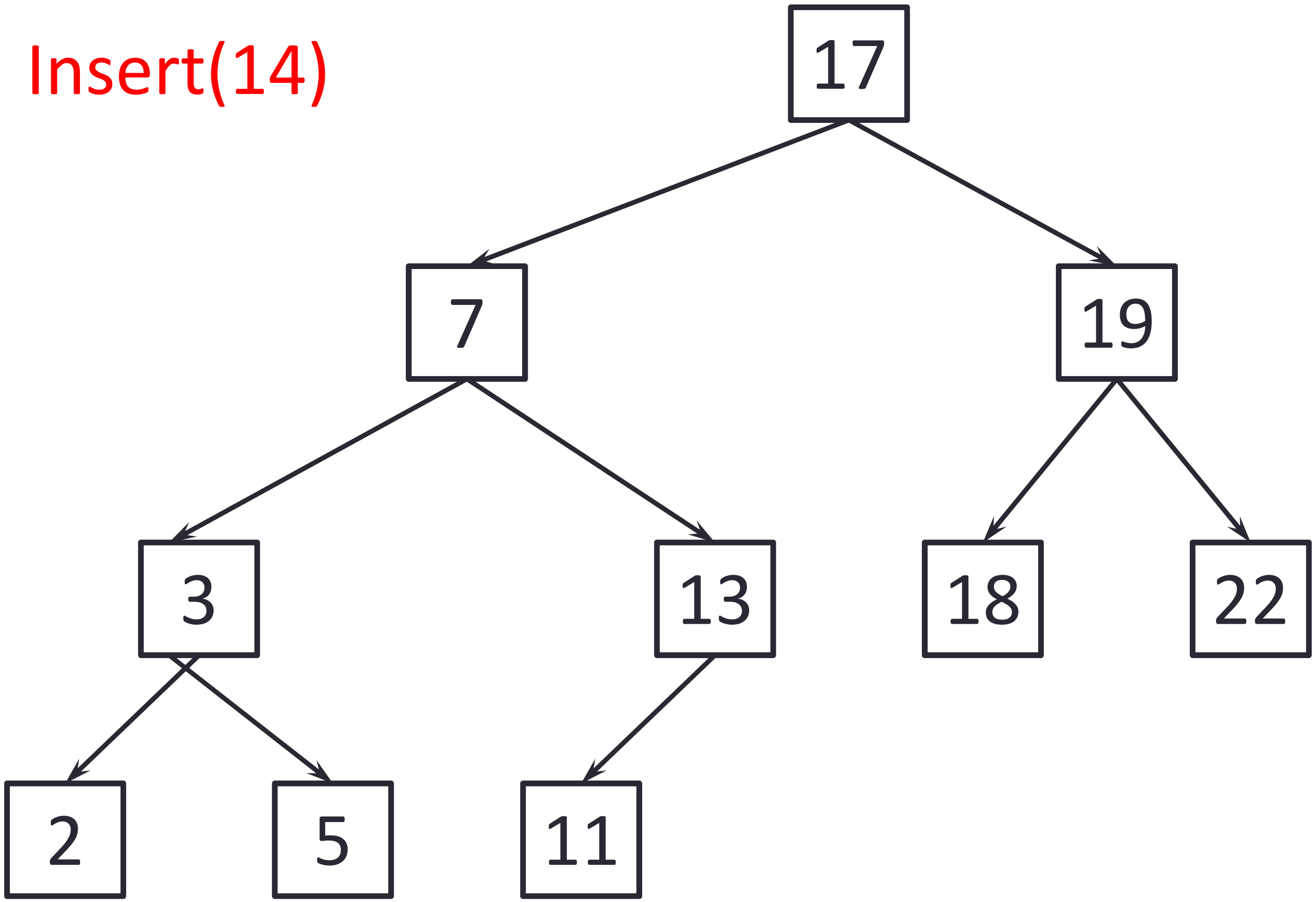
- To perform operation Insert($k$), we search for key $k$ using Find($k$)
- Assume $k$ is not yet in the tree, and let $w$ be the leaf reached by Find($k$)
- We insert $k$ at node $w$ and expand $w$ into an internal node

**Insert($k$)**

1. Find($k$)
2. If not found, element with key $k$ becomes child of the last visited node of Find($k$).

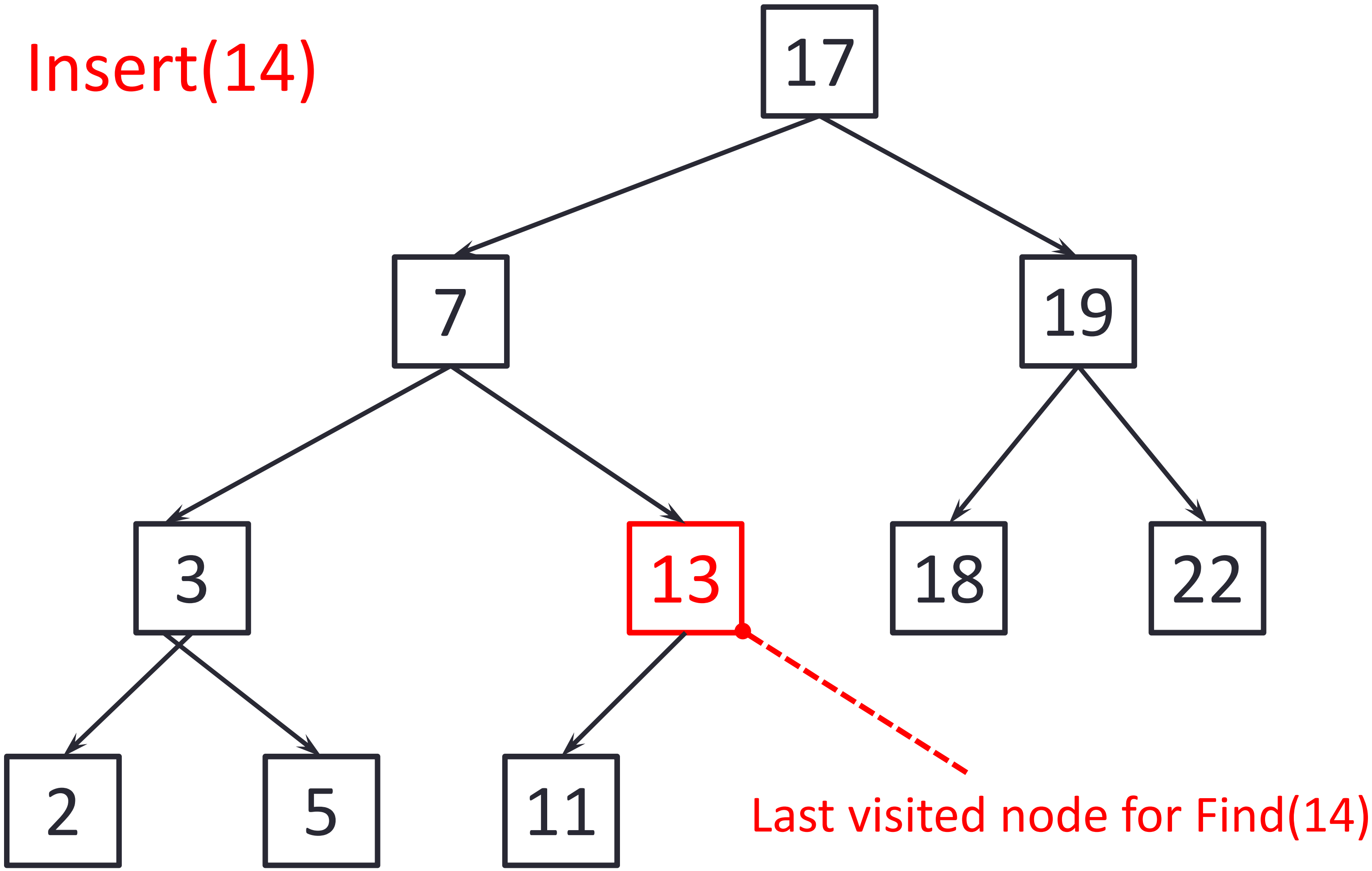
# Binary Search Trees: Insertion

Insert(14)



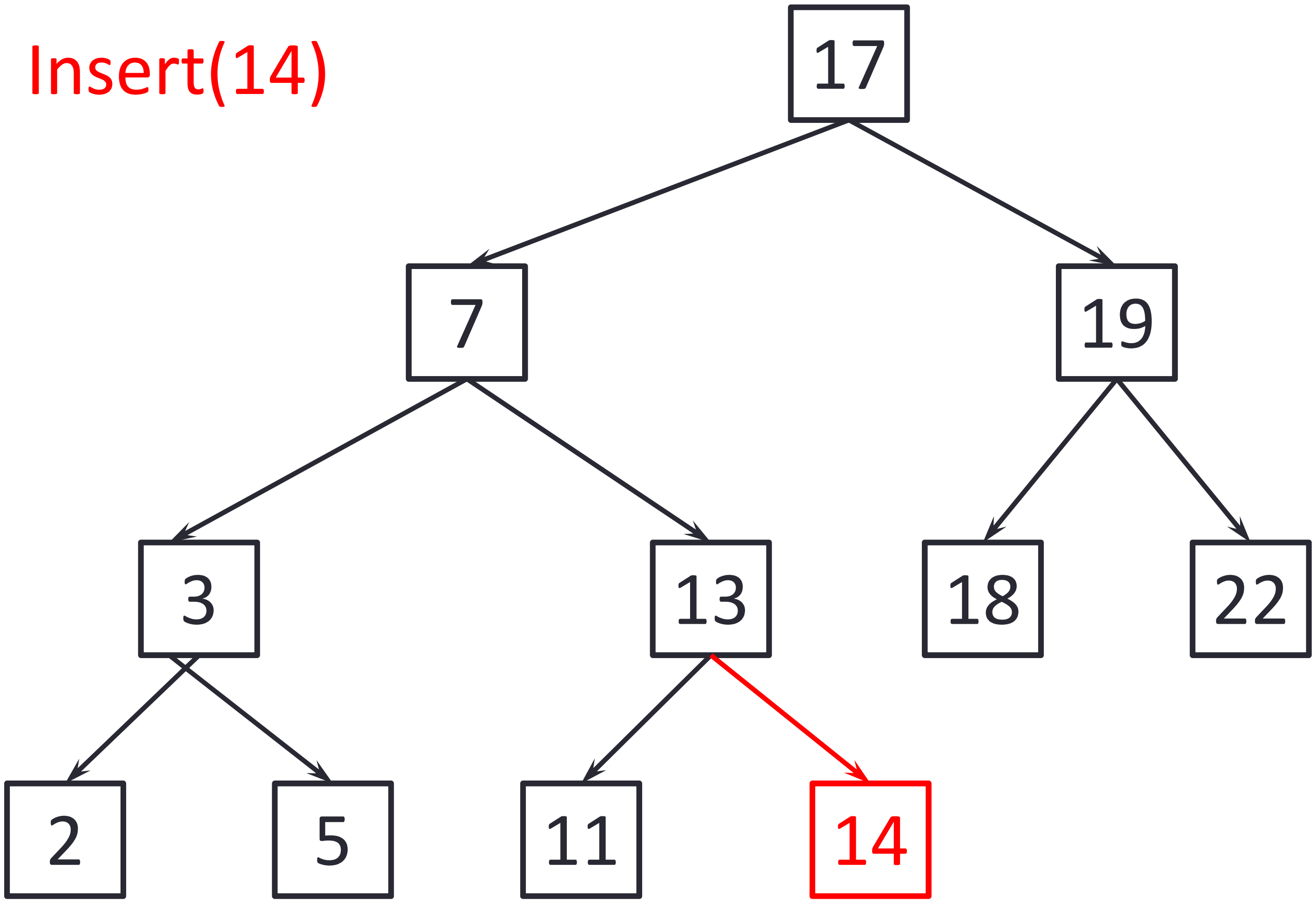
# Binary Search Trees: Insertion

Insert(14)



# Binary Search Trees: Insertion

Insert(14)



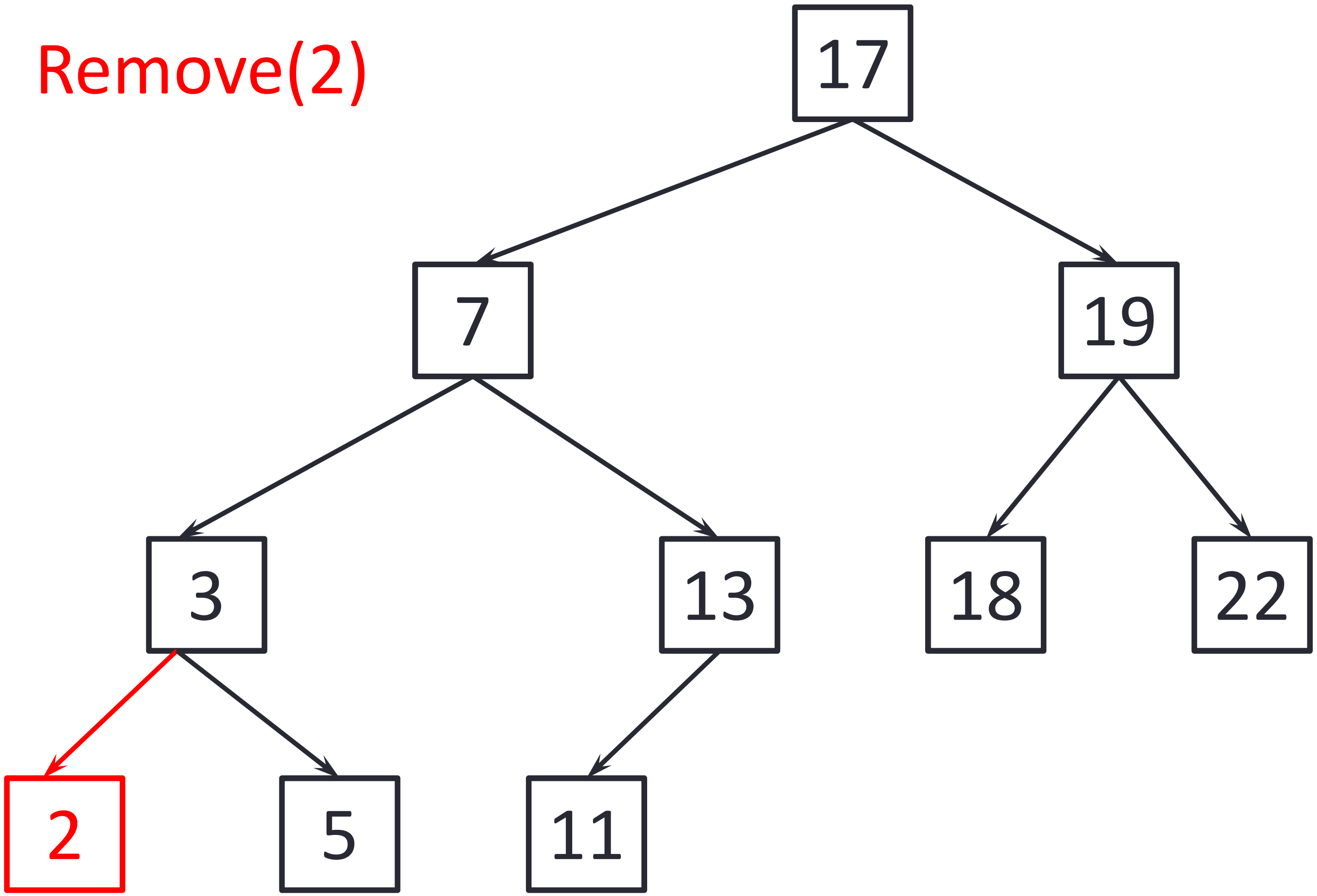
# Binary Search Trees: Deletion

**Remove($k$)**

1. Find($k$)
   - If $k$ is stored at a leaf, delete this leaf and the incoming edge.
   - If $k$ has one child $x$, redirect pointer pointing to $k$ to $x$ and delete $k$.
   - If $k$ has two children
     - search in the tree for the largest element $x$ smaller than $k$.
     - swap $x$ and $k$ and Remove($k$)

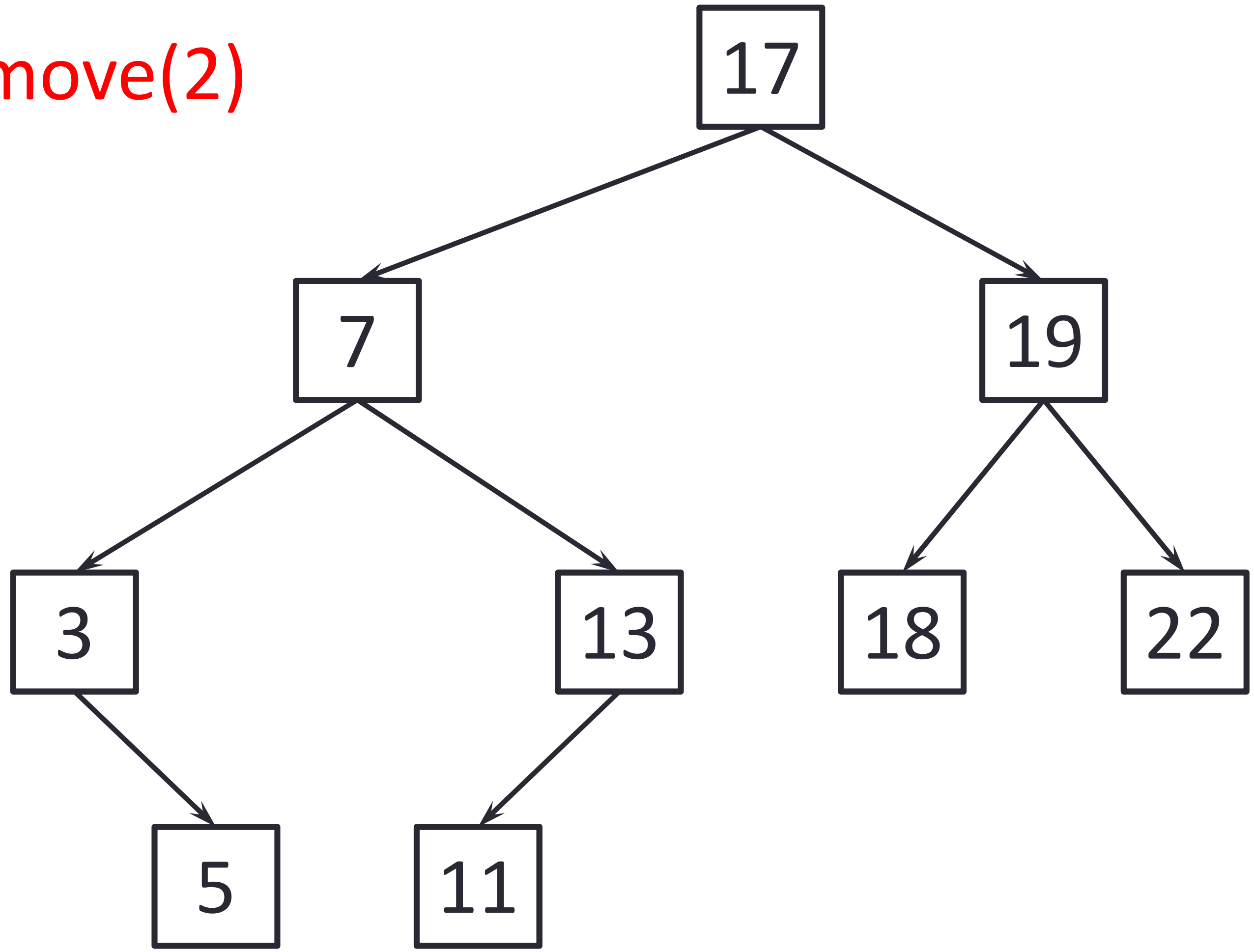
# Binary Search Trees: Deletion

Remove(2)



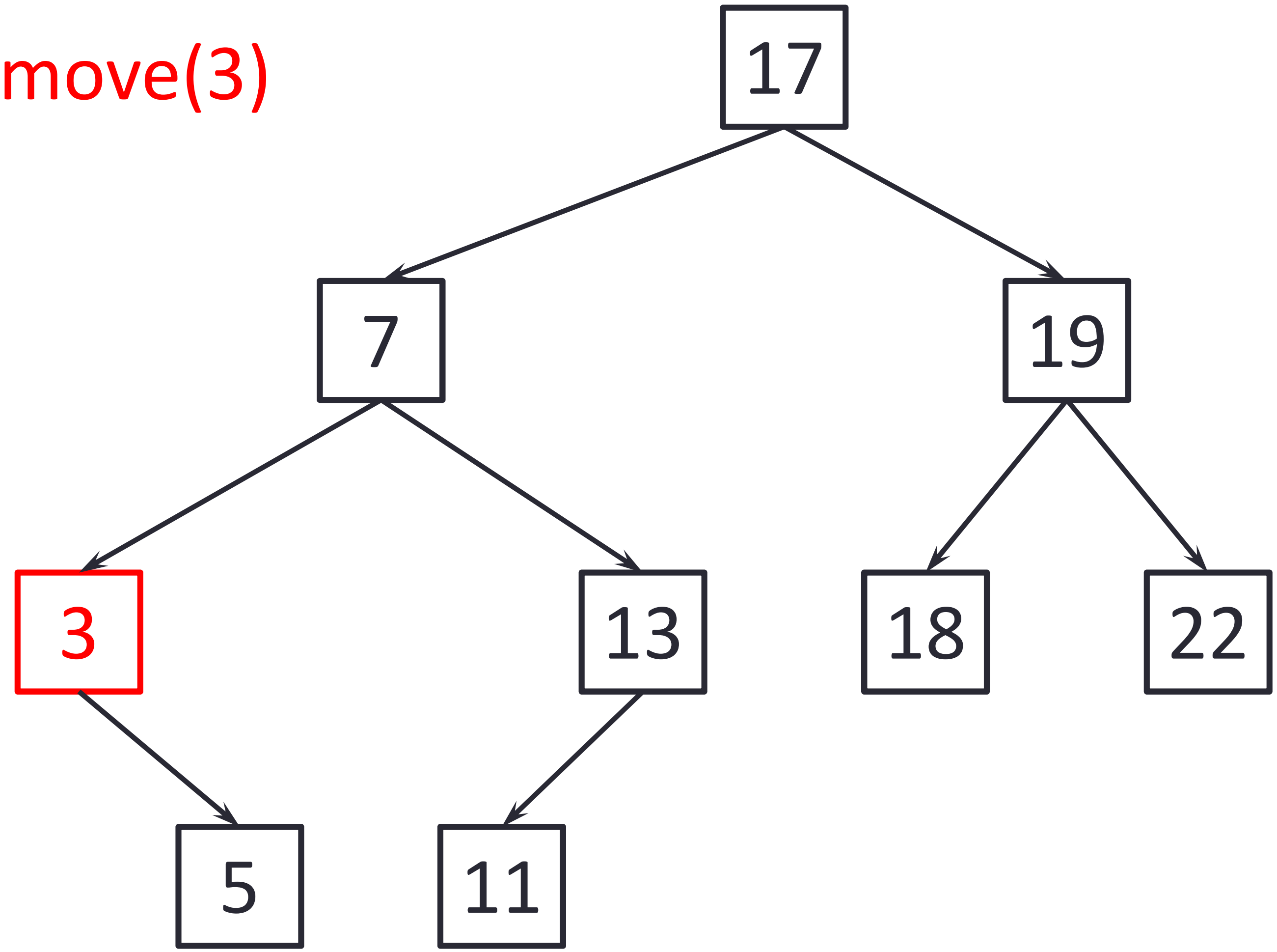
# Binary Search Trees: Deletion

Remove(2)



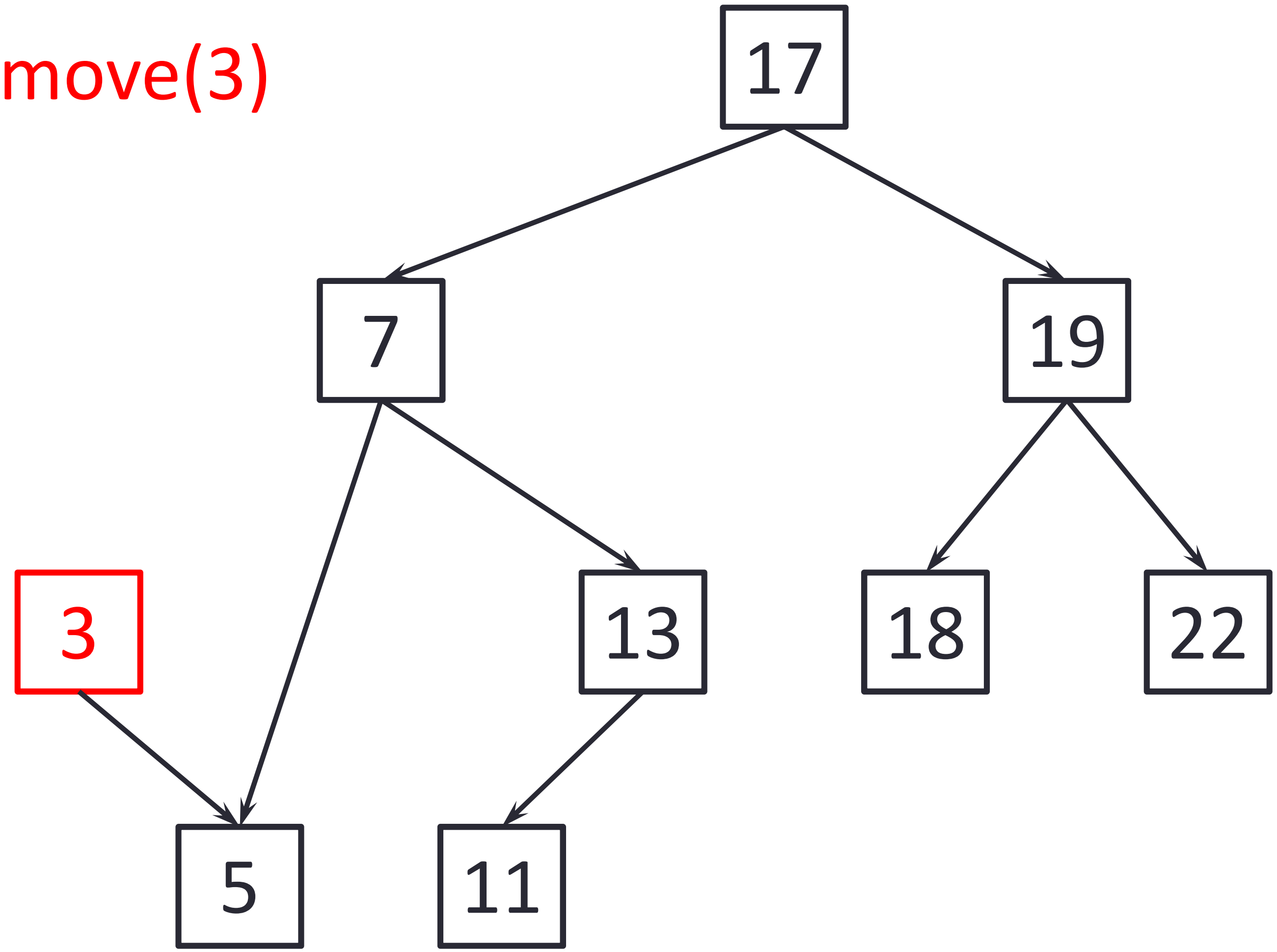
# Binary Search Trees: Deletion (node with 1 child)

Remove(3)



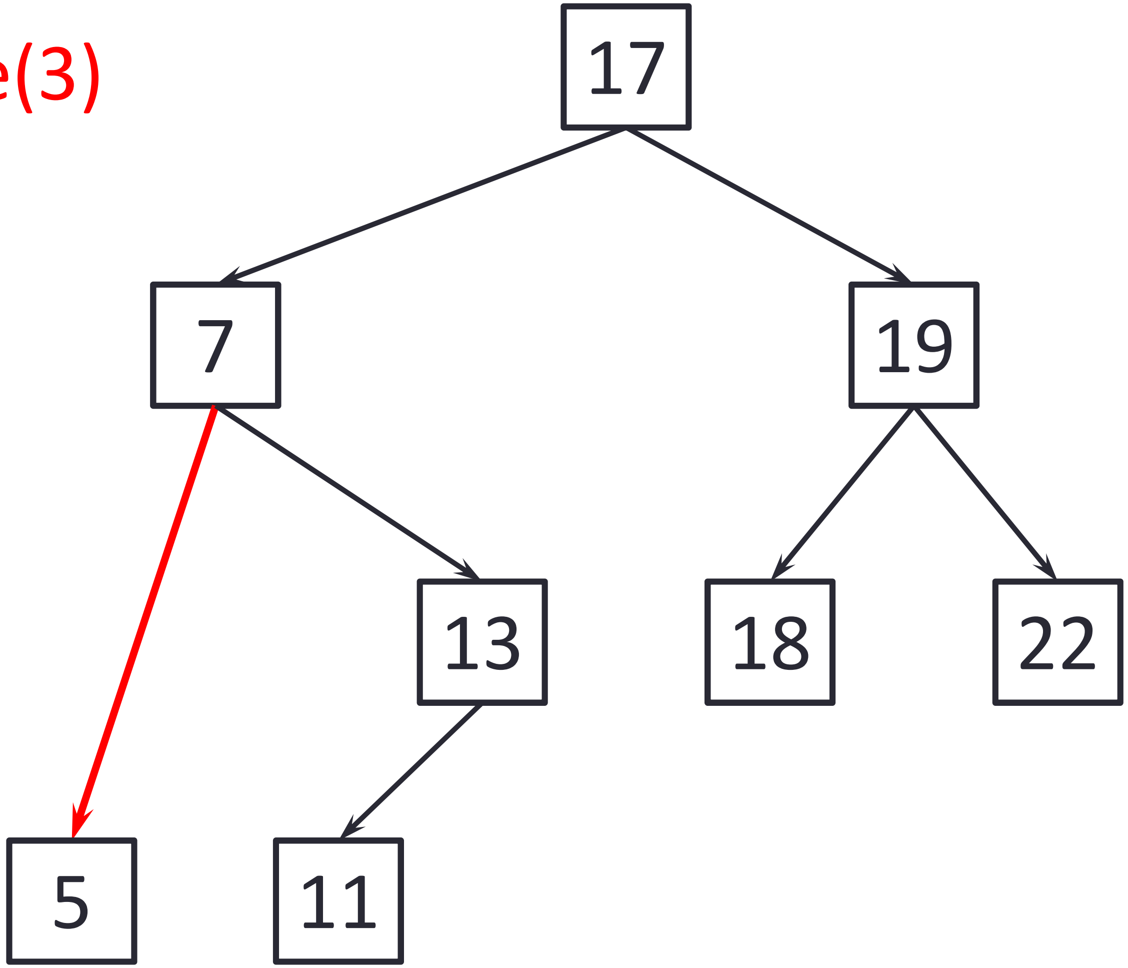
# Binary Search Trees: Deletion (node with 1 child)

Remove(3)



# Binary Search Trees: Deletion (node with 1 child)

Remove(3)

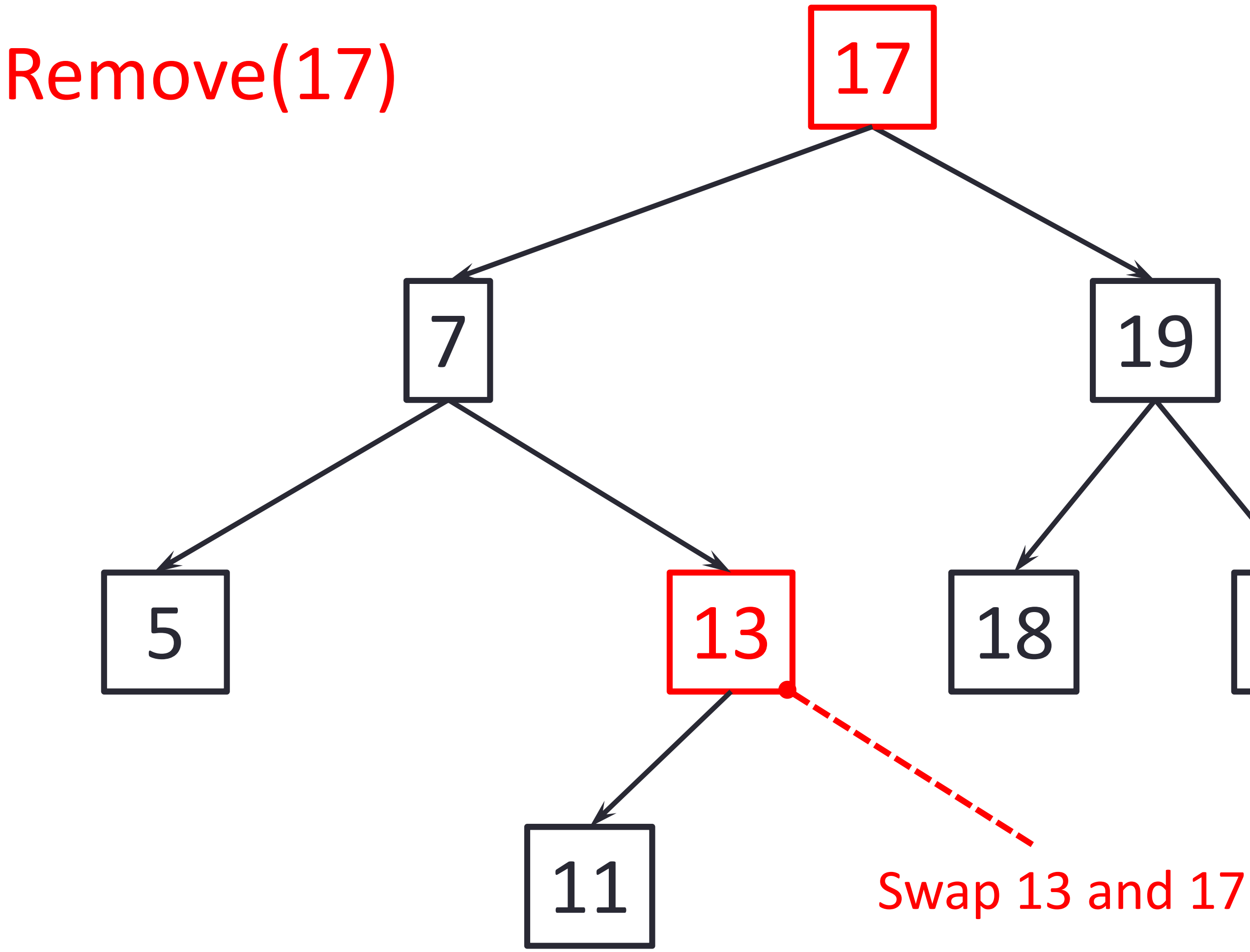


# Binary Search Trees: Deletion (node with 2 children)

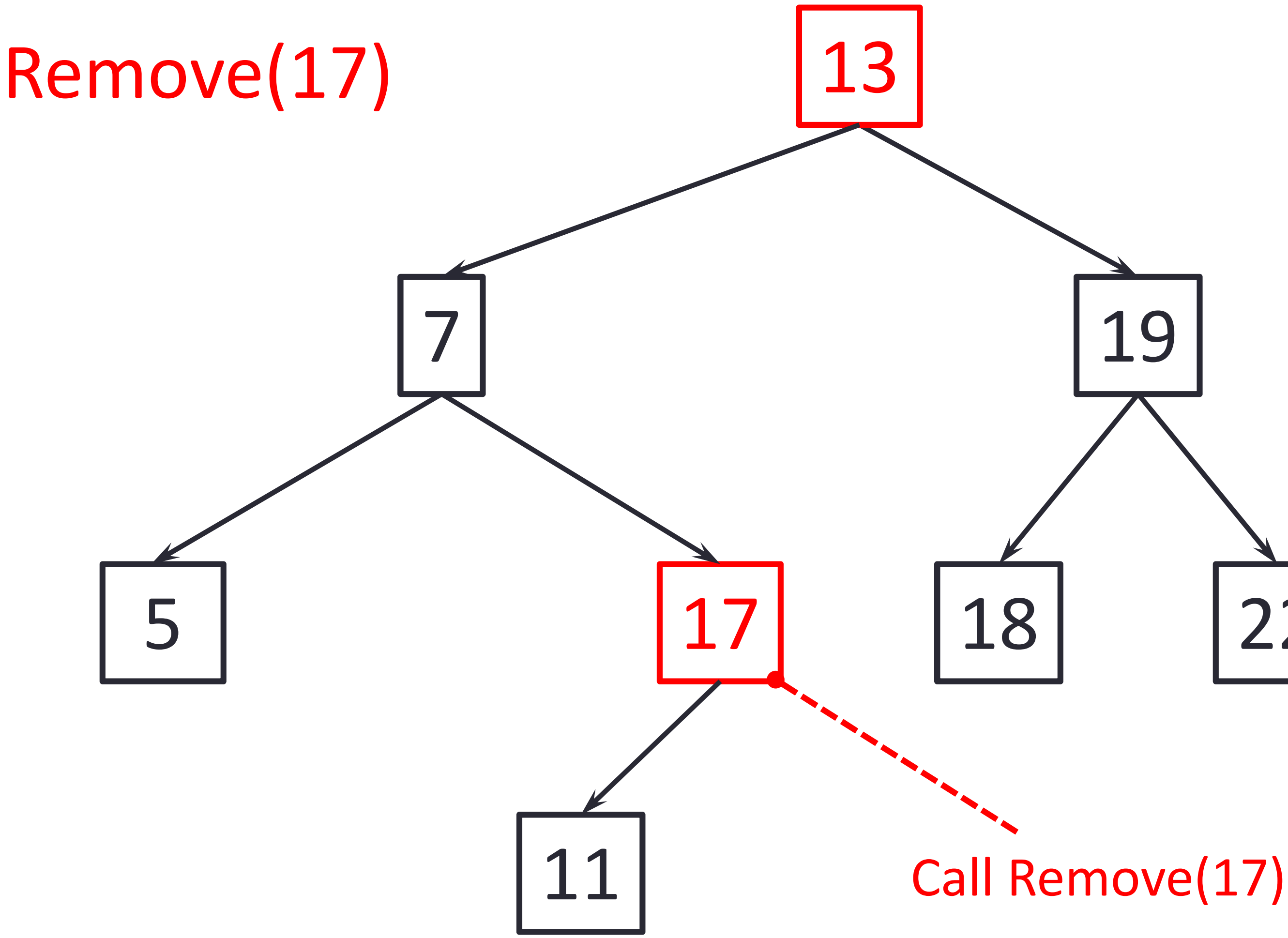
How to find the largest element smaller than $k$?

- Go to the left child of $k$ (has to exist as $k$ has two children.
- Follow the pointer to the right as long as possible.

# Binary Search Trees: Deletion (node with 2 children)

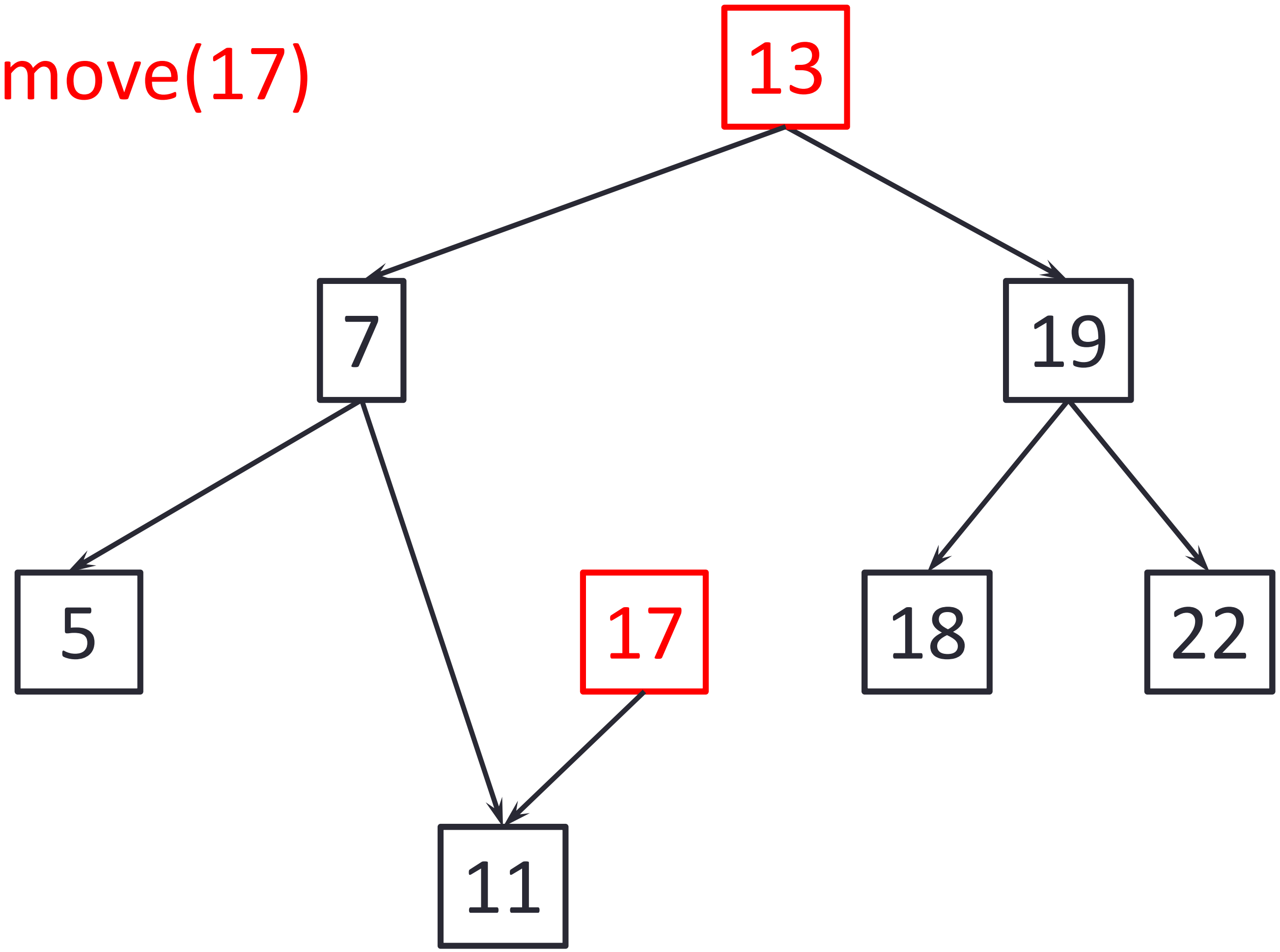


# Binary Search Trees: Deletion (node with 2 children)



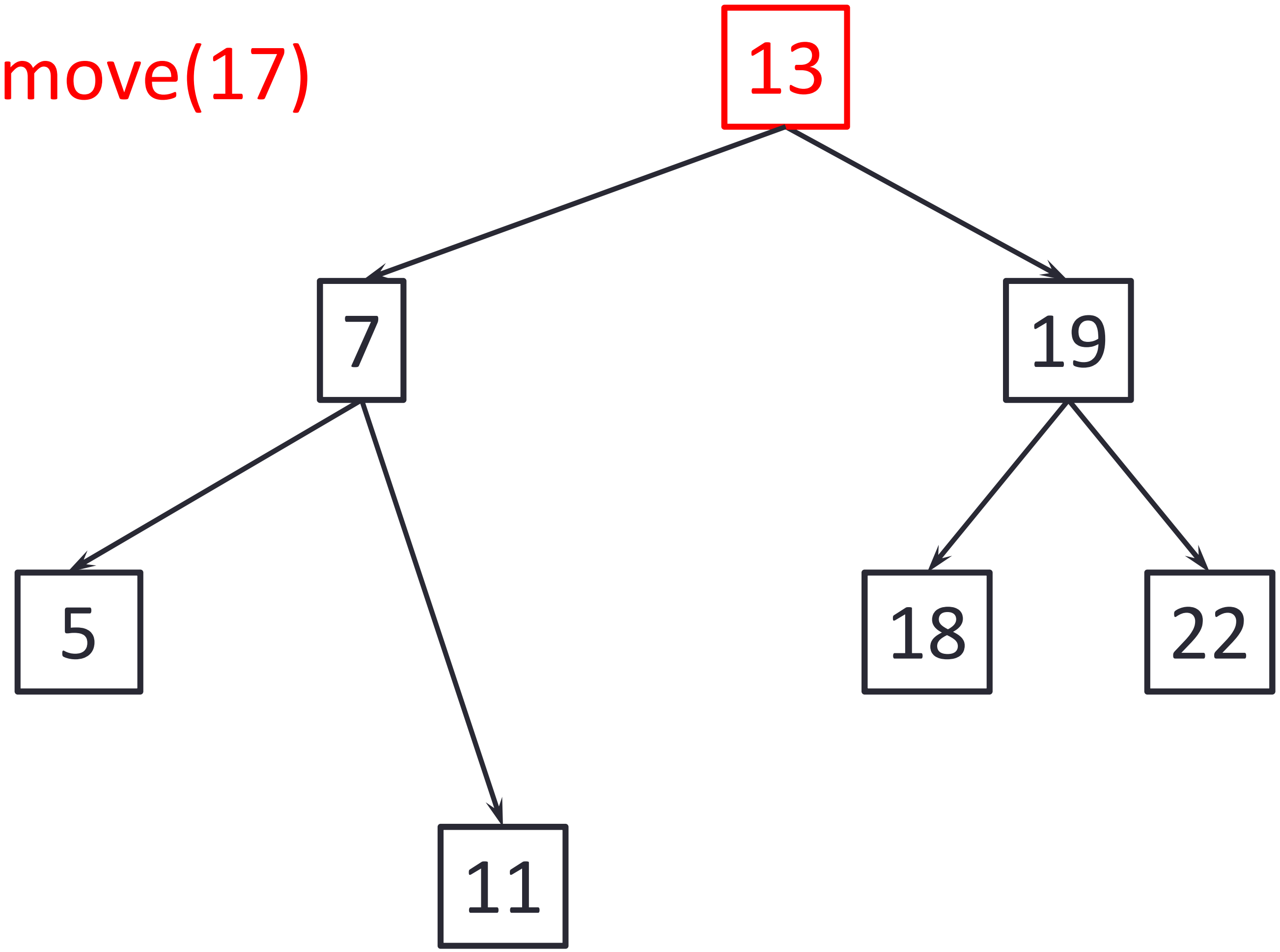
# Binary Search Trees: Deletion (node with 2 children)

Remove(17)



# Binary Search Trees: Deletion (node with 2 children)

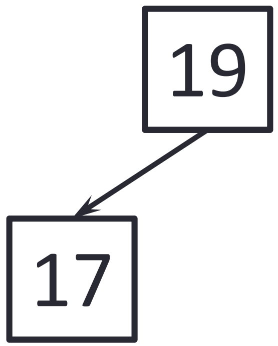
Remove(17)



# Binary Search Trees: Build

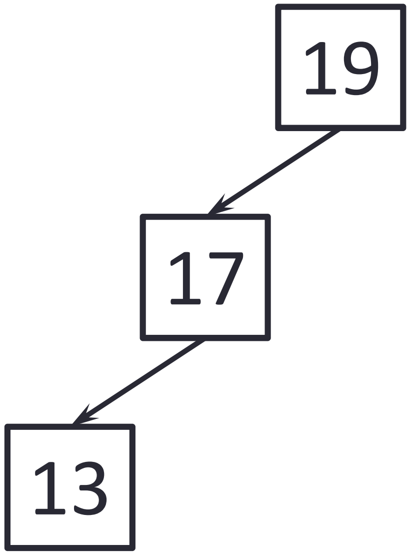

Insert 17

# Binary Search Trees: Build



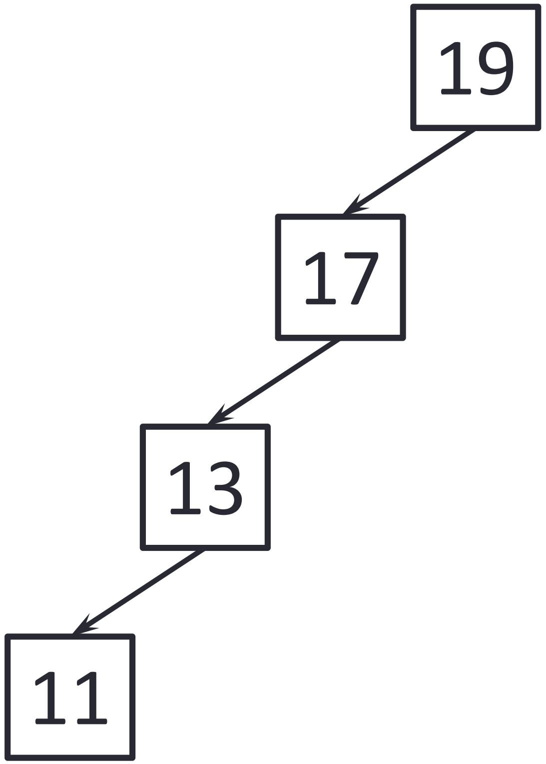

Insert 13

# Binary Search Trees: Build



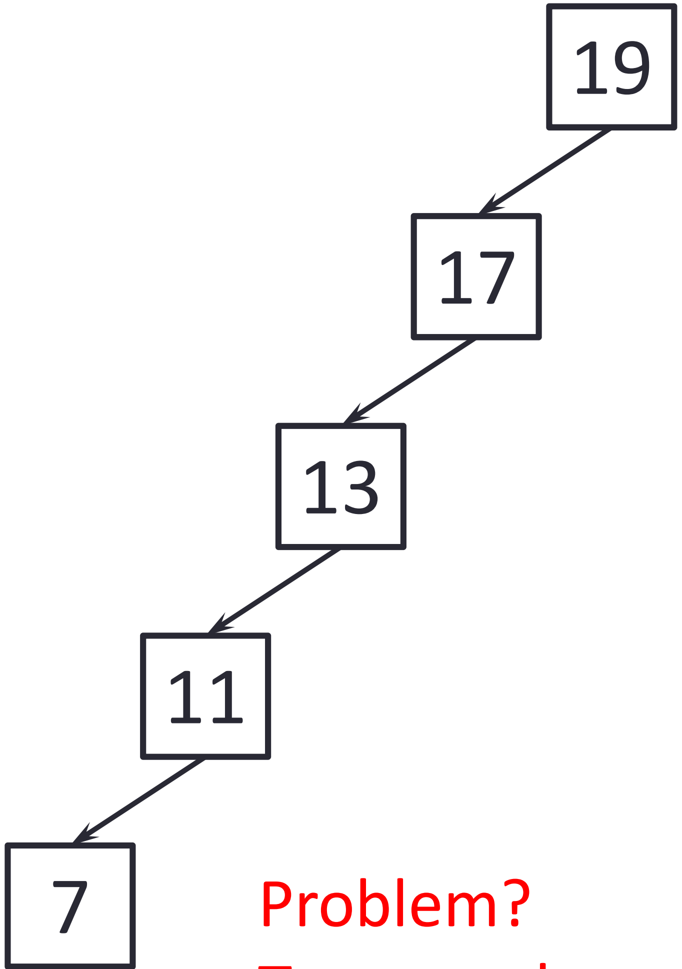

Insert 11

# Binary Search Trees: Build




Insert 7

# Binary Search Trees: Build

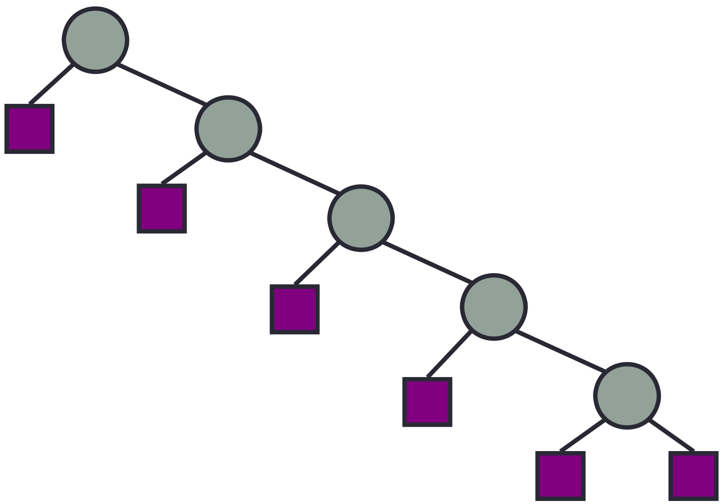


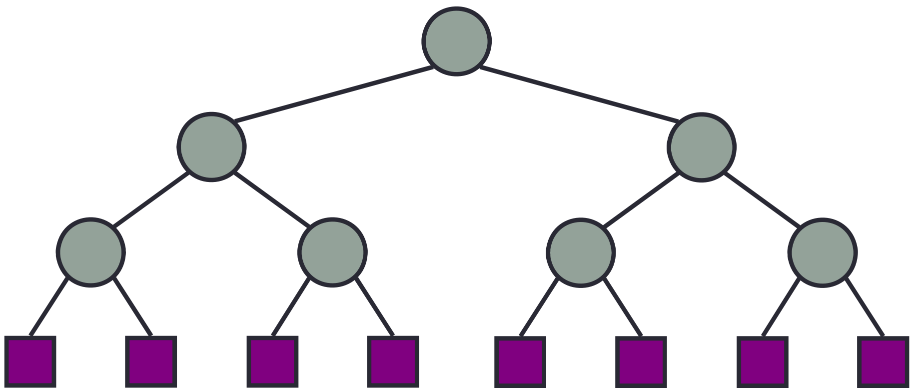

Problem?
Tree can degenerate to a list.

# Binary Search Trees: Build

The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case

- space complexity $O(n)$
- methods Find($k$), Insert($k$) and Remove($k$) take $O(h)$ time





# Perfectly Balanced Binary Search Trees

A binary search tree is perfectly balanced if it has height $\lfloor \log n \rfloor$.

(height is the length of the longest path from the root to a leaf)

Number of nodes in a perfectly balanced tree of height $k$:

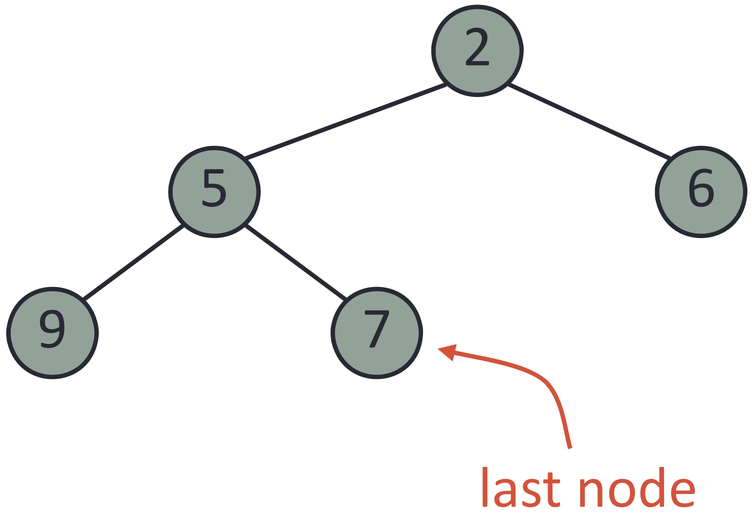
$$\sum_{i=0}^{k} 2^i = 1 + 2 + 4 + \ldots 2^k = 2^{k+1} - 1$$

# Binary Heaps

A heap is a binary tree storing keys at its nodes and satisfying the following properties:

- **Heap-Order:** for every internal node v other than the root, ***key***(***v***) $\geq$ ***key***(***parent***(***v***))
- **Complete Binary Tree:** let *h* be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are $2^i$ nodes of depth $i$
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

The last node of a heap is the rightmost node of depth *h*



# Binary Heaps: Properties

- Smallest element is stored at the root
- Each node is smaller than its children
- All levels are completely filled (except perhaps the last one)
- Last level is filled from left to right

**Observation:** The children of a node with index $i$ have indices $2i$ and $2i + 1$ (if they exist)

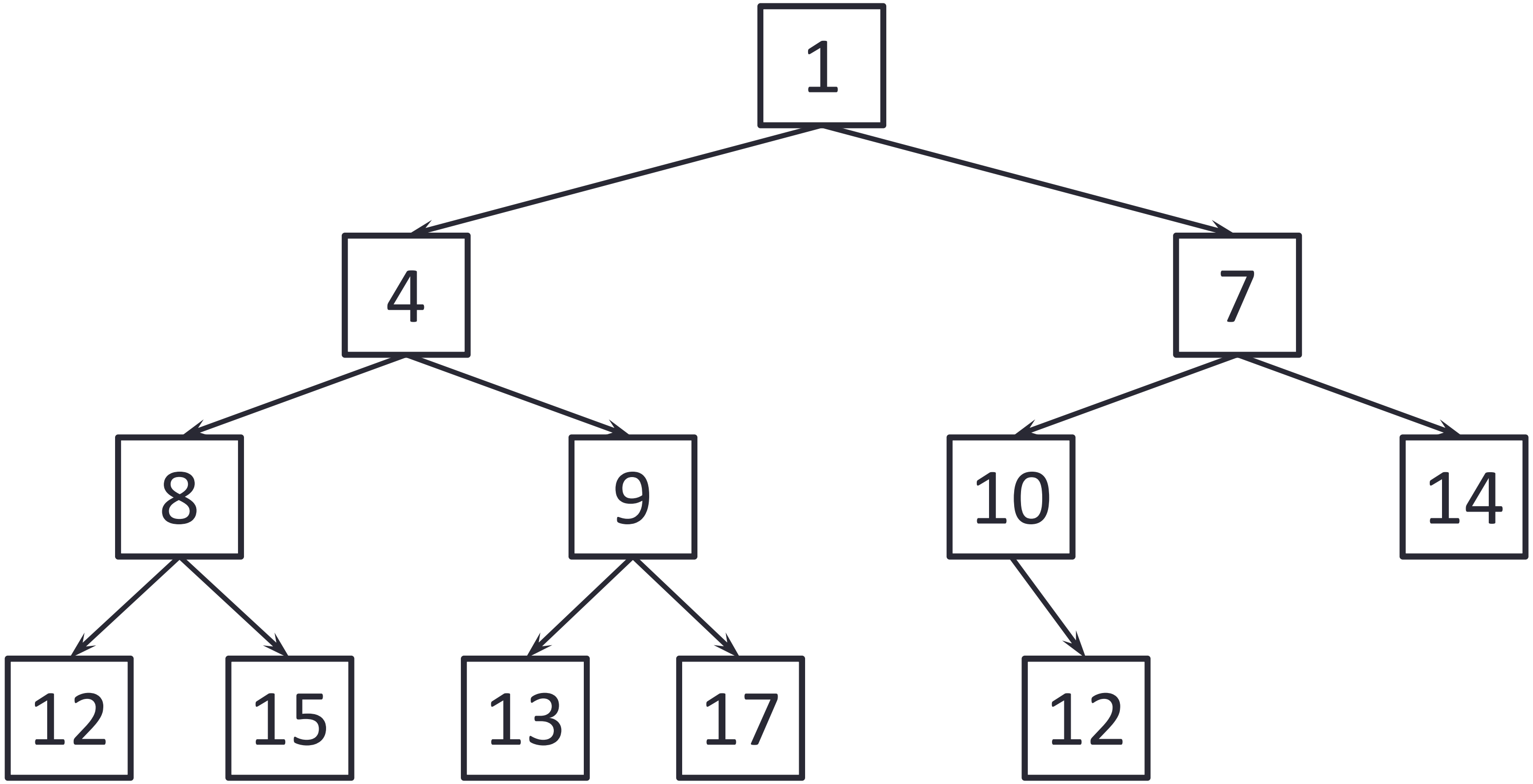
**Heap-Order:** An array h is called heap-ordered if and only if

$$\forall j \in \{2, \dots, n\} : h[\lfloor j/2 \rfloor] \leq h[j]$$

# Binary Heaps: Vector-based Heap Implementation

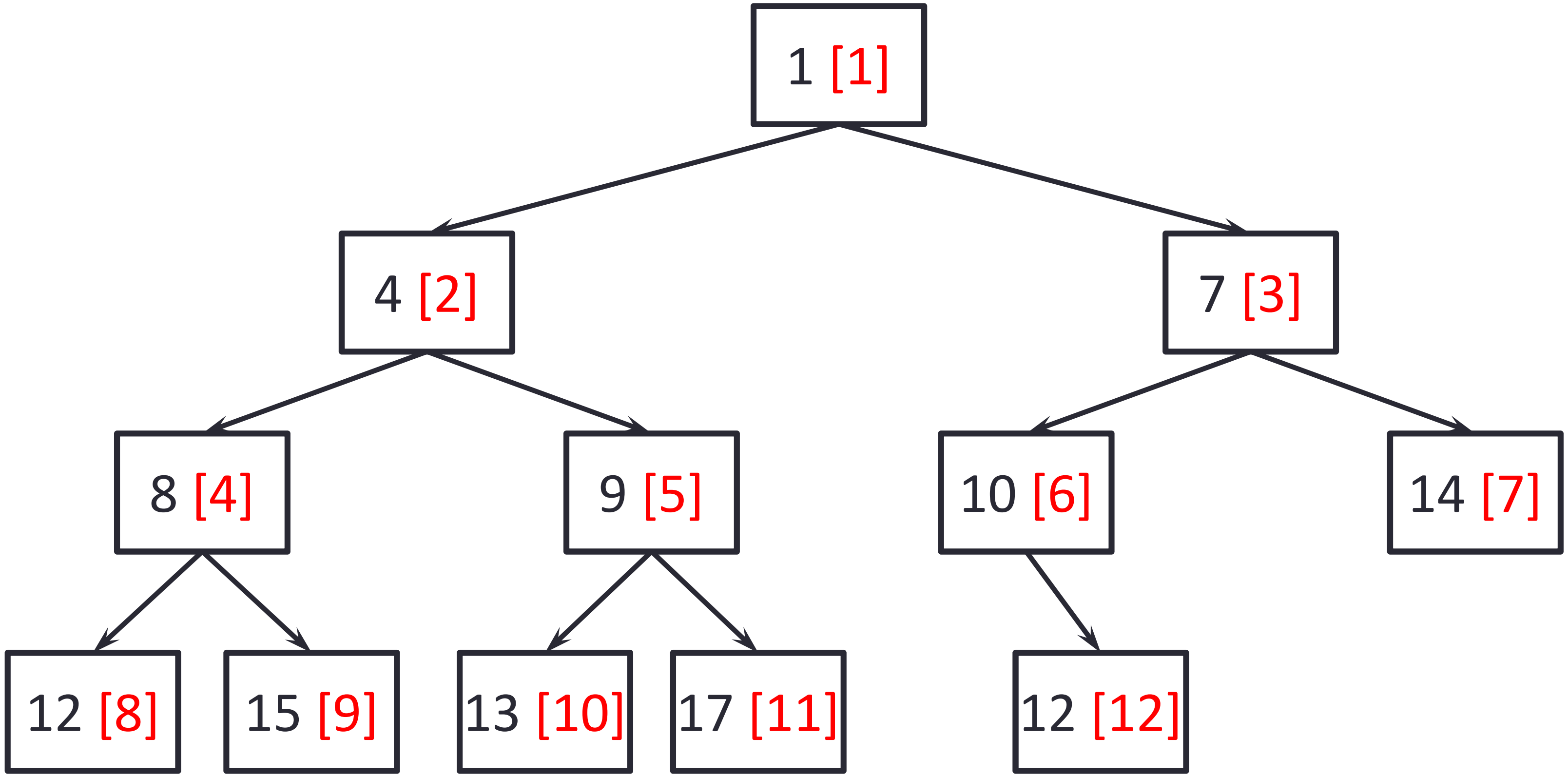
- We can represent a heap with $n$ keys by means of a vector of length $n + 1$
- For the node at rank $i$
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank $n$
- Yields in-place heap-sort

# Binary Heaps as Arrays




| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| value: | 1 | 4 | 7 | 8 | 9 | 10 | 14 | 12 | 15 | 13 | 17 | 12 |
| indices: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Binary Heaps as Arrays




| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| values: | 1 | 4 | 7 | 8 | 9 | 10 | 14 | 12 | 15 | 13 | 17 | 12 |
| indices: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

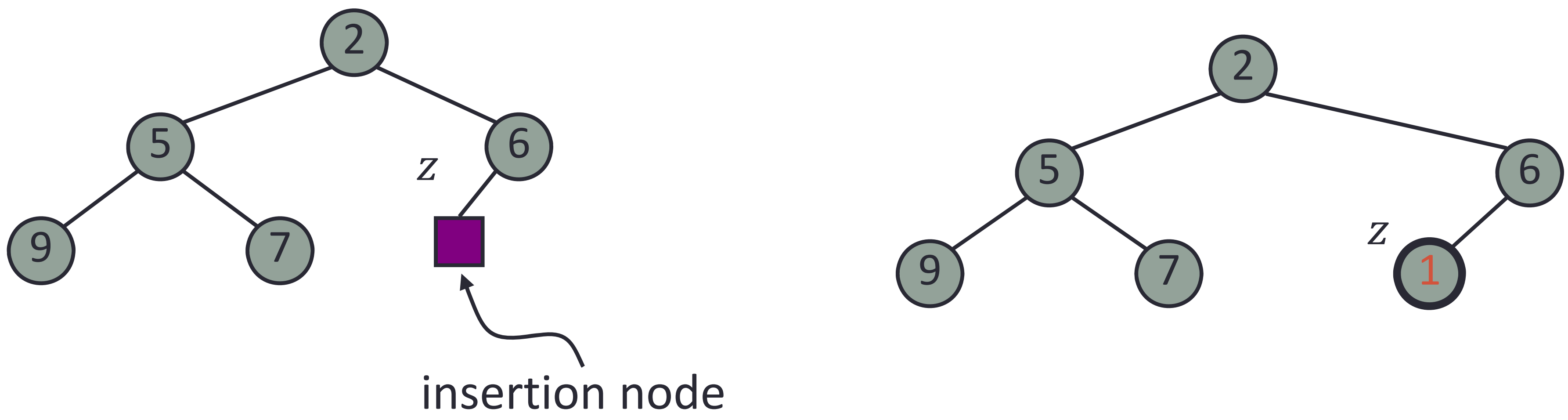
# Binary Heaps: Operations

- M.build $(\{e_1, \ldots, e_n\})$: $M := \{e_1, \ldots, e_n\}$
- M.insert$(e)$: $M := M \cup \{e\}$
- M.min: $\textbf{return} \min M$
- M.deleteMin: $e := \min M$; $M := M \setminus \{e\}$; $\textbf{return}\ e$

# Binary Heaps: Insertion

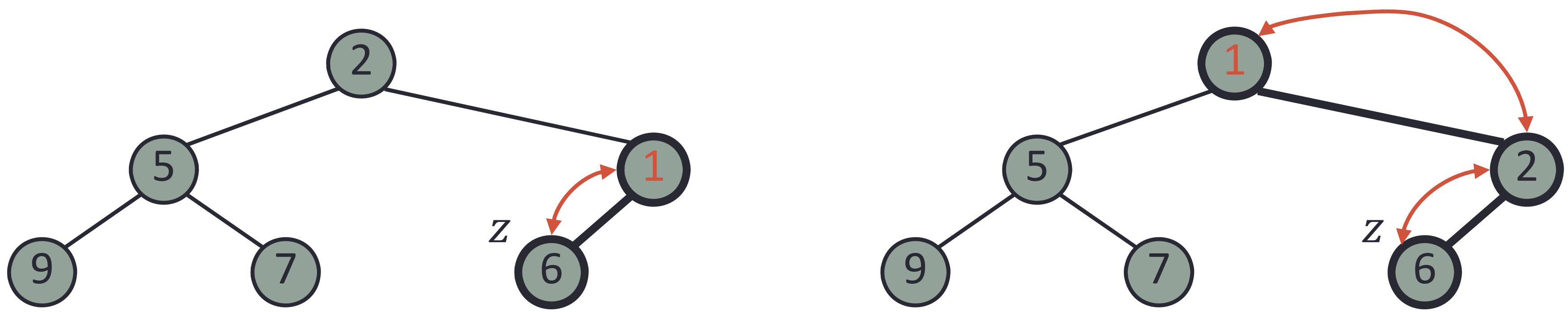
The insertion algorithm to add a key $k$ to the heap consists of three steps:

- Find the insertion node $z$ (the new last node)
- Store $k$ at $z$
- Restore the heap-order property (discussed next)



# Binary Heaps: UpHeap

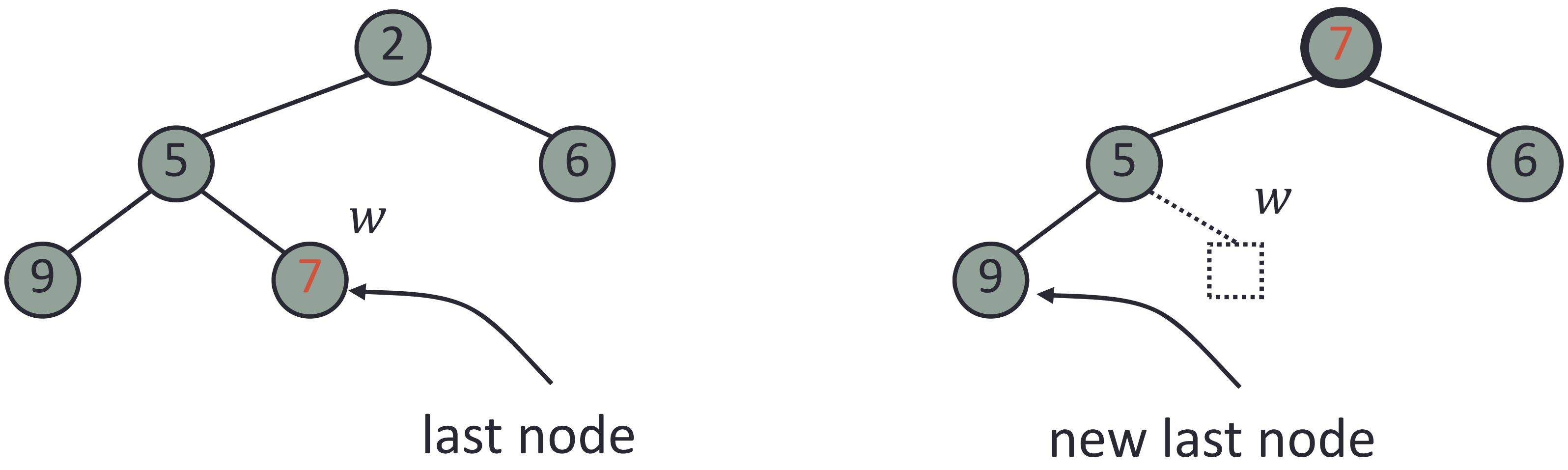
- After the insertion of a new key $k$, the heap-order property may be violated.
- Algorithm UpHeap restores the heap-order property by swapping $k$ along an upward path from the insertion node.
- UpHeap terminates when the key $k$ reaches the root or a node whose parent has a key smaller than or equal to $k$.
- Since a heap has height $O(\log n)$, UpHeap runs in $O(\log n)$ time.



# Binary Heaps: RemoveMin

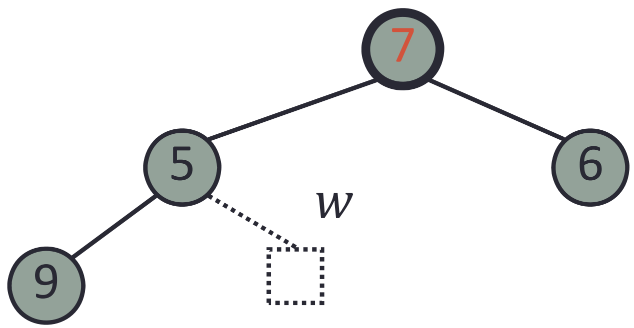
The removal algorithm consists of three steps

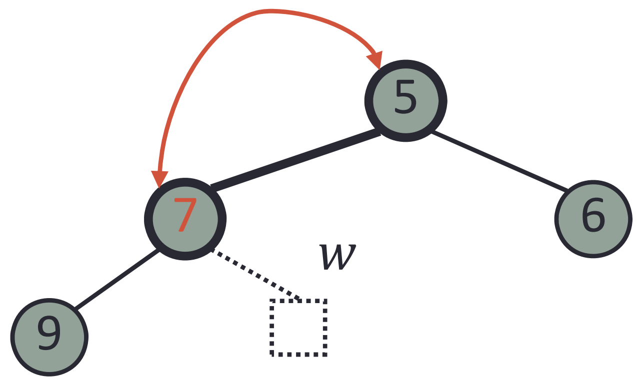
- Replace the root key with the key of the last node $w$
- Remove $w$
- Restore the heap-order property (discussed next)



# Binary Heaps: DownHeap

- After replacing the root key with the key $k$ of the last node, the heap-order property may be violated.
- Algorithm DownHeap restores the heap-order property by swapping key $k$ along a downward path from the root.
- DownHeap terminates when key $k$ reaches a leaf or a node whose children have keys greater than or equal to $k$.
- Since a heap has height $O(\log n)$, DownHeap runs in $O(\log n)$ time



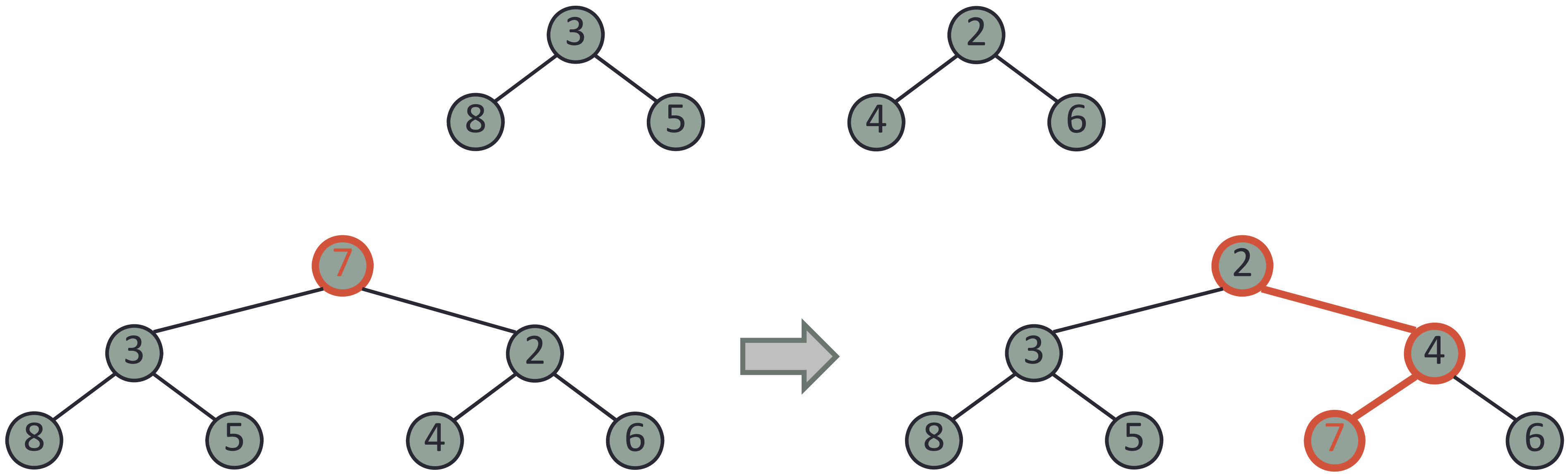
# Binary Heaps: Build Heap

- We can build a binary heap by inserting the $n$ elements one after the other.
- This implies runtime $O(n \log n)$. But can you do better?

- Assume that the heap property holds for all subtrees of height $h$.
- Then we can establish the heap property for height $h + 1$ by DownHeap.

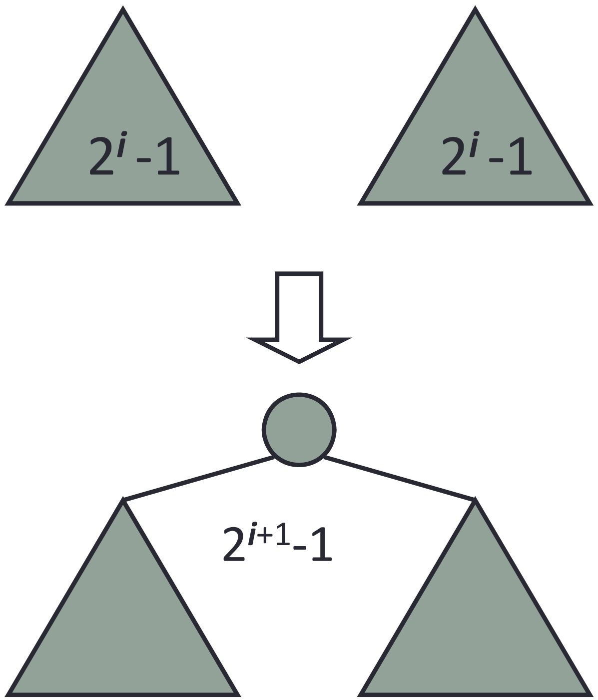
# Binary Heaps: Merging Two Heaps

- We are given two heaps and a key $k$.
- We create a new heap with the root node storing $k$ and with the two heaps as subtrees.
- We perform DownHeap to restore the heap-order property.



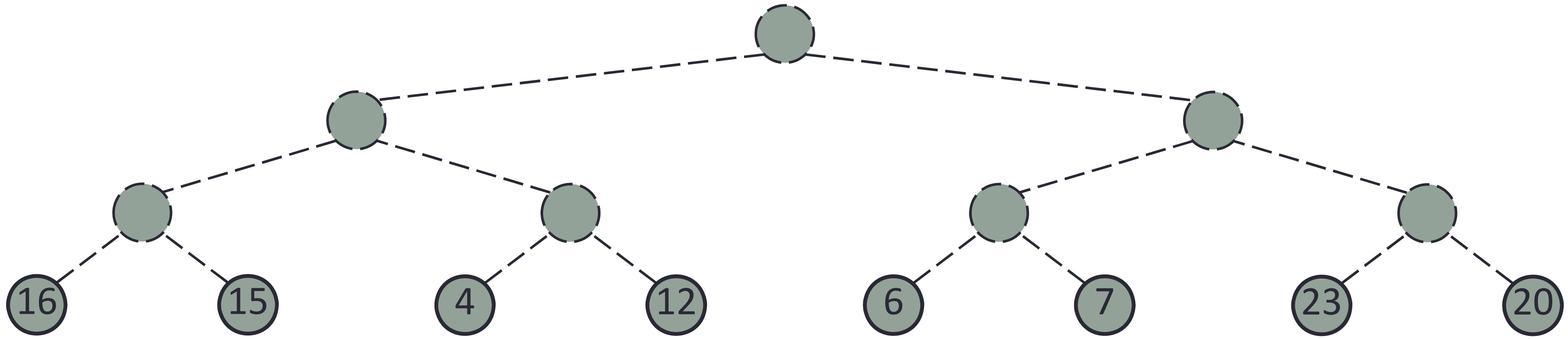
# Binary Heaps: Bottom-up Heap Construction

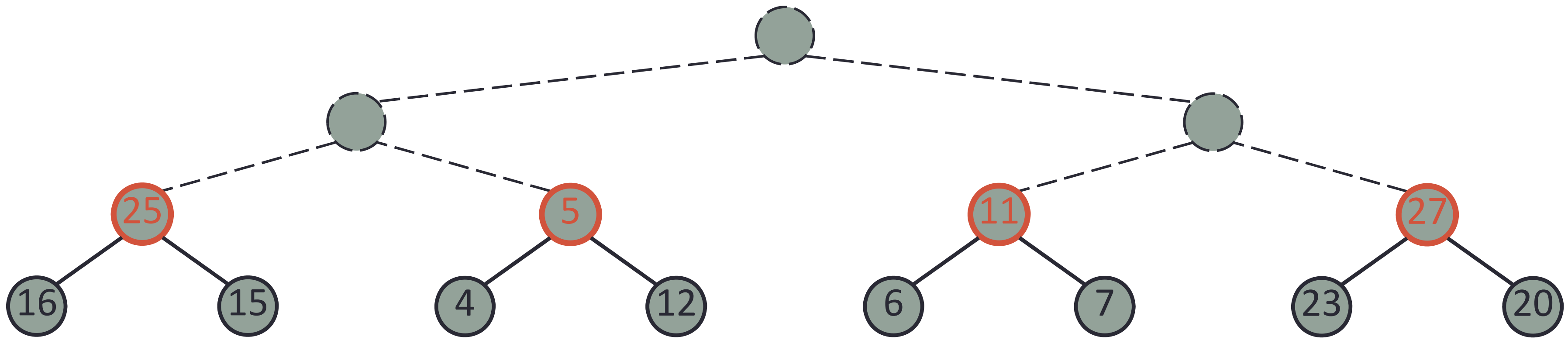
- We can construct a heap storing $n$ given keys in using a bottom-up construction with $\log n$ phases
- In phase $i$, pairs of heaps with $2^i$-1 keys are merged into heaps with $2^{i+1}$-1 keys



# Binary Heaps: Bottom-up Heap Construction

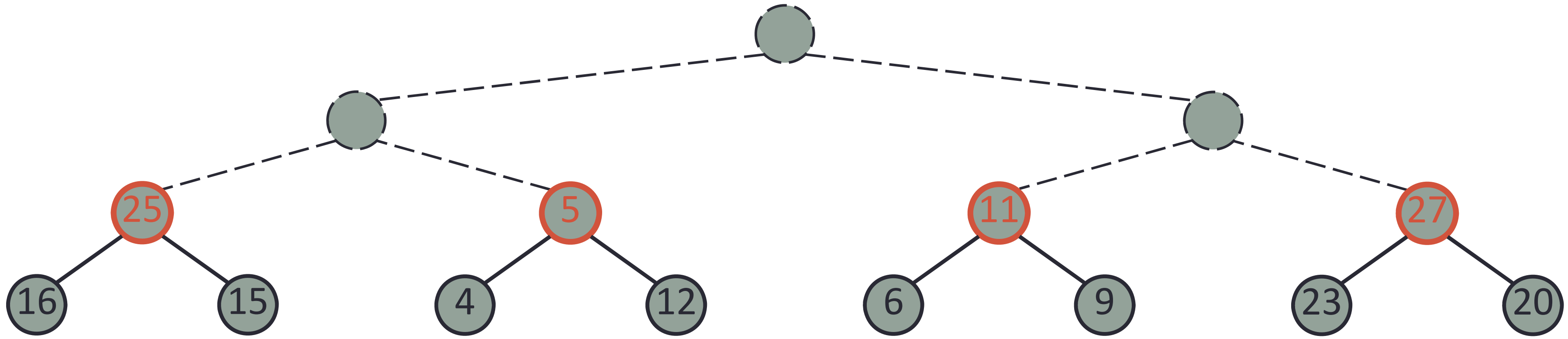
## Example

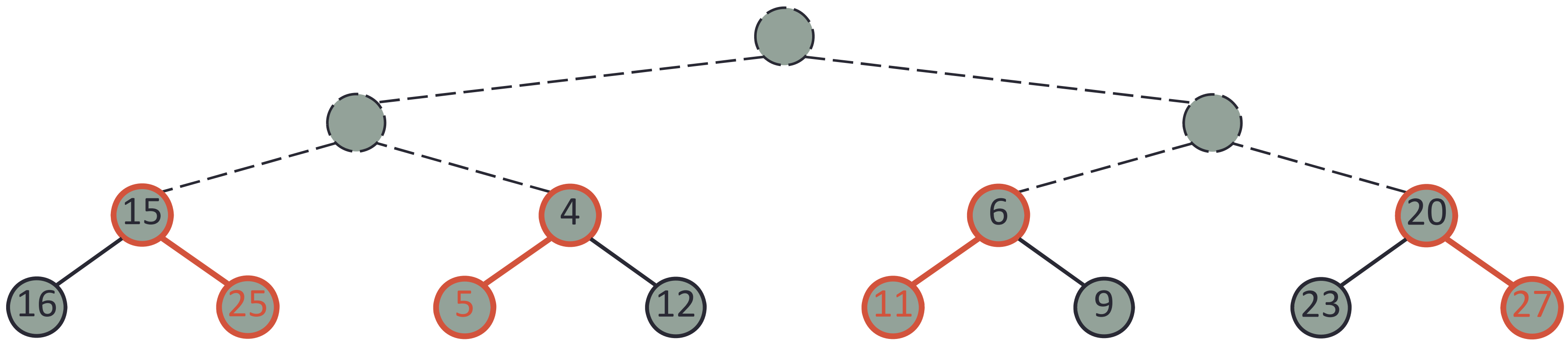


# Binary Heaps: Bottom-up Heap Construction

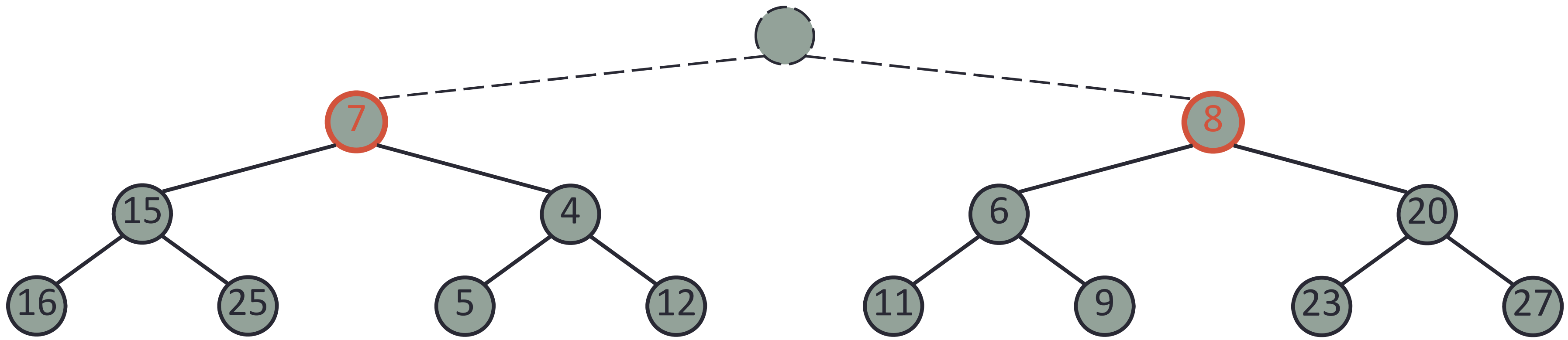
## Example (cont.)

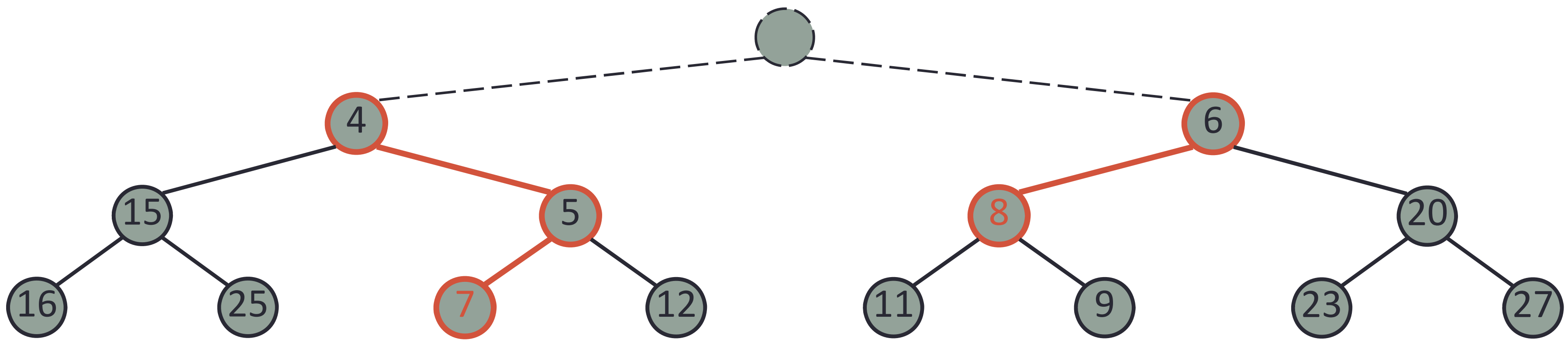


# Binary Heaps: Bottom-up Heap Construction

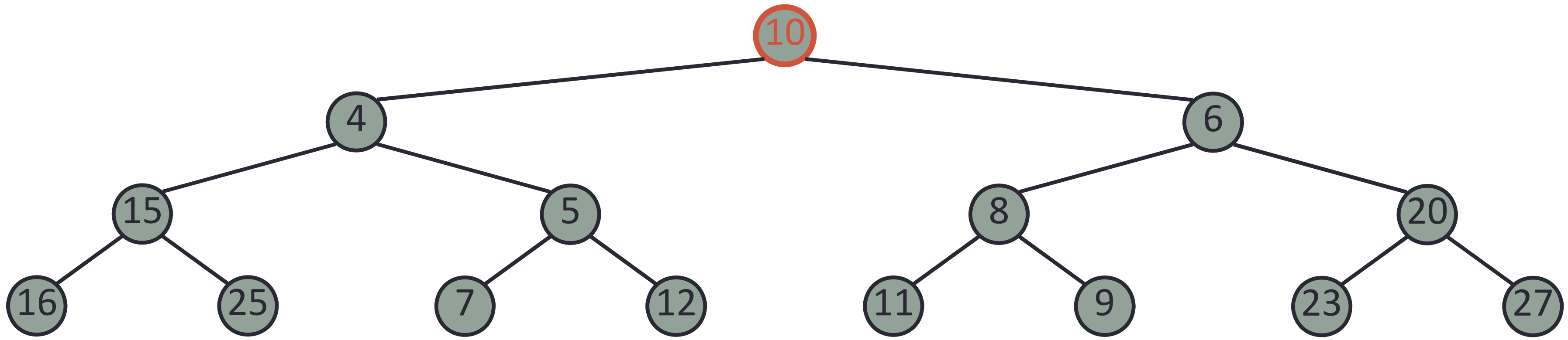
## Example (cont.)

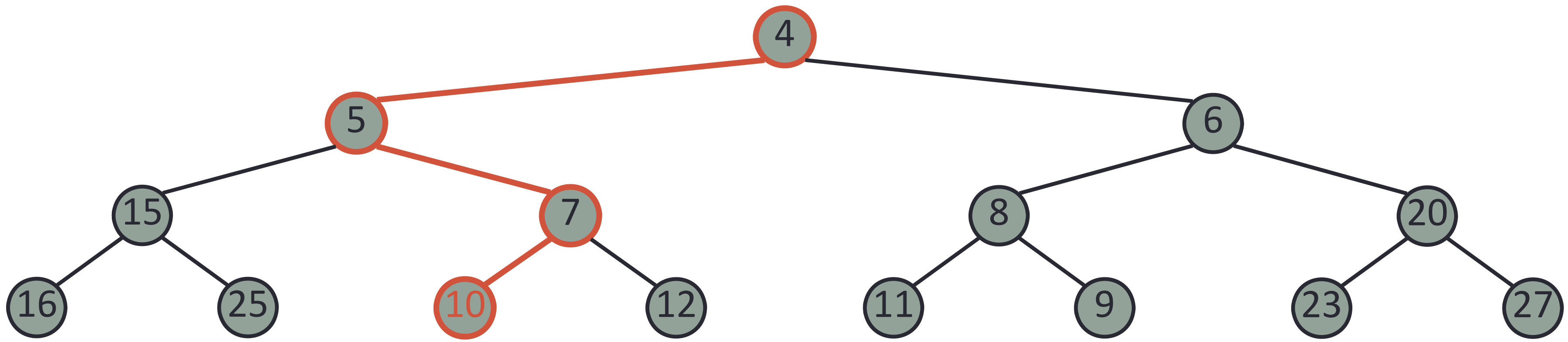


# Binary Heaps: Bottom-up Heap Construction

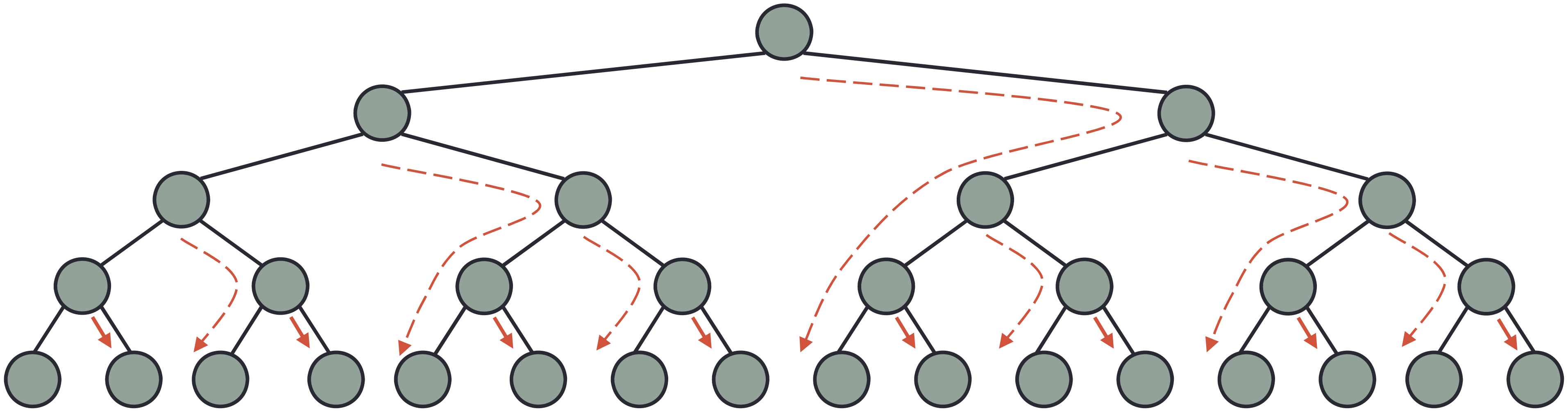
## Example (cont.)



# Binary Heaps: Analysis

- We visualize the worst-case time of a DownHeap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual DownHeap path).
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$.
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than $n$ successive insertions and speeds up the first phase of the Heap Sort sorting algorithm.



# Other references and things to do

- Read relevant sections of chapter 8 and 9.3 in Data Structures and Algorithms in Java. Michael T. Goodrich, Irvine Roberto Tamassia, and Michael H. Goldwasser, 2014.