CHAPTER 09

BST, Priority Queue, Heaps - Heapsort

Compiled by: Dr. Mohammad Alhawarat

Chapter Contents

Part I: Binary Search Tree

Part II: Priority Queue & Heaps

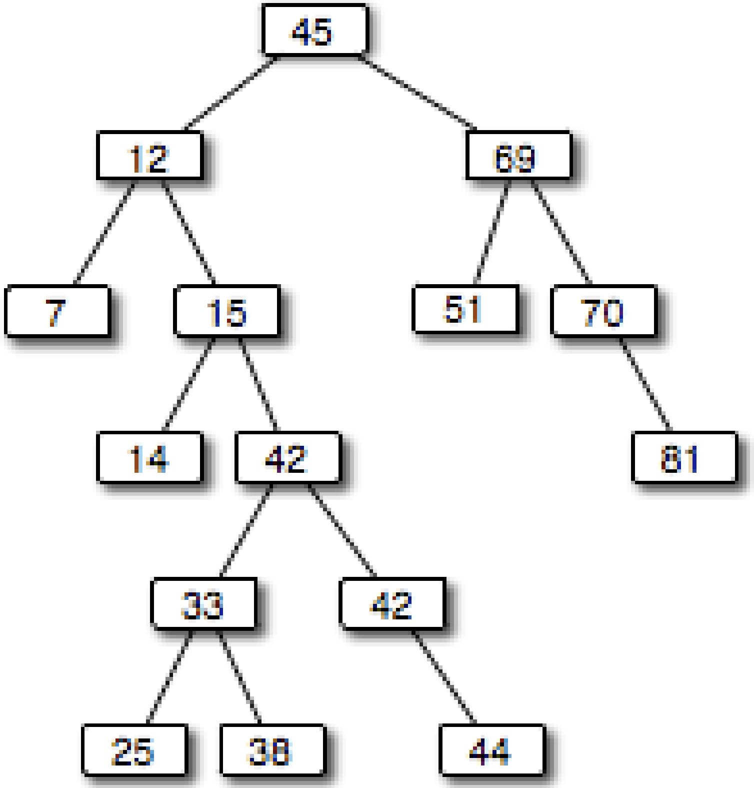
Part III: Heapsort

Part I: Binary Search Tree

Binary Search Trees (BSTs)

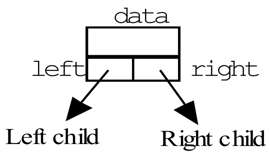
- A *search tree* is a tree whose elements are organized to facilitate finding a particular element when needed
- A *binary search tree* is a binary tree that, for each node *n*
  - the left **subtree** of *n* contains elements less than the element stored in *n*
  - the right **subtree** of *n* contains elements greater than or equal to the element stored in *n*

Binary Search Trees (BSTs)

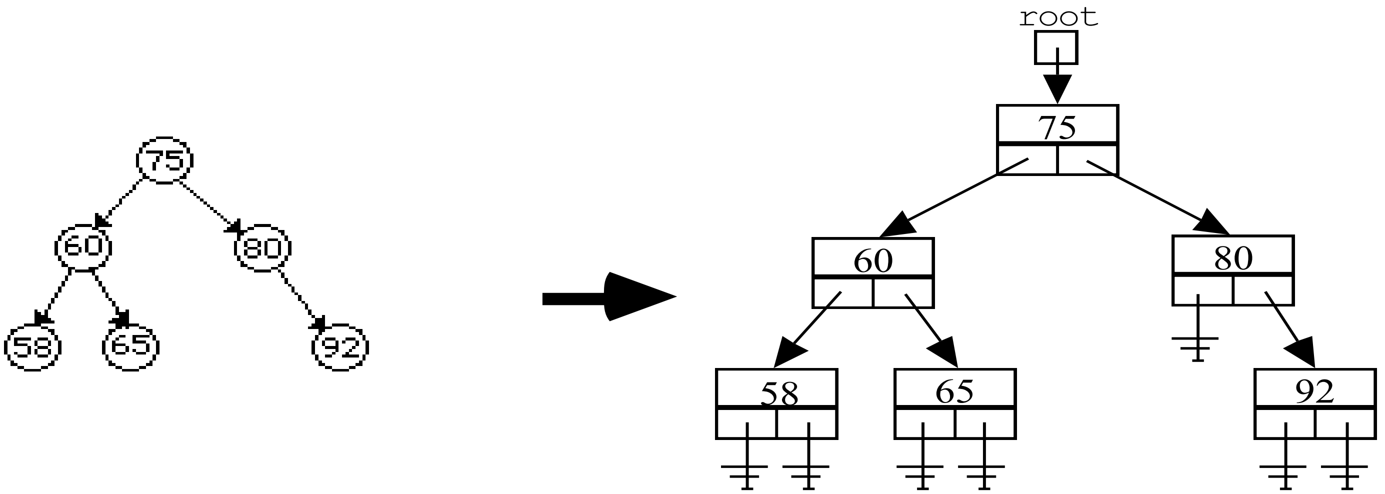


Implementation of BST

Linked Implementation: Use nodes of the form



and maintain a pointer to the root.



Traversing BST


```
void inorder(Node r)          //yields ordered sequence
{ if (r != null)
    {inorder(r.left);  // Left
     visit(r.data);  // Root
     inorder(r.right); // Right
   }
)
void preorder(Node r)
{ if (r != null)
    { visit(r.data);   // Root
    preorder(r.left);  // Left
       preorder(r.right); // Right
   }
)
void postorder(Node r)
{ if (r != null)
    {postorder(r.left);  // Left
     postorder(r.right);  // Right
     visit (r.data);  // Root
   }
)
```

Binary Search Trees (BSTs)

- To determine if a particular value exists in a tree
  - start at the root
  - compare target to element at current node
  - move left from current node if target is less than element in the current node
  - move right from current node if target is greater than element in the current node
- We eventually find the target or encounter the end of a path (target is not found)

Searching in BST

1. Set pointer locPtr = root.
2. Repeat the following:

```
If locPtr is null
        Return False
If Value <  locPtr.Data
        locPtr = locPtr.Left
Else if Value > locPtr.Data
        locPtr = locPtr.Right
Else
        Return True
```

Search time: $O(\log_2 n)$ if tree is balanced.

Binary Search Trees (BSTs)

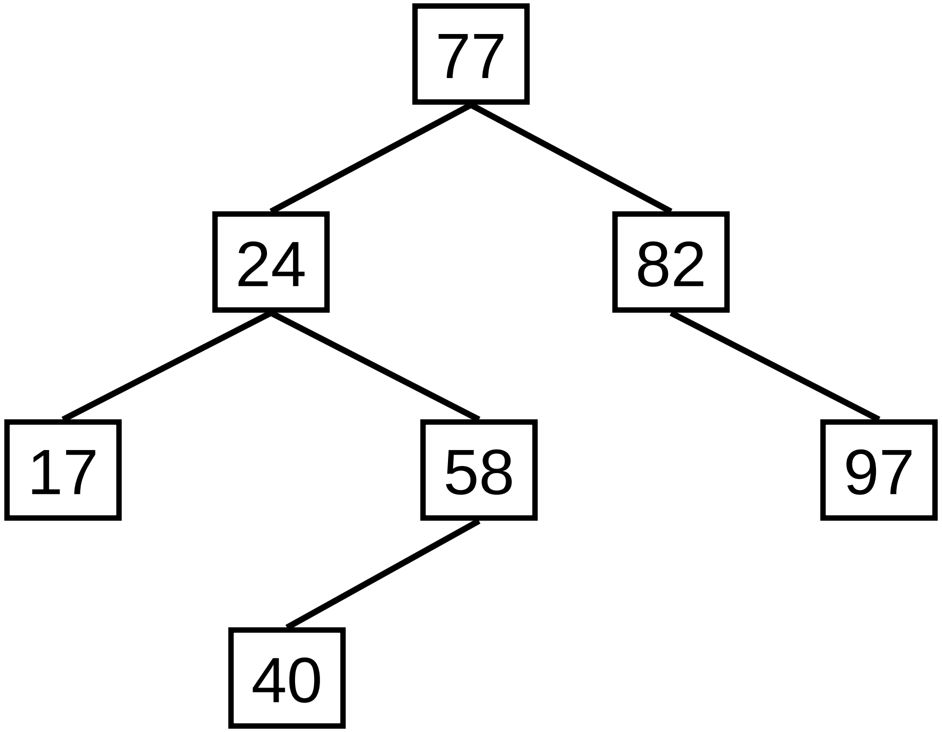
- The particular shape of a binary search tree depends on the order in which the elements are added to the tree
- The shape may also be dependant on any additional processing performed on the tree to reshape it
- Binary search trees can hold any type of data, so long as we have a way to determine relative ordering

Adding an Element to a BST

- Process of adding an element is similar to finding an element
- New elements are added as leaf nodes
- Start at the root, follow path dictated by existing elements until you find no child in the desired direction
- Add the new element

Adding an Element to a BST

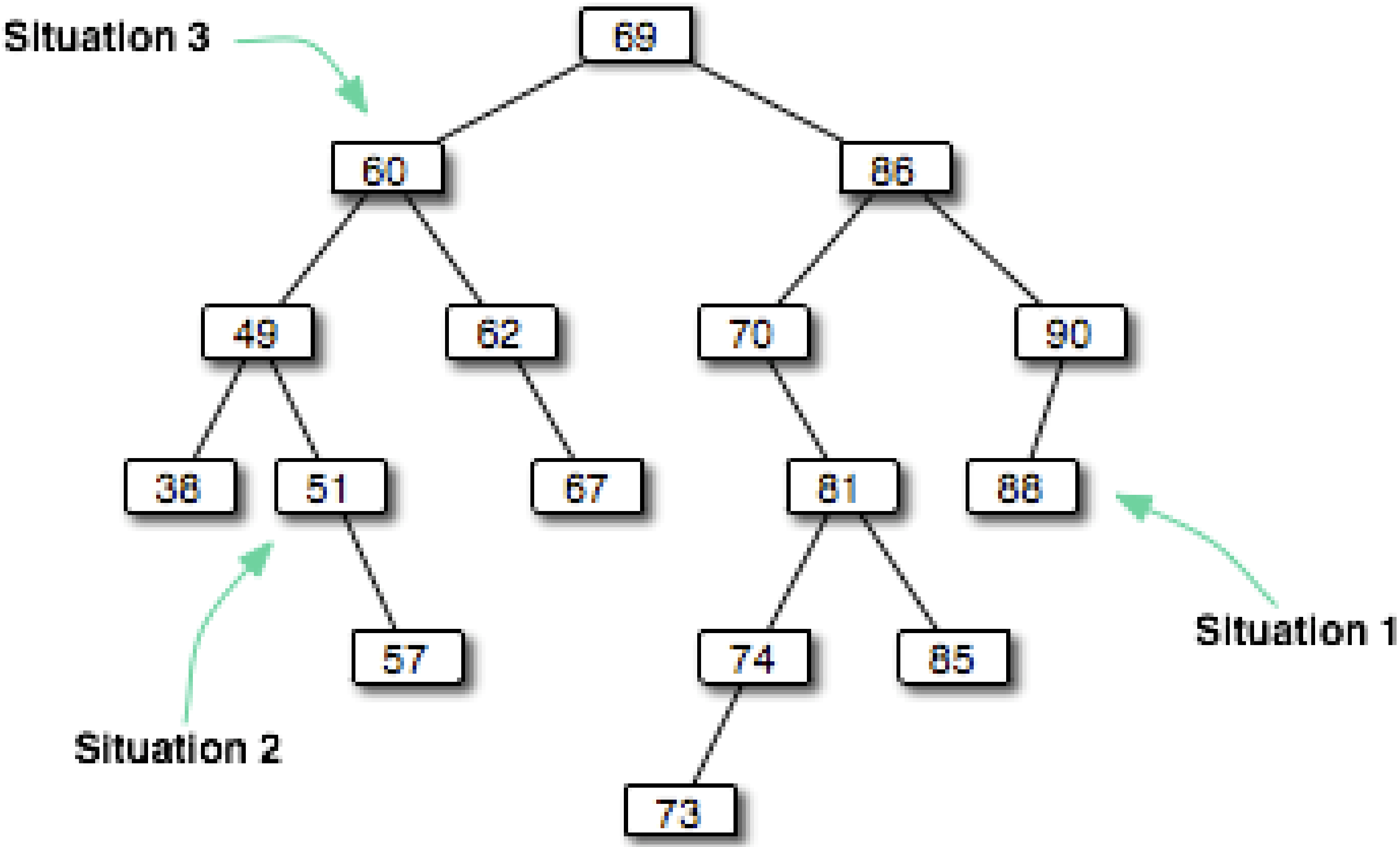
Next to add: 77 24 82 97 58 17 40



Removing an Element from a BST

- Removing a target in a BST is not as simple as that for linear data structures
- After removing the element, the resulting tree must still be valid
- Three distinct situations must be considered when removing an element
  - The node to remove is a leaf
  - The node to remove has one child
  - The node to remove has two children

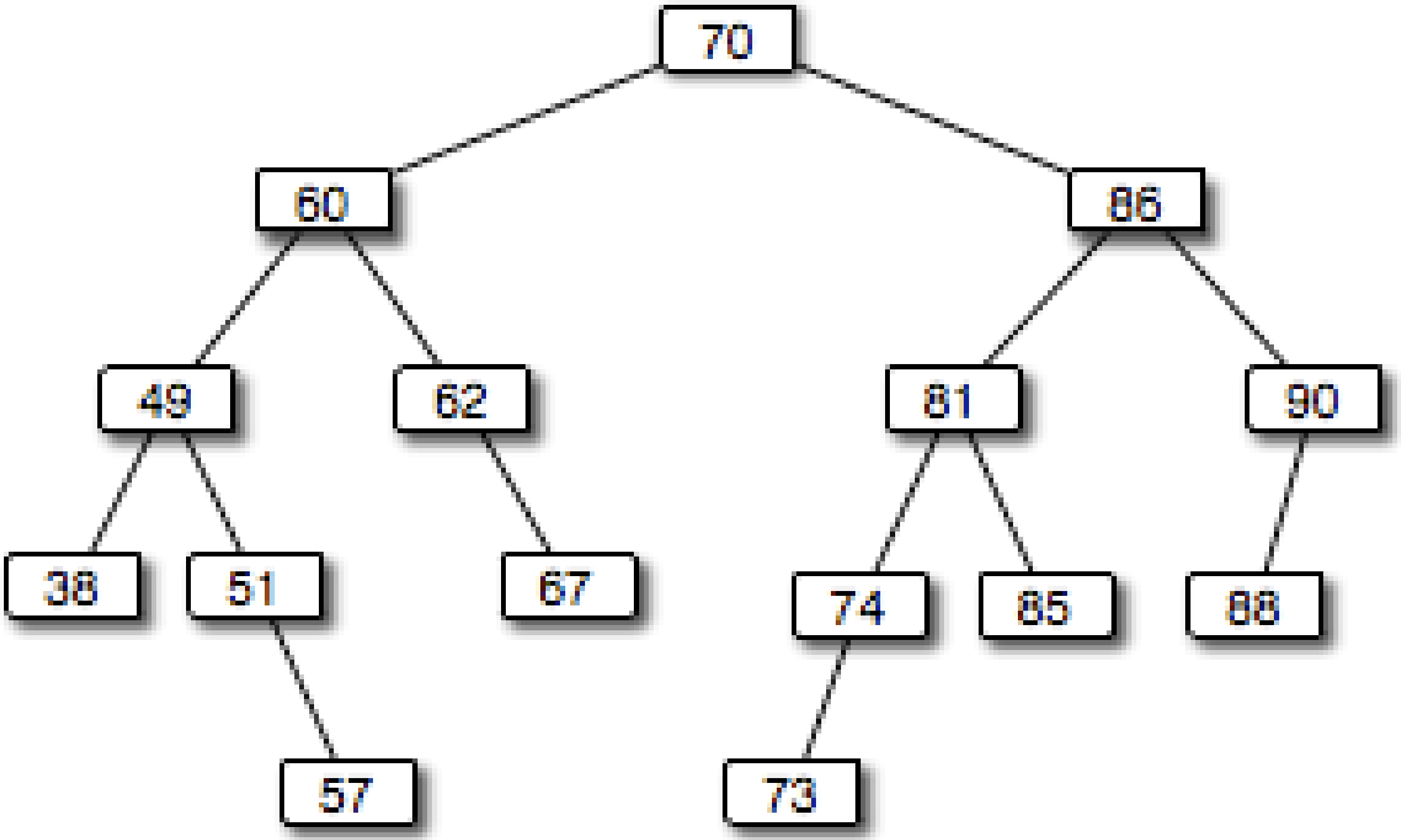
Removing an Element from a BST



Removing an Element from a BST

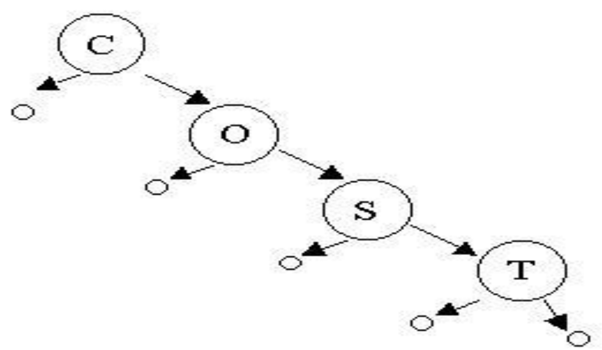
- Dealing with the situations
  - Node is a leaf: it can simply be deleted
  - Node has one child: the deleted node is replaced by the child
  - Node has two children: an appropriate node is found lower in the tree and used to replace the node:
    - Either selecting the largest element in the left subtree.
    - Or selecting the smallest element in the right subtree.

After the Root Node is Removed



Complexity

- Logarithmic, depends on the shape of the tree $O(\log_2 N)$
- In the worst case – $O(N)$ comparisons



Advantages of BST

- Simple
- Efficient
- Dynamic

- One of the most fundamental algorithms in CS
- The method of choice in many applications

Disadvantages of BST

- The shape of the tree depends on the order of insertions, and it can be degenerated. (Becomes a Linked List)

- When inserting or searching for an element, the key of each visited node has to be compared with the key of the element to be inserted/found.

Improvements of BST

Keeping the tree balanced:

- AVL trees (Adelson - Velskii and Landis)
- Balance condition: left and right subtrees of each node can differ by at most one level.
- It can be proved that if this condition is observed the depth of the tree is $O(\log_2 N)$.

Part II: Priority Queue & Heaps

Priority Queue ADT

- The data in a ***priority queue*** is (conceptually) a queue of elements

- The "queue" can be thought of as sorted with the largest in front, and the smallest at the end
  - Its physical form, however, may differ from this conceptual view considerably

Priority Queue ADT Operations

- ***enqueue***, an operation to add an element to the queue
- ***dequeue***, an operation to take the largest element from the queue
- an operation to determine whether or not the queue is empty
- an operation to empty out the queue

Priority Queue Implementation

- Priority Queue could be implemented in different ways.
- One way is to use vectors.
- Another way is to use Binary Heap.
- What’s a Heap?

Heaps

- A heap is a complete binary tree in which the value of each node is greater than or equal to the values of its children (if any)
- Technically, this is called a maxheap
- In a minheap, the value of each node is less than or equal to the values of its children

Heaps (cont.)

- The element stored in each node of a heap can be an object
- When we talk about the value of a node, we are really talking about some data member of the object that we want to prioritize
- For example, in employee objects, we may want to prioritize age or salary

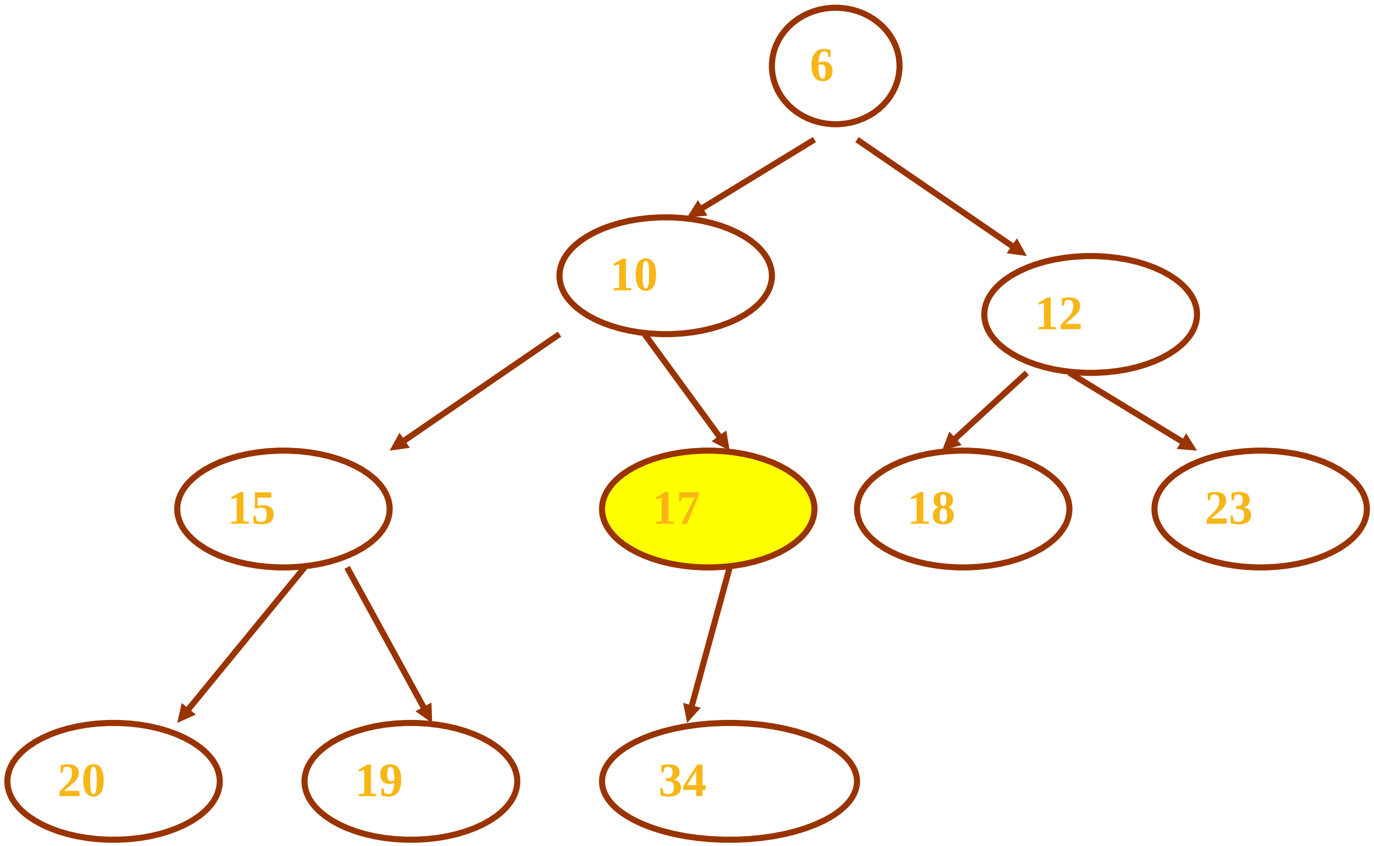
Implementations

- Binary heap
- Better than BST because it does not support links.
- Insert: $O(\log_2 N)$
- Find minimum $O(\log_2 N)$
- Deleting the minimal element takes a constant time, however after that the heap structure has to be adjusted, and this requires $O(\log_2 N)$ time.

Binary Heap

- Heap-Structure Property:
- Complete Binary Tree - Each node has two children, except for the last two levels.
- The nodes at the last level do not have children. New nodes are inserted at the last level from left to right.
- Heap-Order Property:
- Each node has a higher priority than its children

Binary Heap



Next node to be inserted - right child of the yellow node

Basic Operations

- Build the heap
- Insert a node – Percolate Up
- Delete a node – Percolate Down

Build Heap - O(N)

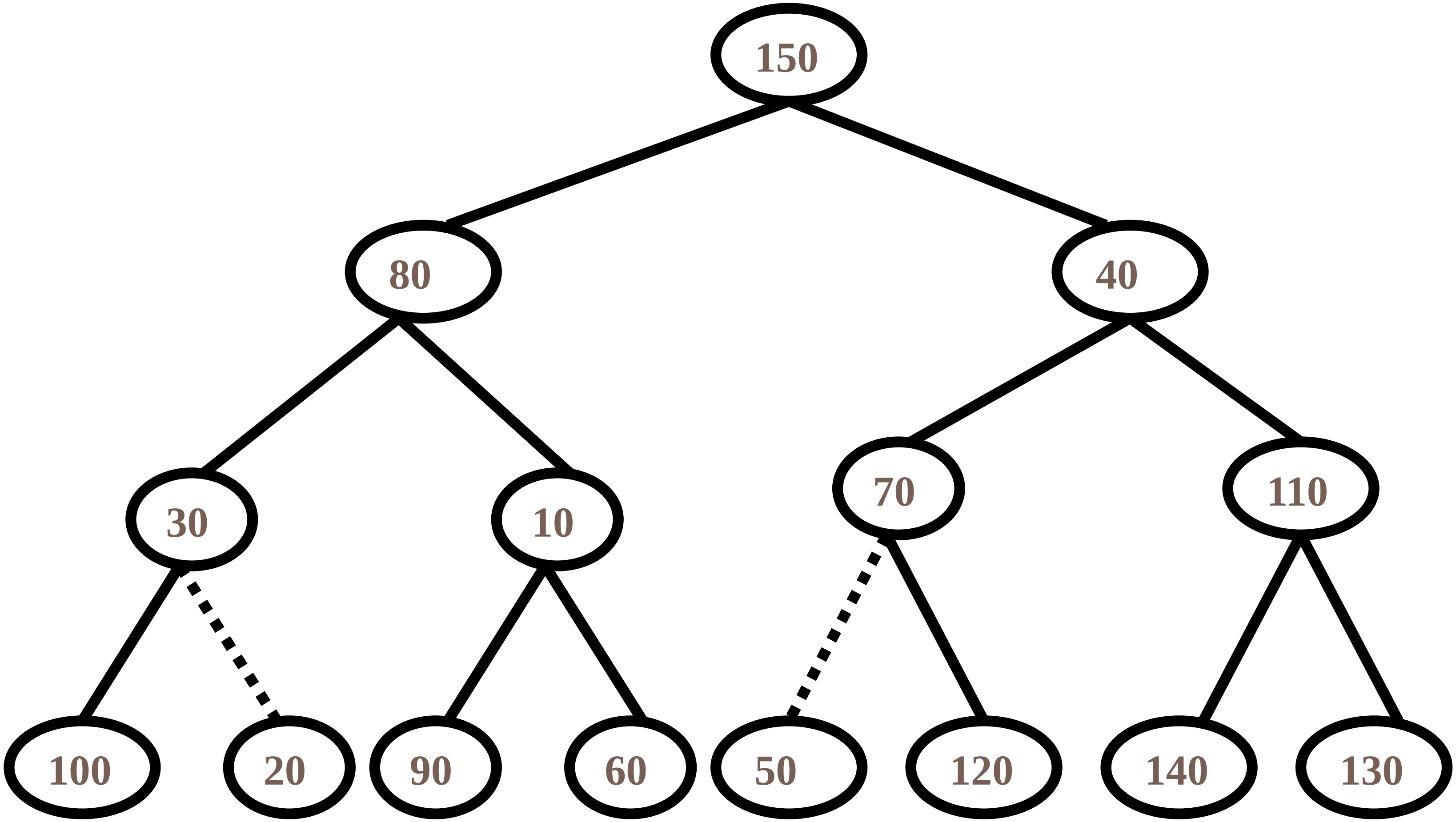
- Given an array of elements to be inserted in the heap,

- treat the array as a heap with order property violated,

- and then do operations to fix the order property.

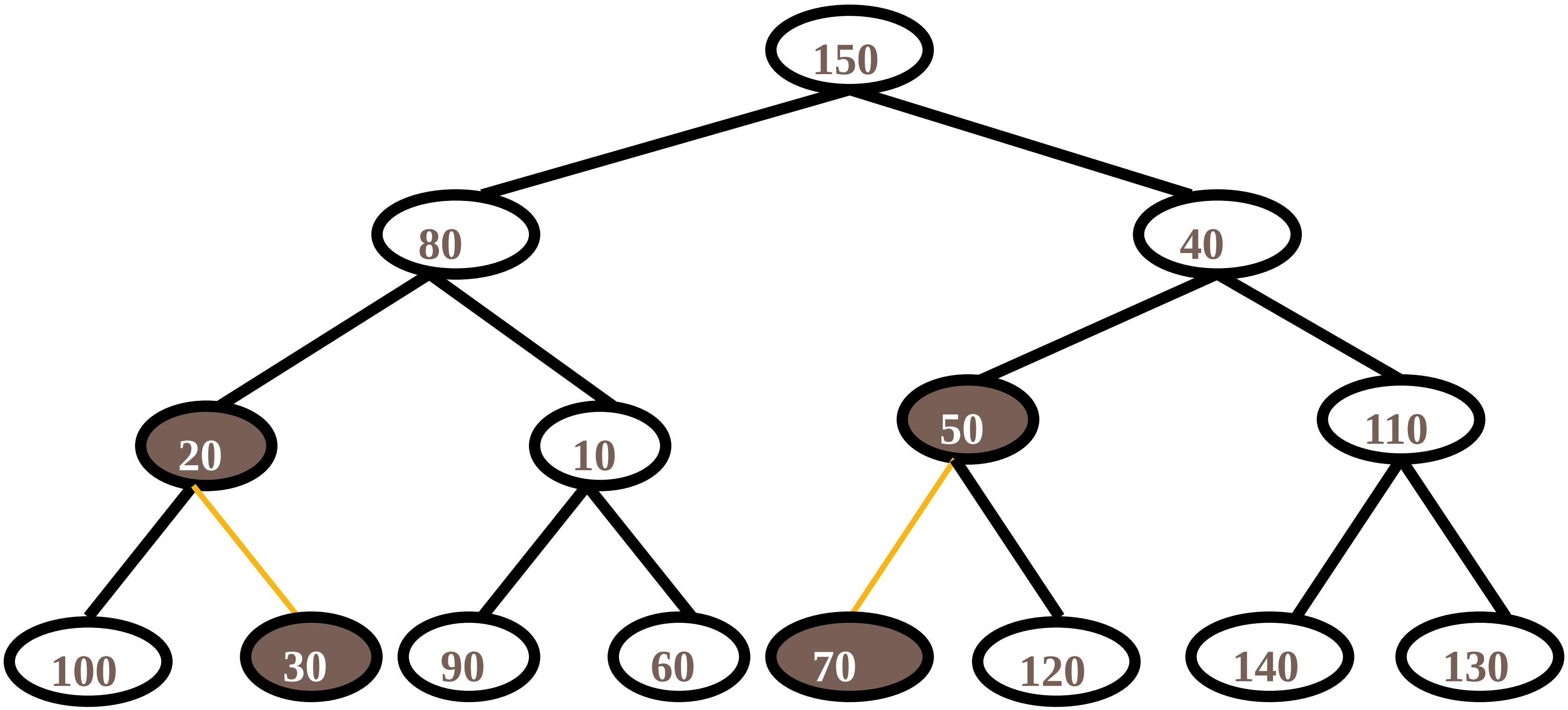
Example

150 80 40 30 10 70 110 100 20 90 60 50 120 140 130



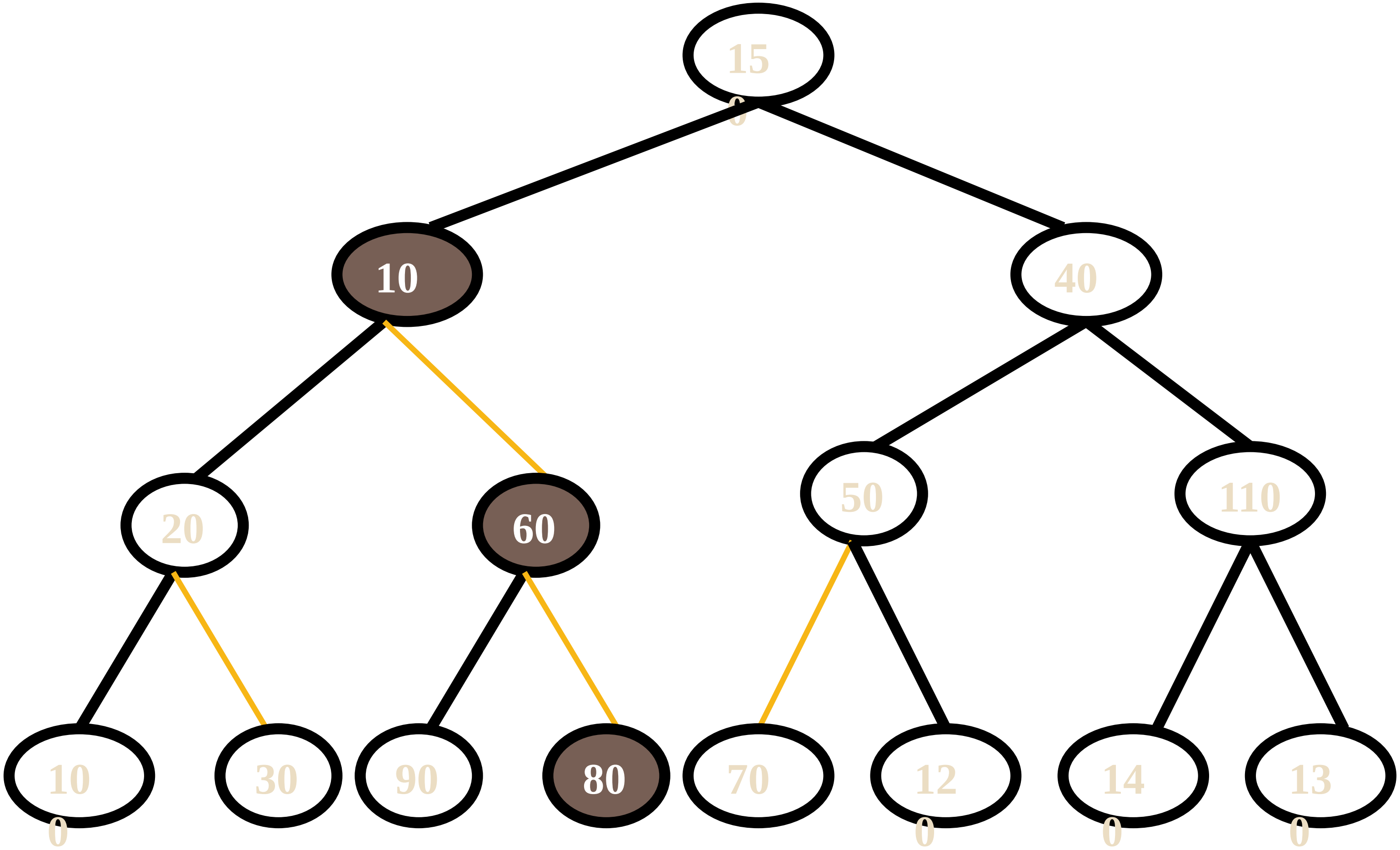
Example (cont)

After processing height 1



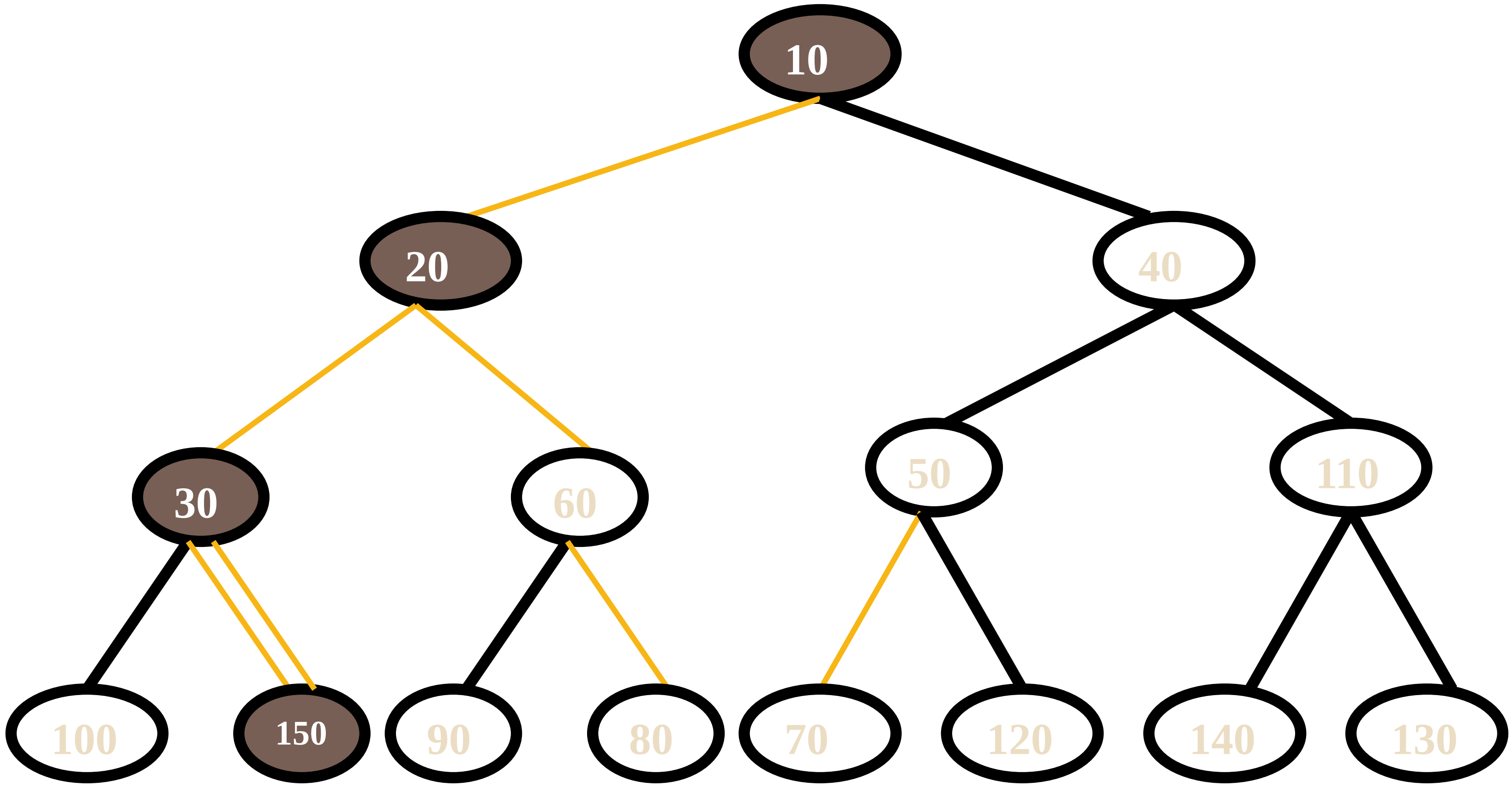
Example (cont)

After processing height 2



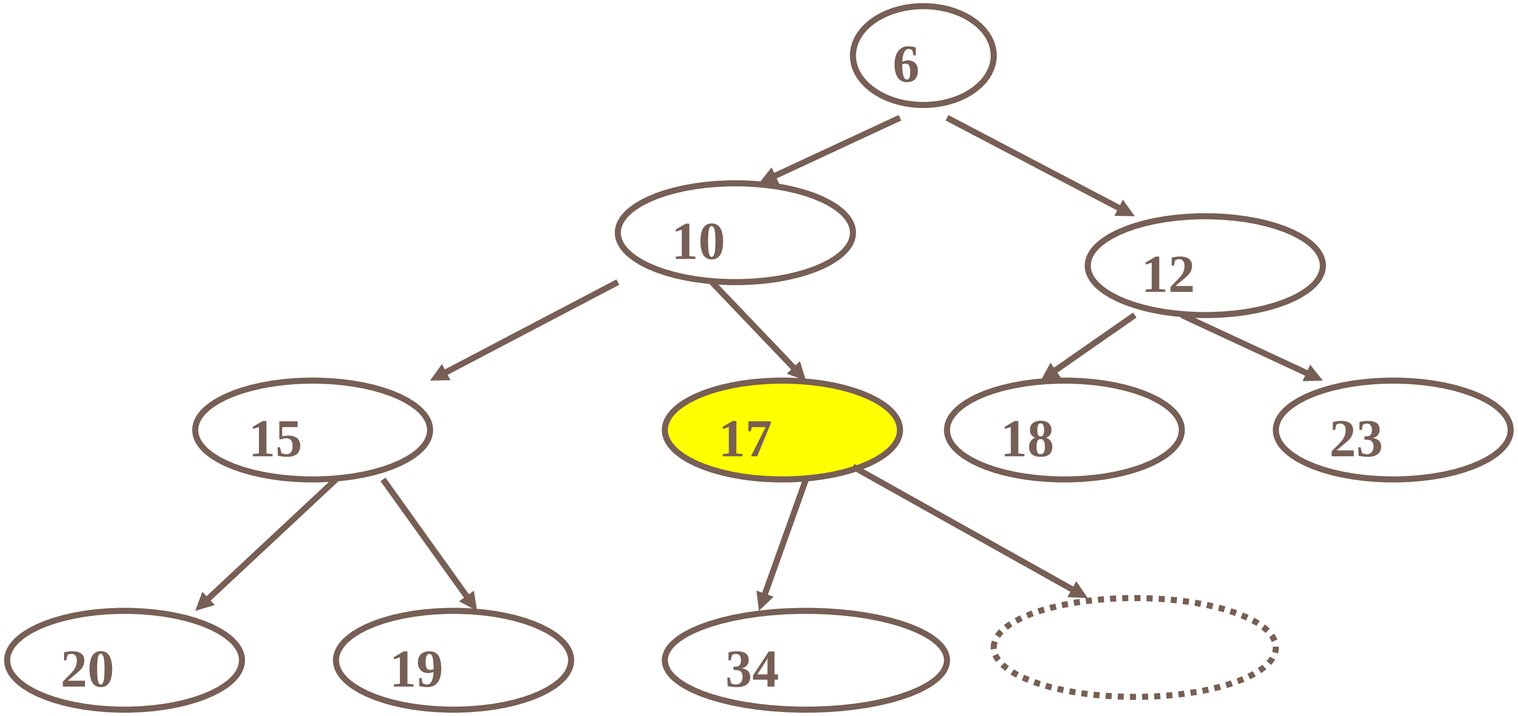
Example (cont)

After processing height 3



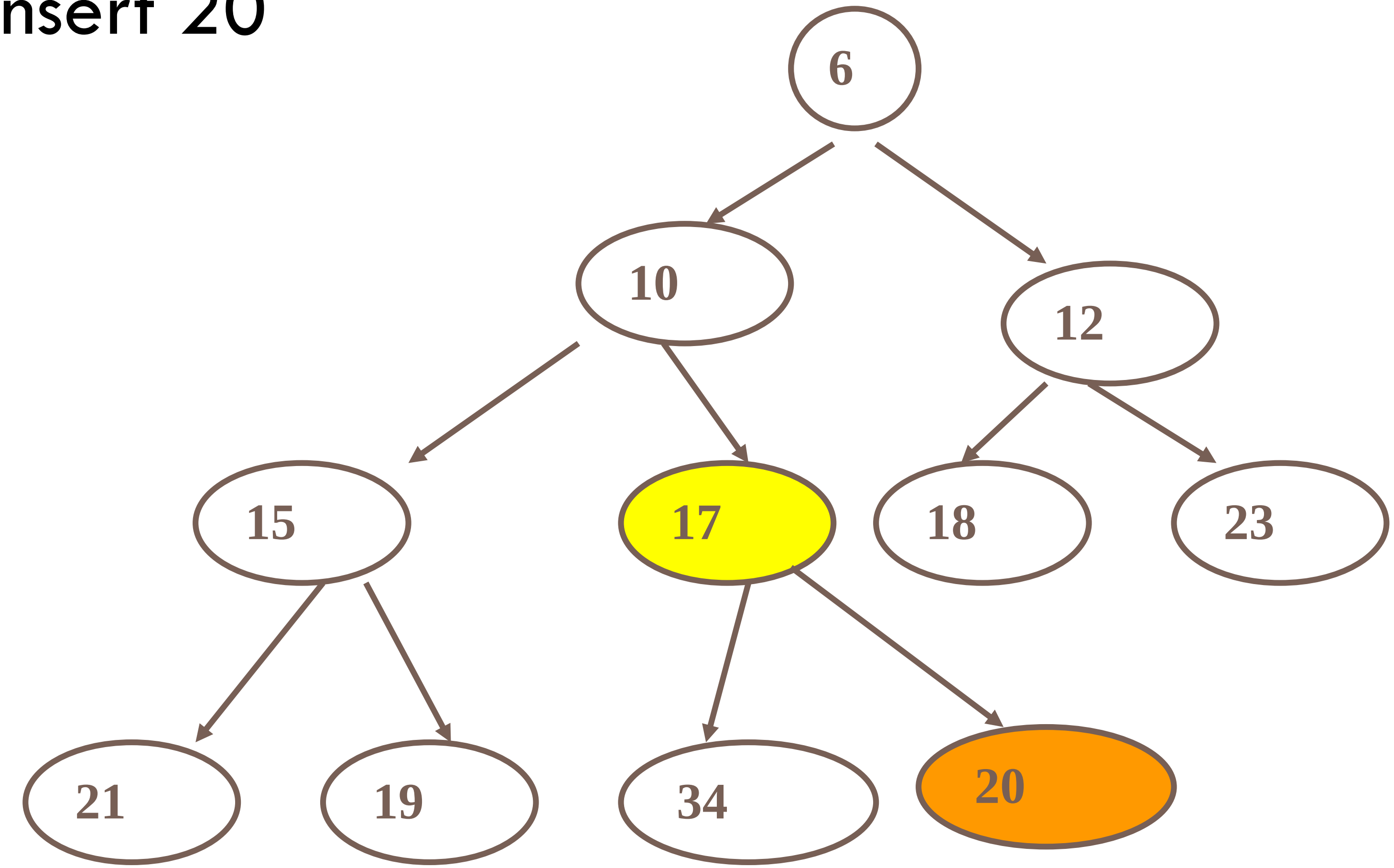
Percolate Up – Insert a Node

- A hole is created at the bottom of the tree, in the next available position.



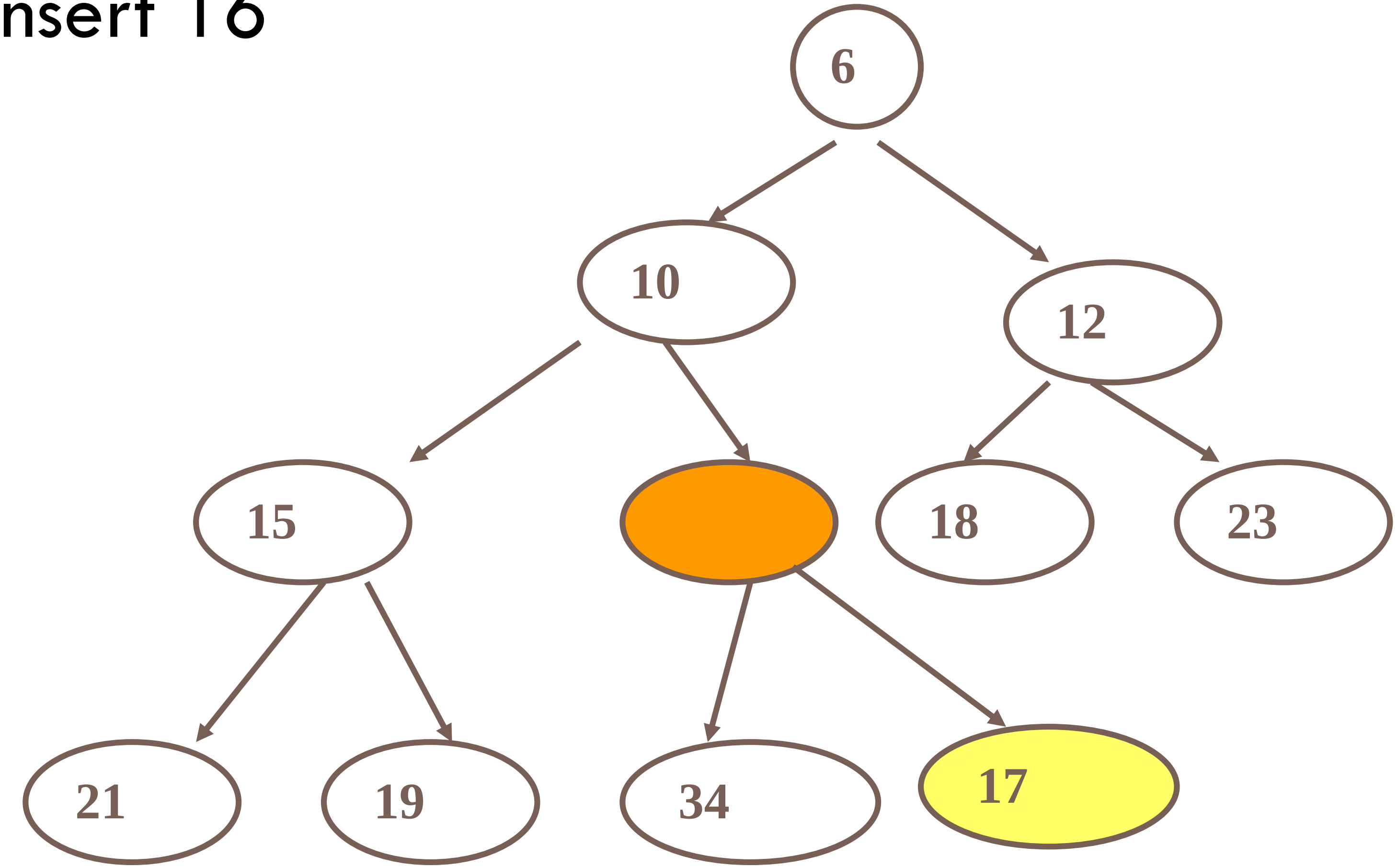
Percolate Up

Insert 20

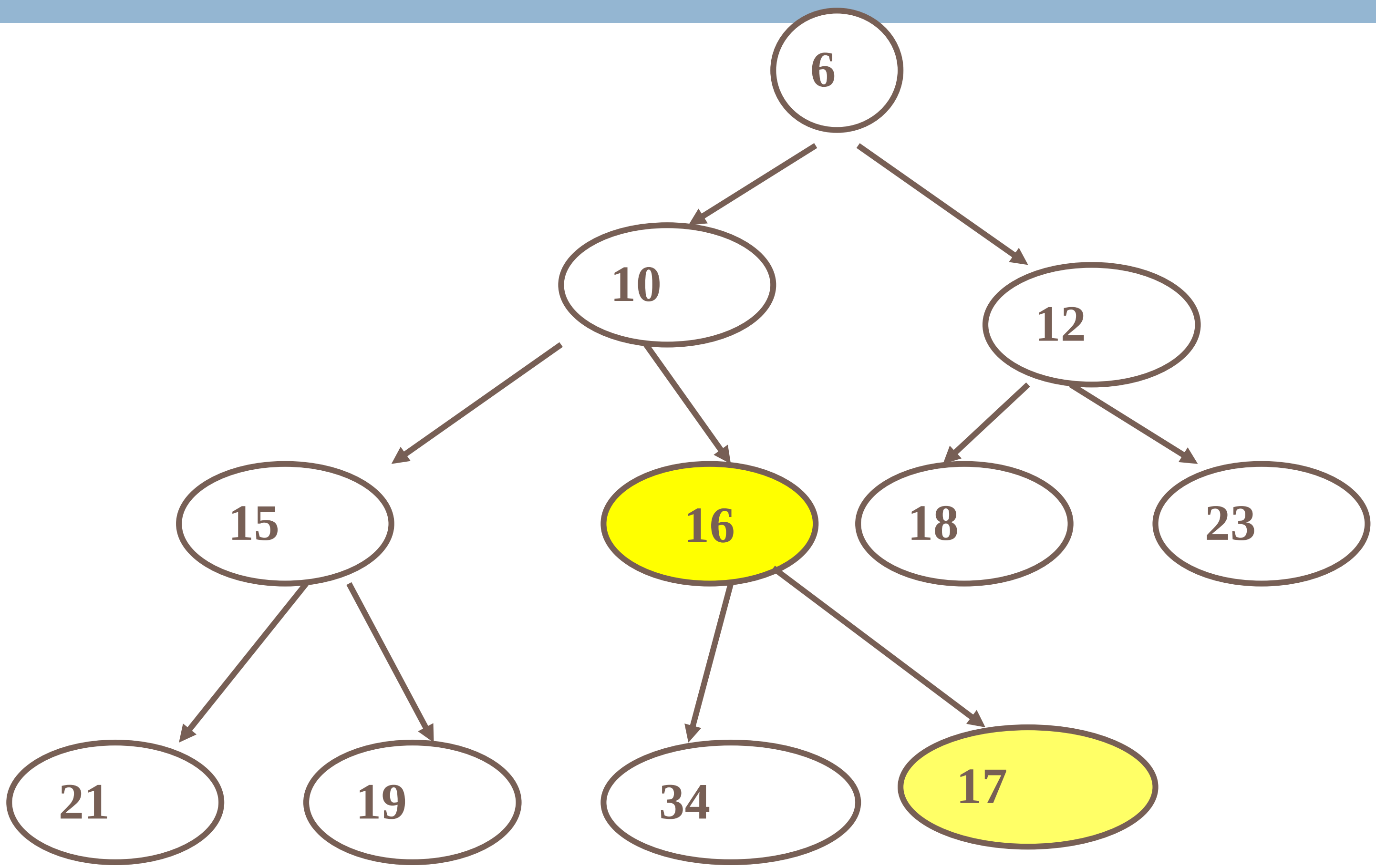


Percolate Up

Insert 16

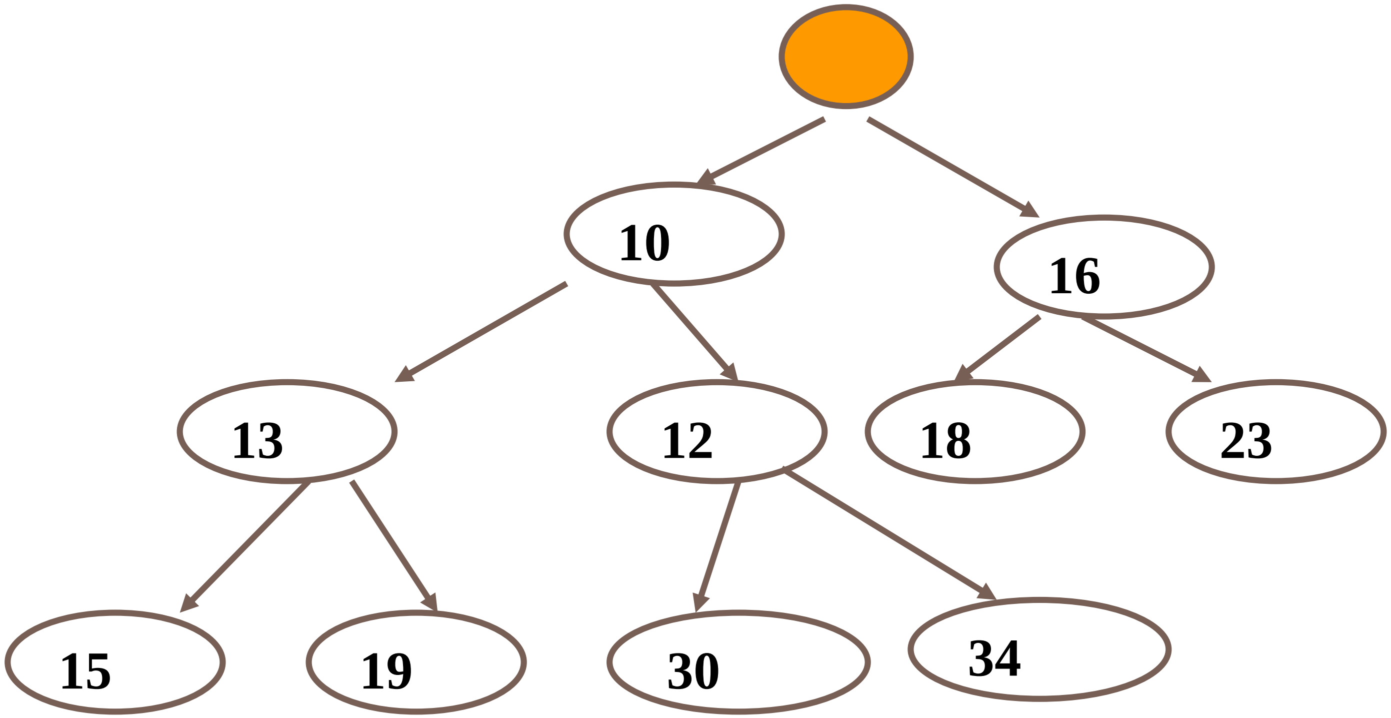


Percolate Up

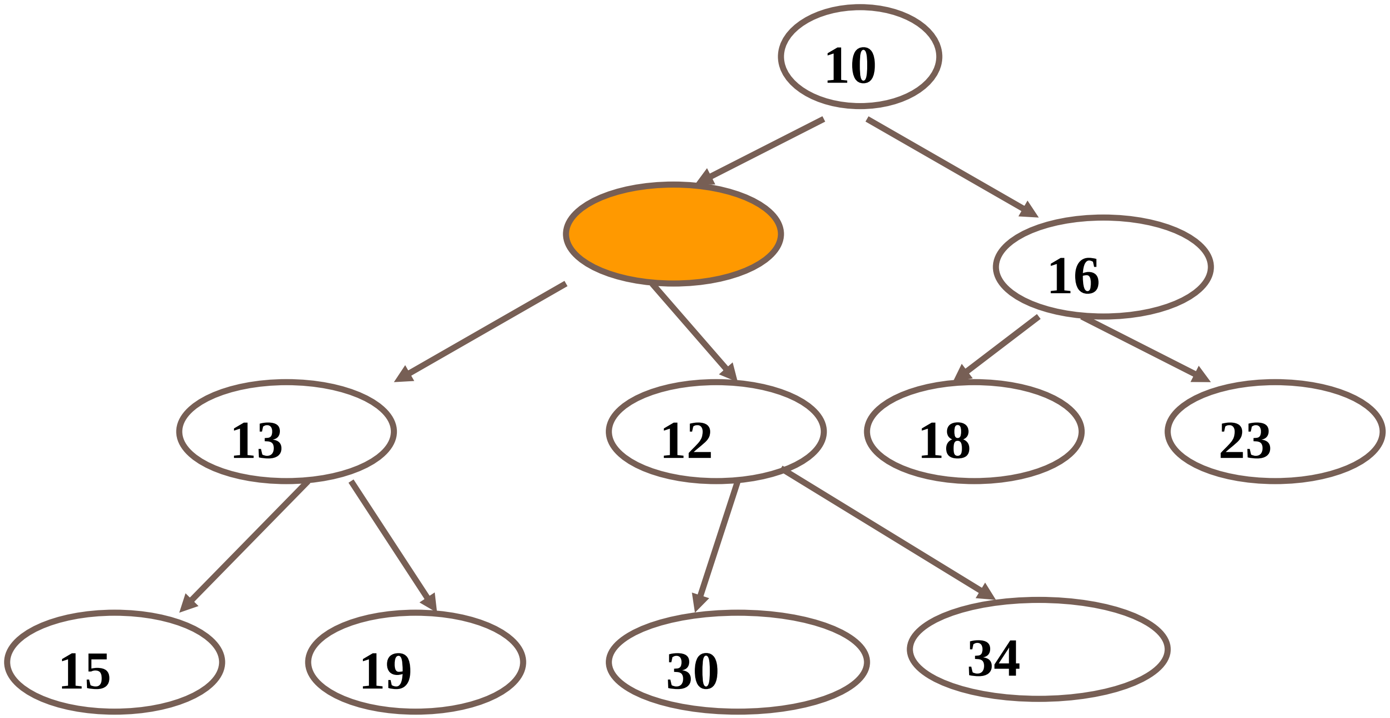


Complexity of insertion: $O(\log_2 N)$

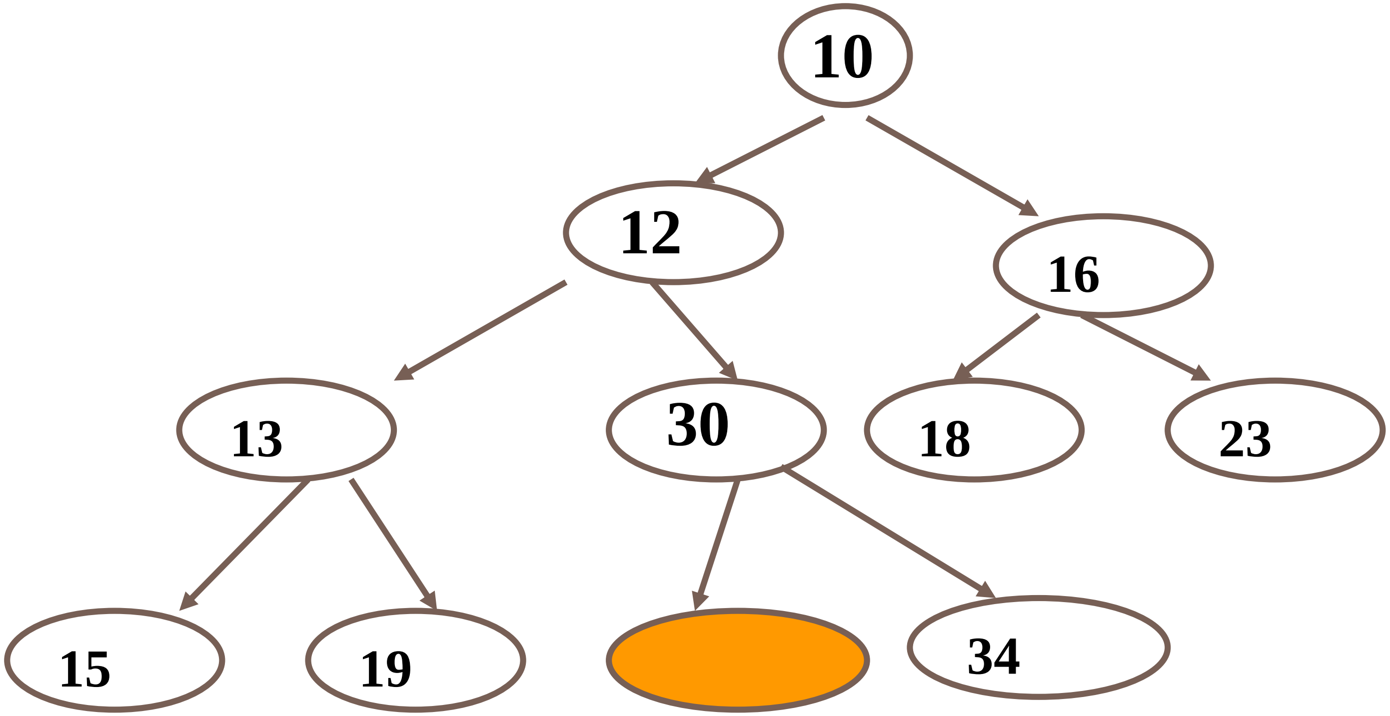
Percolate Down – Delete a Node



Percolate Down – the wrong way



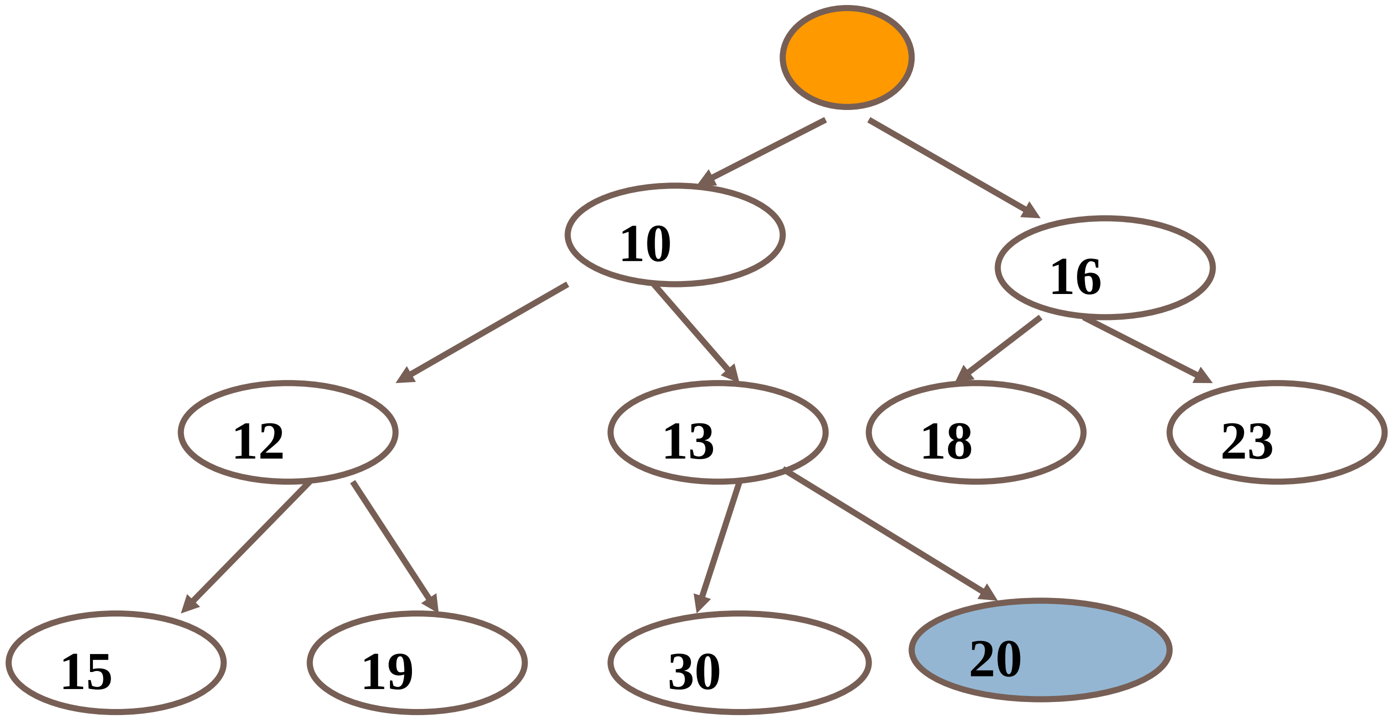
Percolate Down – the wrong way



The empty hole violates the heap-structure property

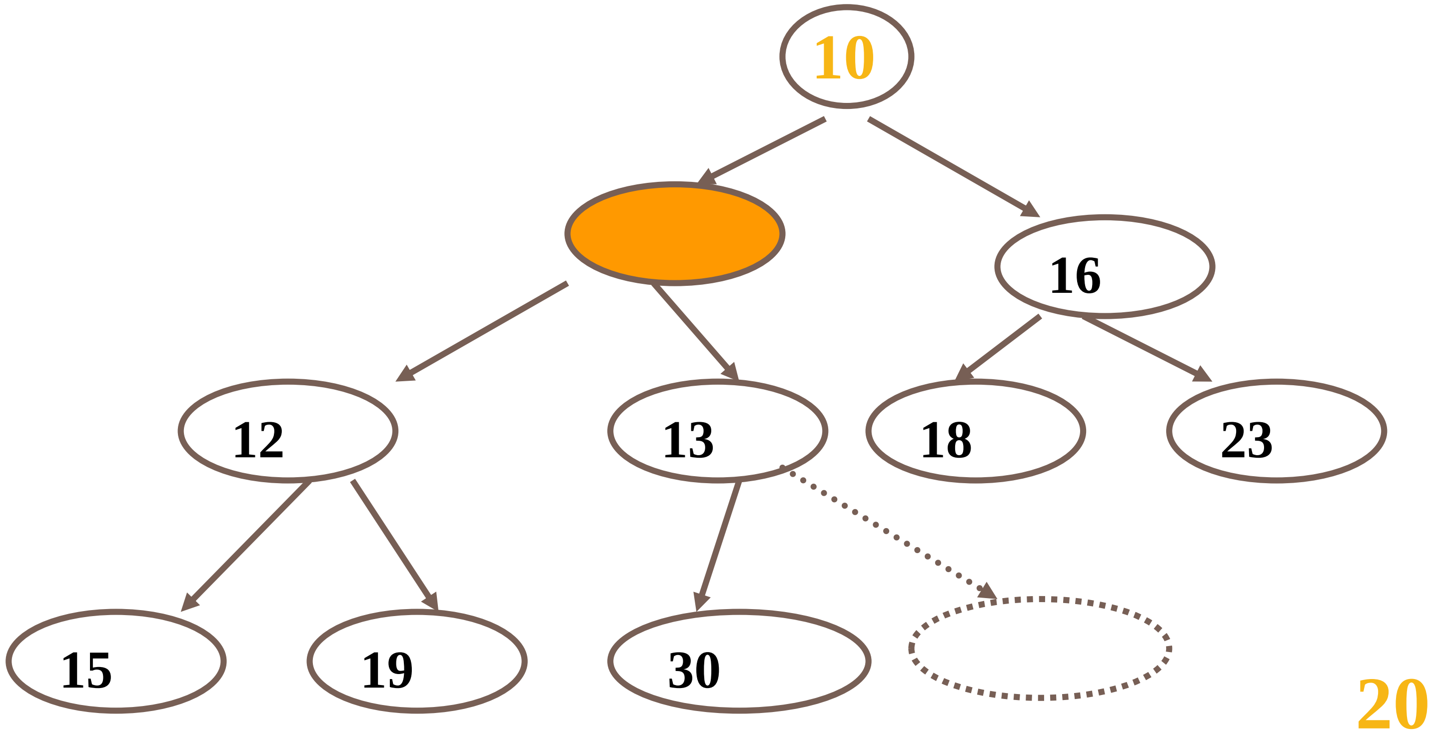
Percolate Down

Last element - 20. The hole at the root.

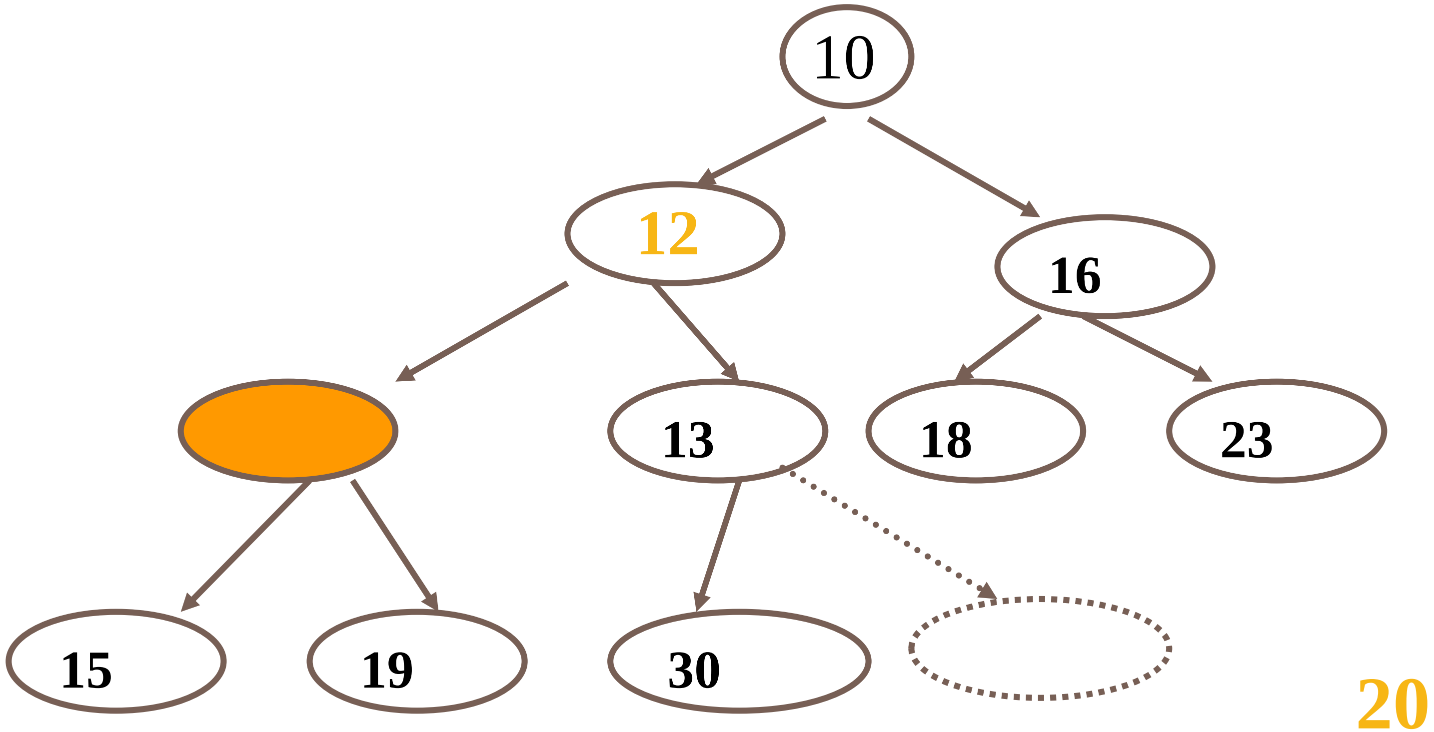


We try to insert 20 in the hole by percolating the hole down

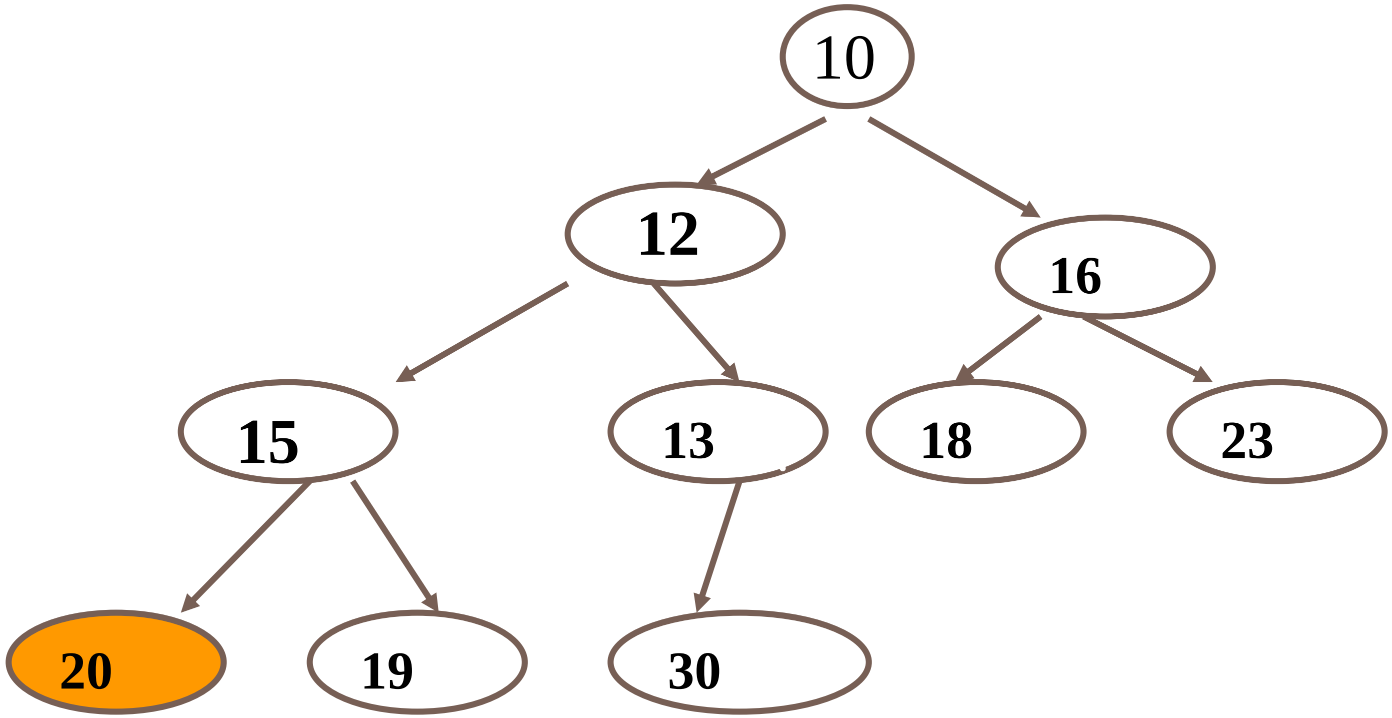
Percolate Down



Percolate Down

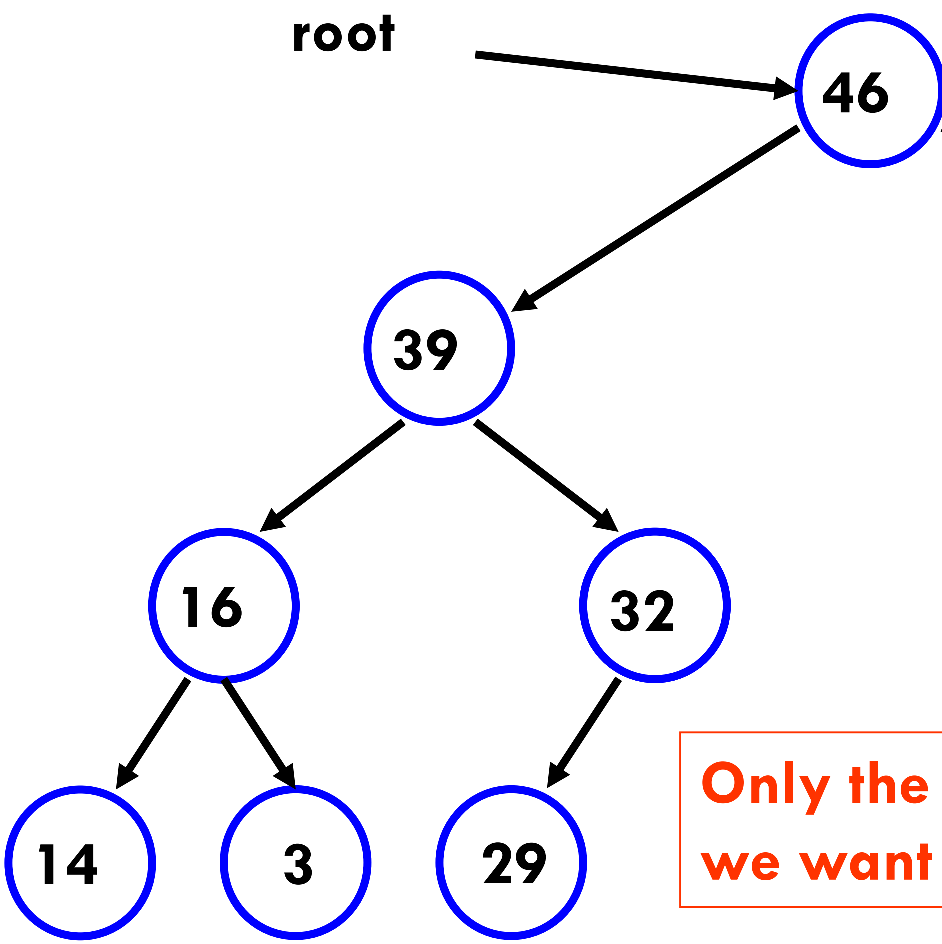


Percolate Down



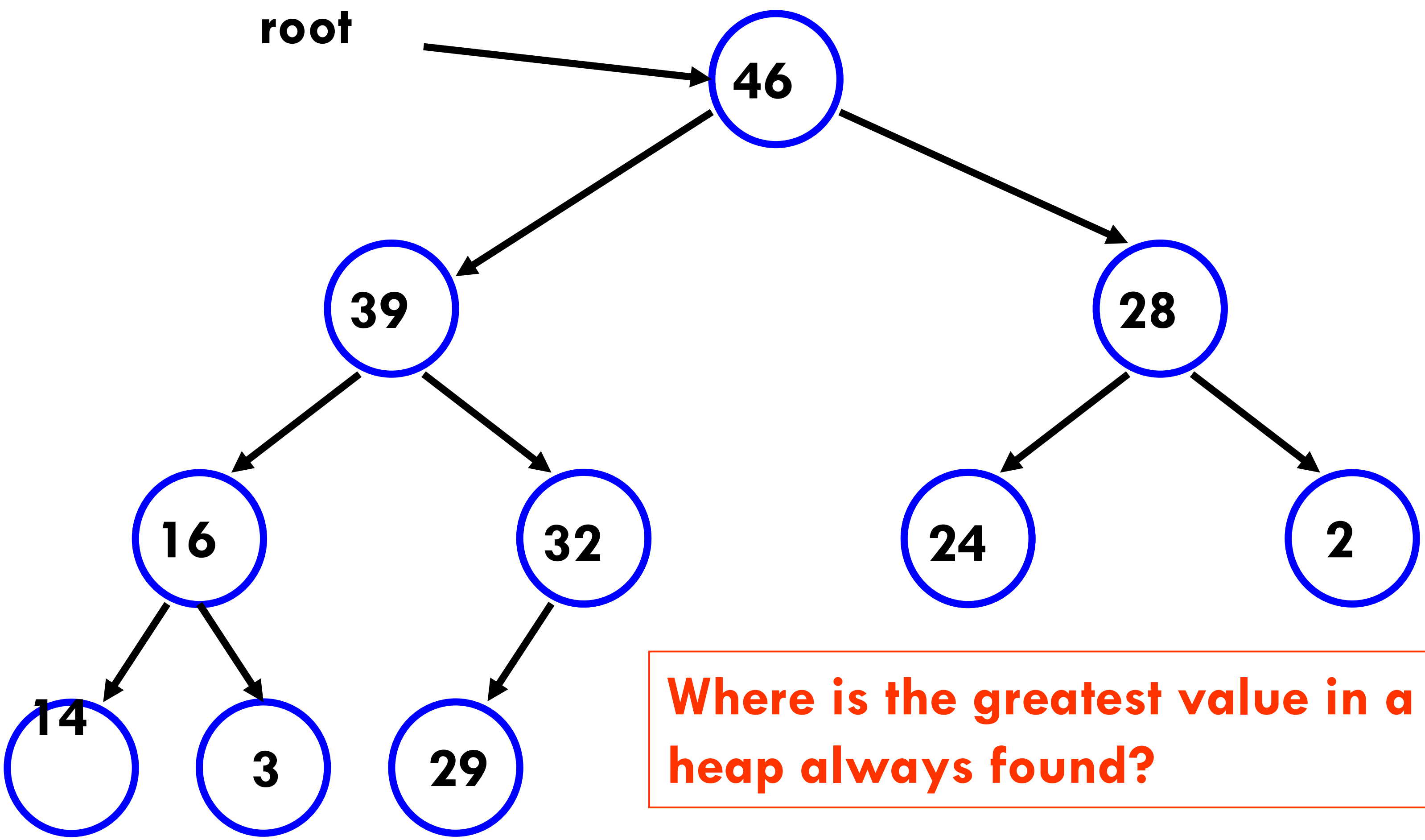
Complexity of deletion: O(logN)

Example of a Heap



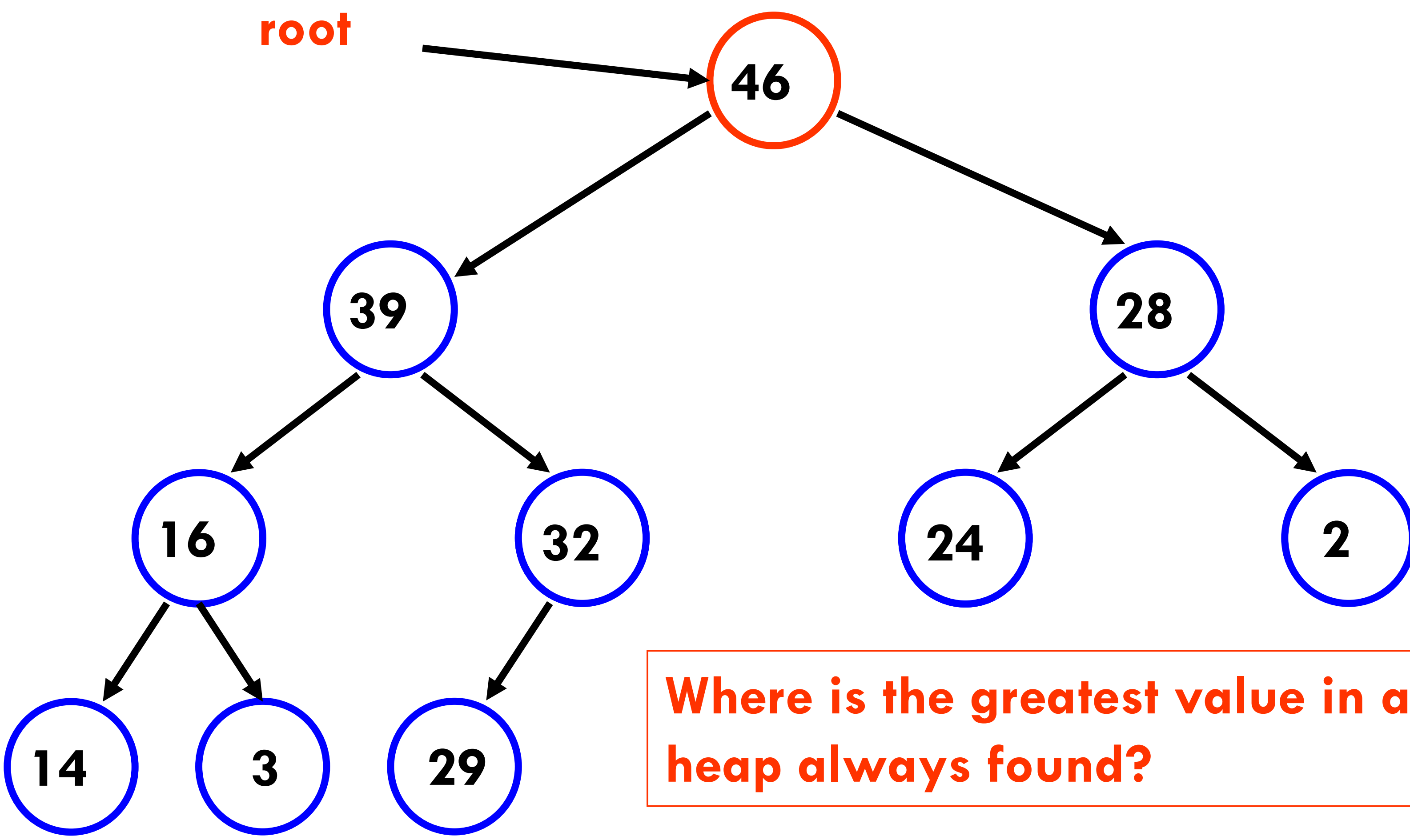
Only the data member is shown that we want to prioritize

Example of a Heap (cont.)



Where is the greatest value in a heap always found?

Example of a Heap (cont.)



Where is the greatest value in a heap always found?

Dequeue

- Dequeuing the object with the greatest value appears to be a $\Theta(1)$ operation

- However, after removing the object, we must turn the resultant structure into a heap again, for the next dequeue

- Fortunately, it only takes $O(\log_2 n)$ time to turn the structure back into a heap again (which is why dequeue in a heap is a $O(\log_2 n)$ operation

Heapify

- The process of swapping downwards to form a new heap is called heapifying
- When, we heapify, it is important that the rest of the structure is a heap, except for the root node that we are starting off with; otherwise, a new heap won't be formed
- A loop is used for heapifying; the number of times through the loop is always lg n or less, which gives the O(lg n) complexity
- Each time we swap downwards, the number of nodes we can travel to is reduced by approximately half

Part III: Heapsort

Heapsort

- Consider a priority queue with n items implemented by means of a heap
  - The space used is $O(n)$
  - Methods enqueue and removeMax take $O(\log n)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort

Heapsort

- Build a binary heap of N elements
  - $O(N)$ time
- Then perform N `deleteMax` operations
  - $log(N)$ time per `deleteMax`
- Total complexity $O(N\ log\ N)$

Questions