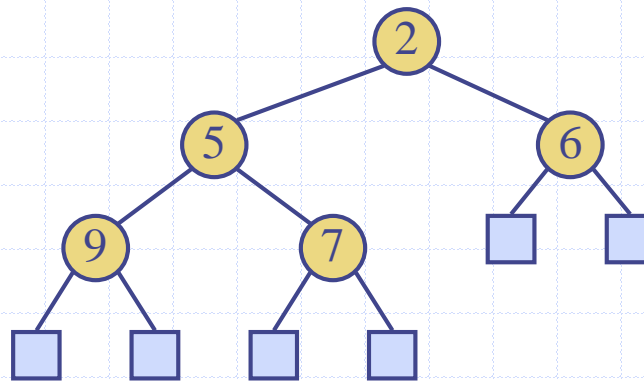


Heaps and Priority Queues

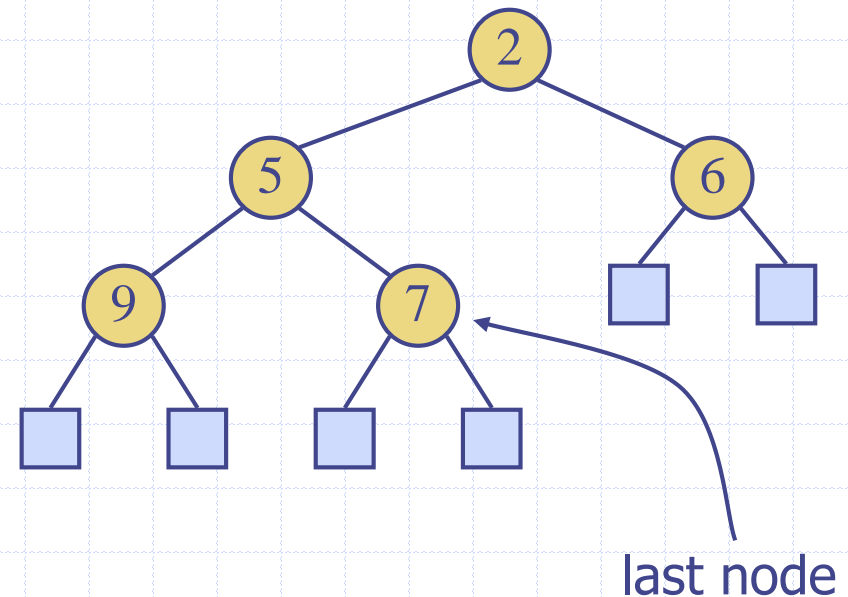


What is a heap

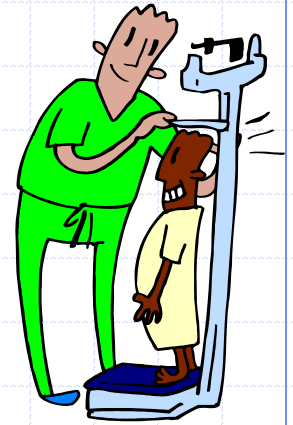


- ◆ A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
 - **Heap-Order:** for every internal node v other than the root, $key(v) \geq key(parent(v))$
 - **Complete Binary Tree:** let h be the height of the heap
 - ◆ for $i = 0, \dots, h - 1$, there are 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 internal node of depth $h - 1$



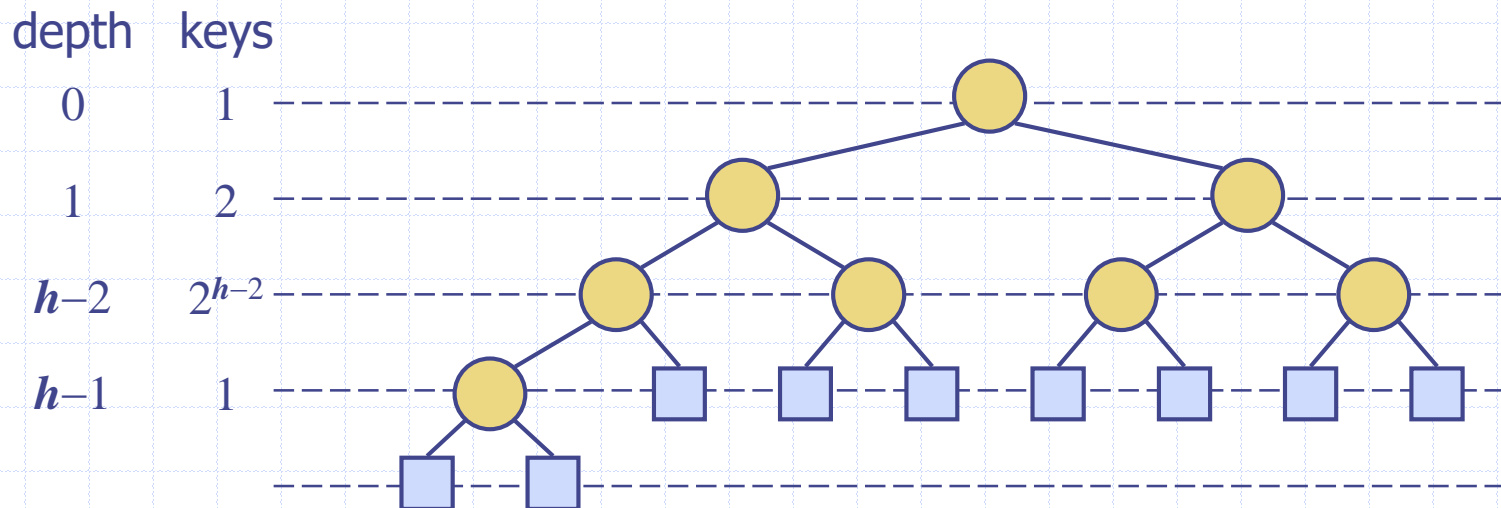
Height of a Heap



◆ **Theorem:** A heap storing n keys has height $O(\log n)$

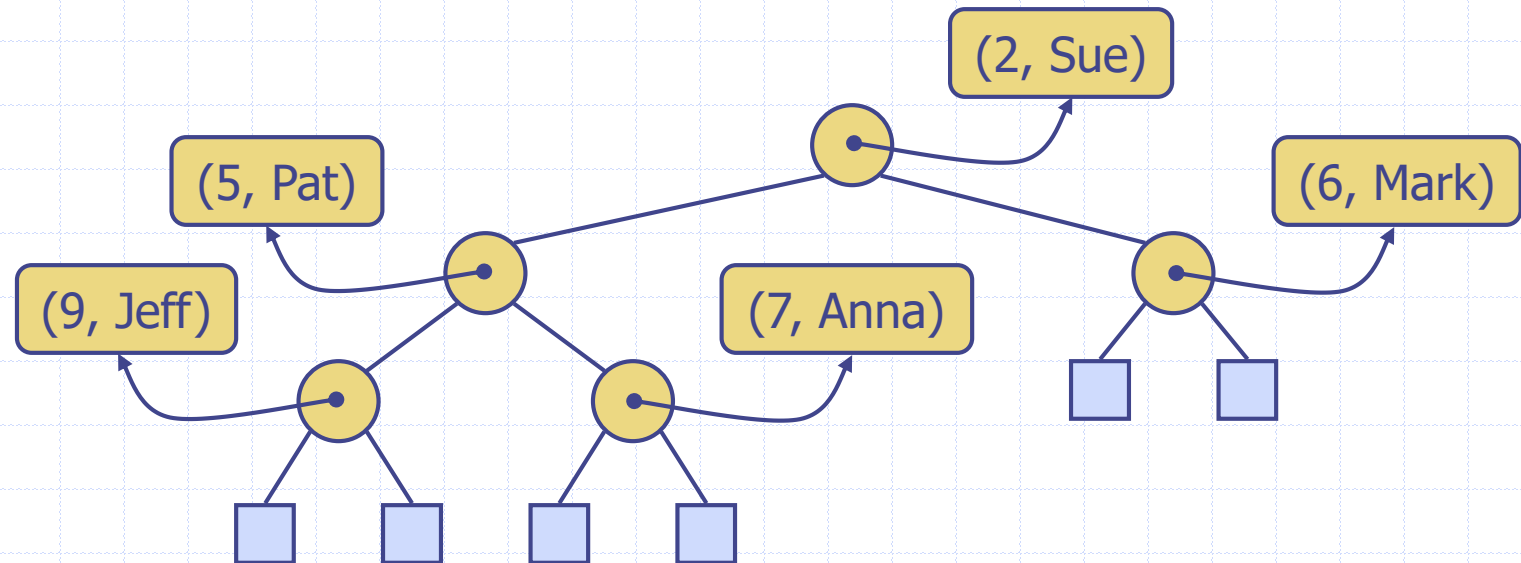
Proof: (we apply the complete binary tree property)

- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1$
- Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



Heaps and Priority Queues

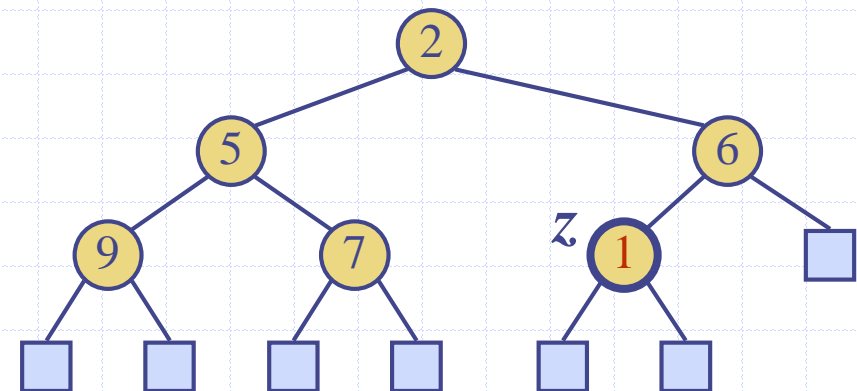
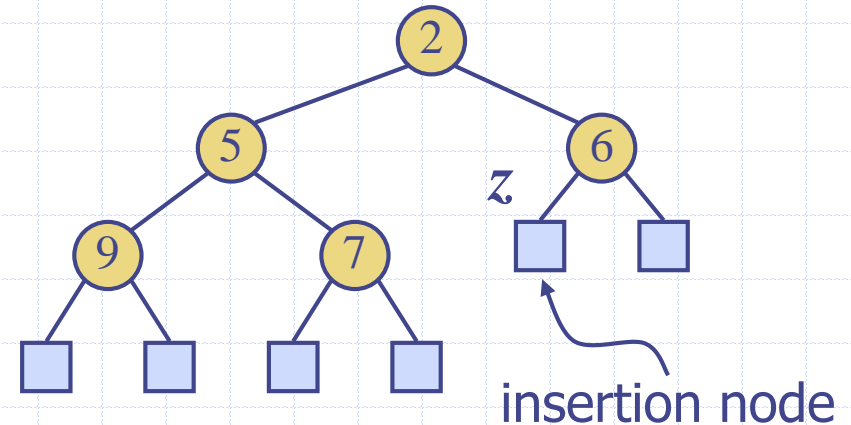
- ◆ We can use a heap to implement a priority queue
- ◆ We store a (key, element) item at each internal node
- ◆ We keep track of the position of the last node
- ◆ For simplicity, we show only the keys in the pictures



Insertion into a Heap

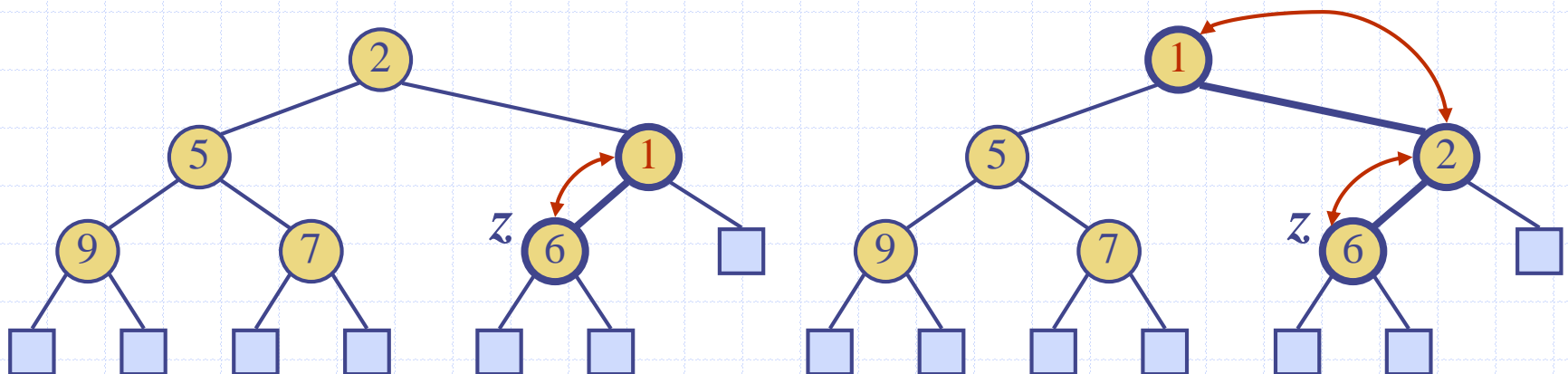


- ◆ The insertion algorithm corresponds to the insertion of a key k to the heap
- ◆ The insertion algorithm consists of three steps
 - Find the insertion node z (the new last node)
 - Store k at z and expand z into an internal node
 - Restore the heap-order property (discussed next)



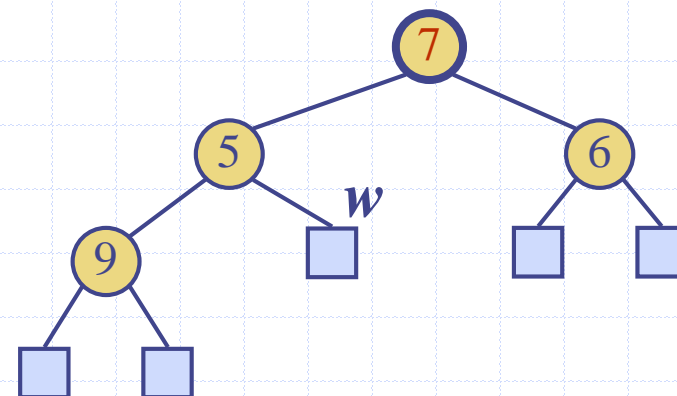
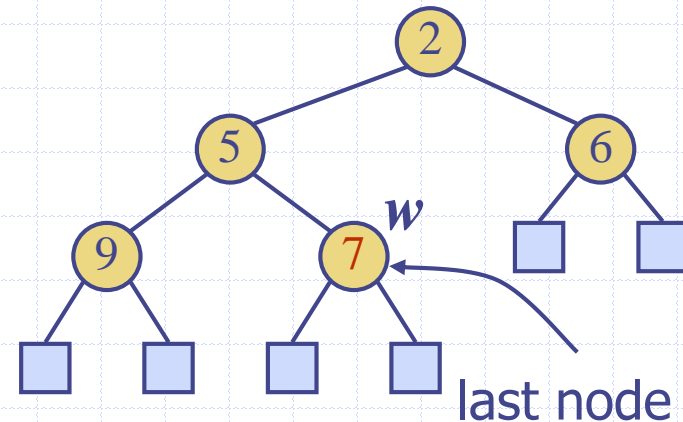
Upheap

- ◆ 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



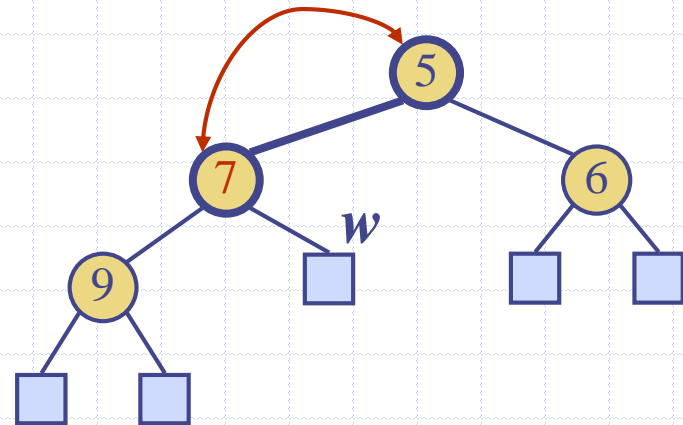
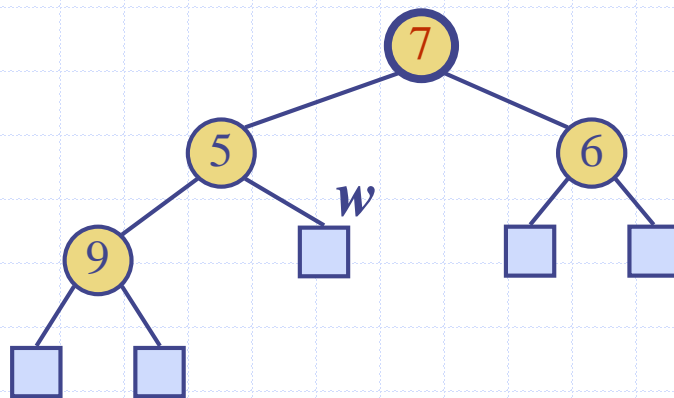
Removal from a Heap

- ◆ Method `removeMin` of the priority queue ADT corresponds to the removal of the root key from the heap
- ◆ The removal algorithm consists of three steps
 - Replace the root key with the key of the last node w
 - Compress w and its children into a leaf
 - Restore the heap-order property (discussed next)

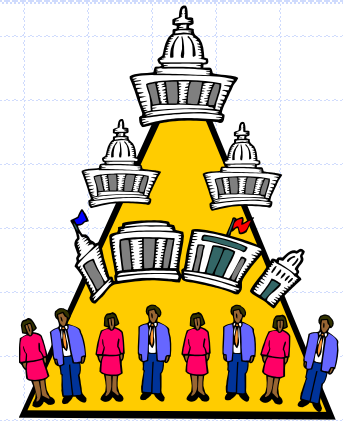


Downheap

- ◆ 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



Heap-Sort

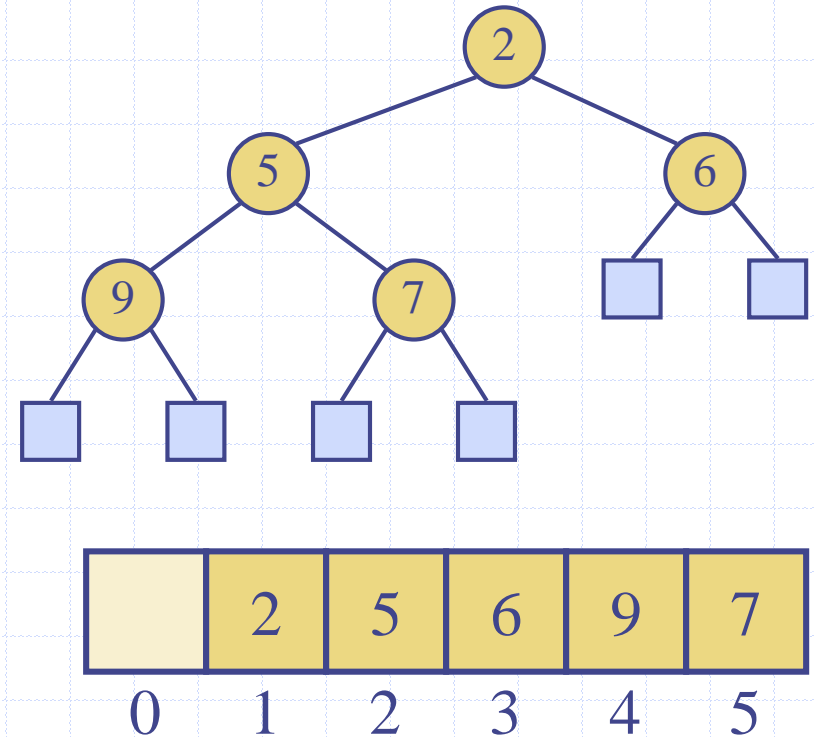


- ◆ Consider a priority queue with n items implemented by means of a heap
 - the space used is $O(n)$
 - methods **insertItem** and **removeMin** take $O(\log n)$ time

- ◆ 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
- ◆ Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

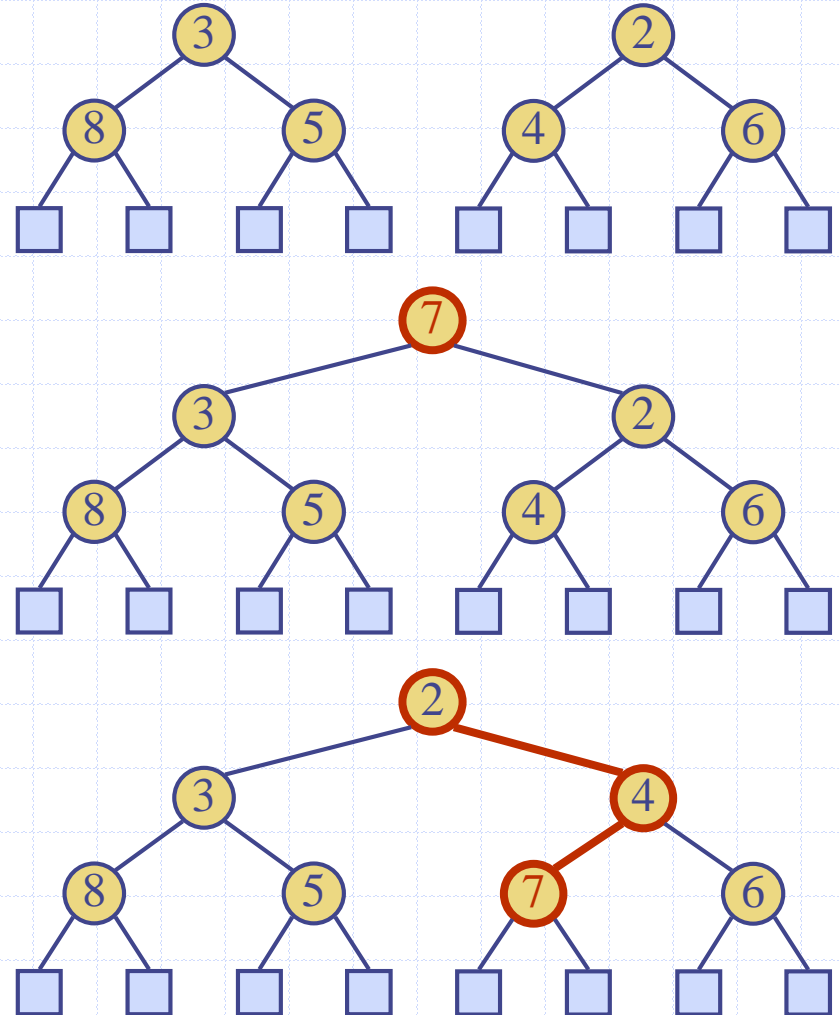
Vector-based Heap Implementation

- ◆ 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 leaves are not represented
- ◆ The cell at rank 0 is not used
- ◆ Yields in-place heap-sort



Merging Two Heaps

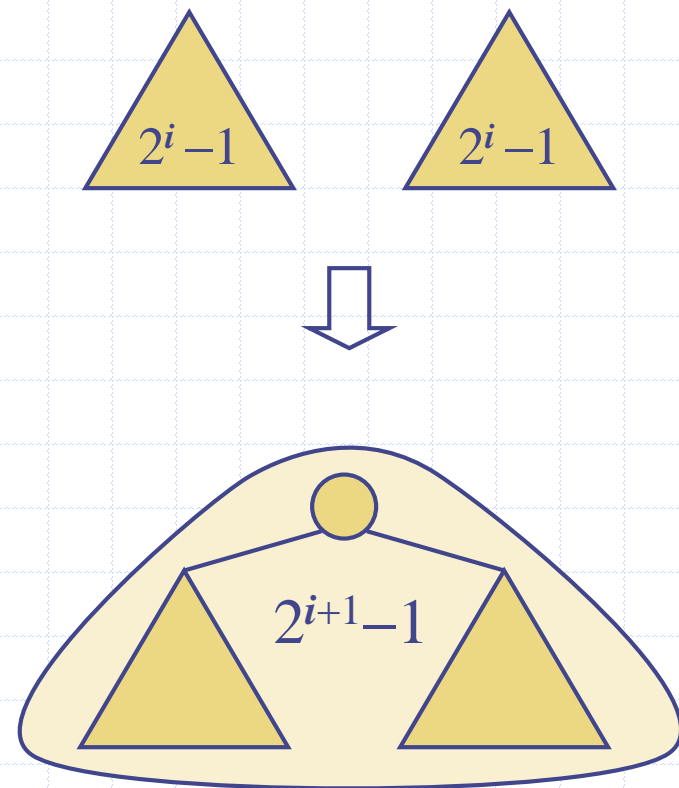
- ◆ We are given two two heaps and a key k
- ◆ We create a new heap with the root node storing k and with the two heaps as subtrees
- ◆ We perform downheap to restore the heap-order property

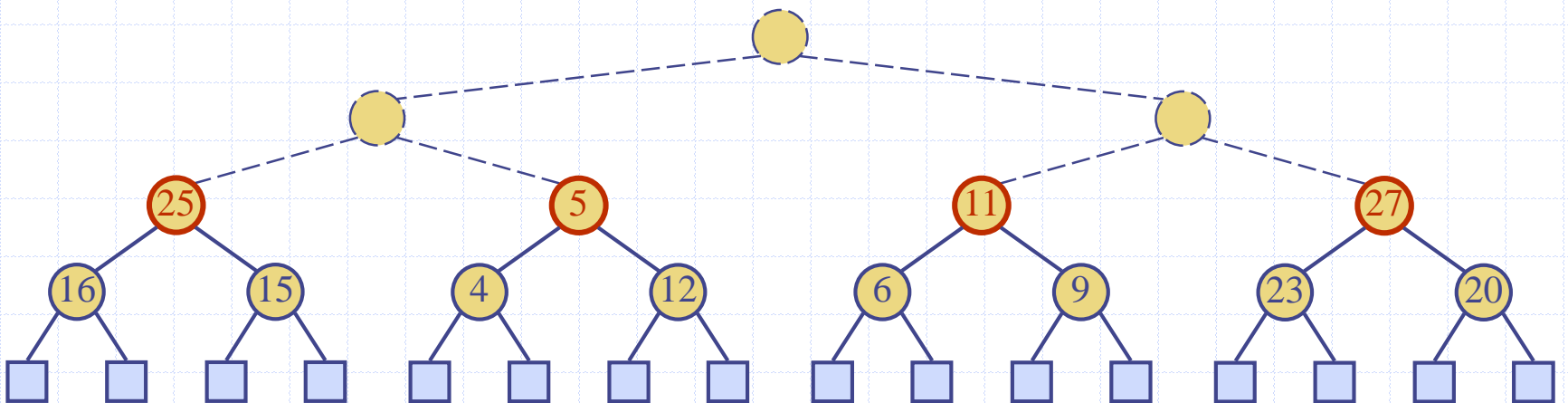


Bottom-up Heap Construction

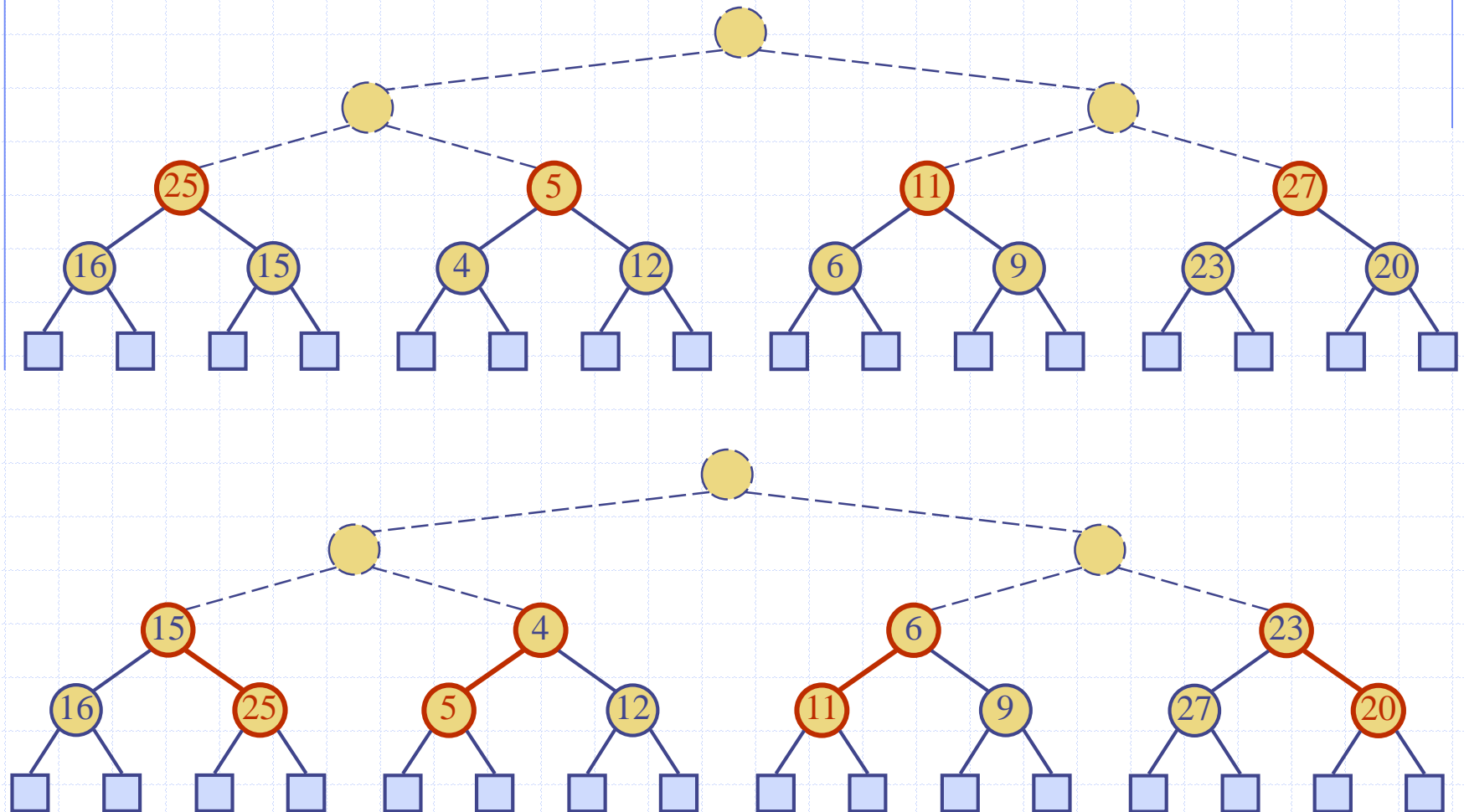


- ◆ We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- ◆ In phase i , pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

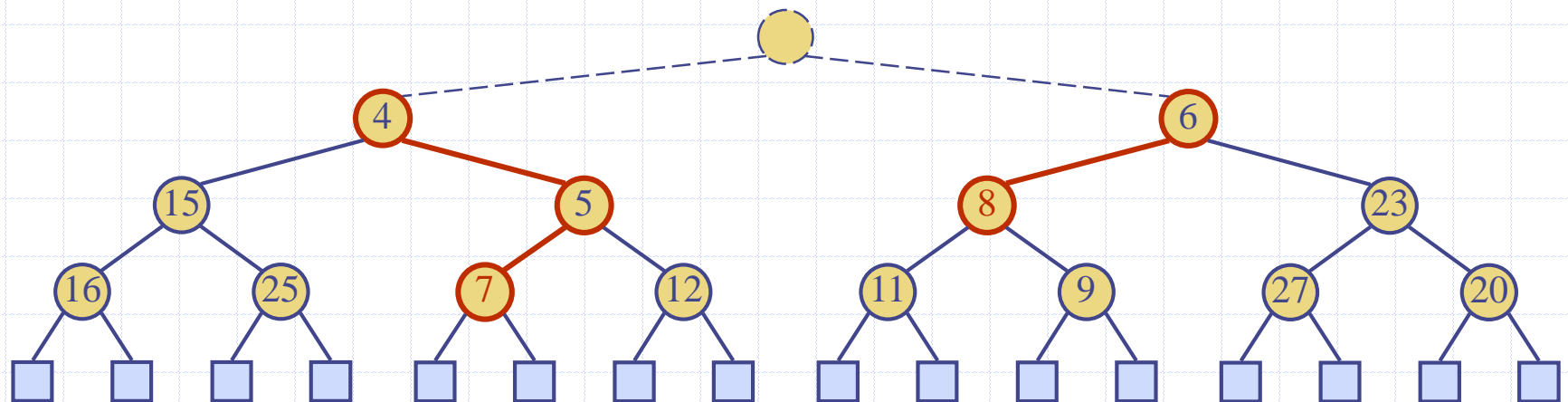
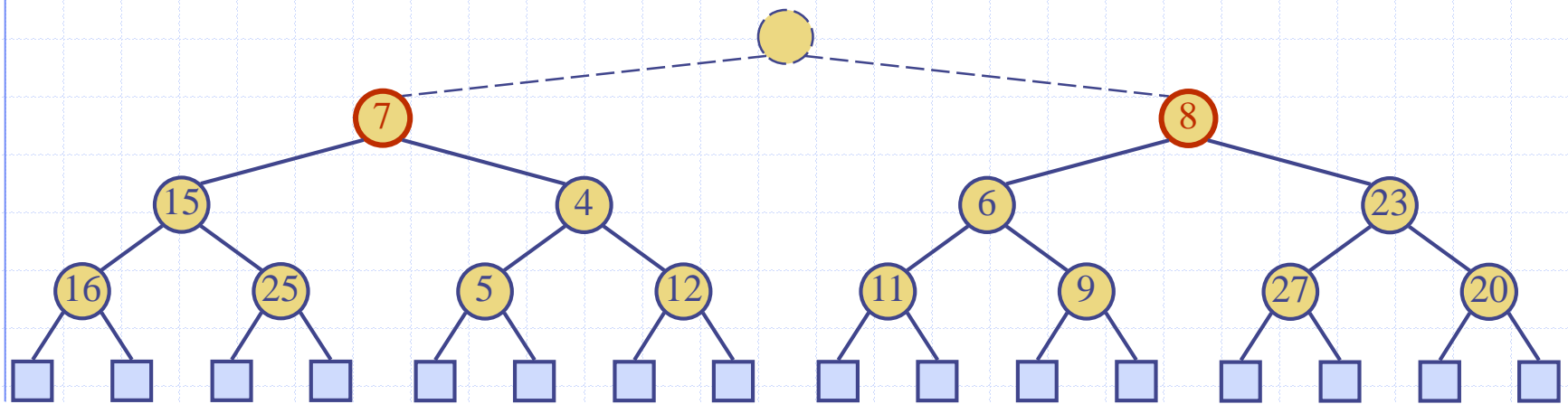




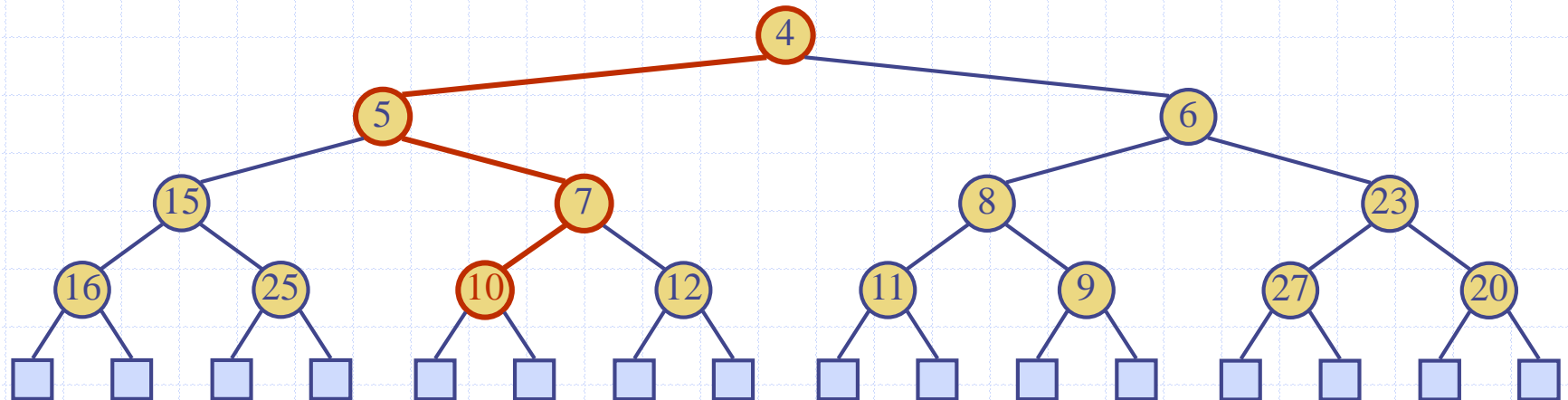
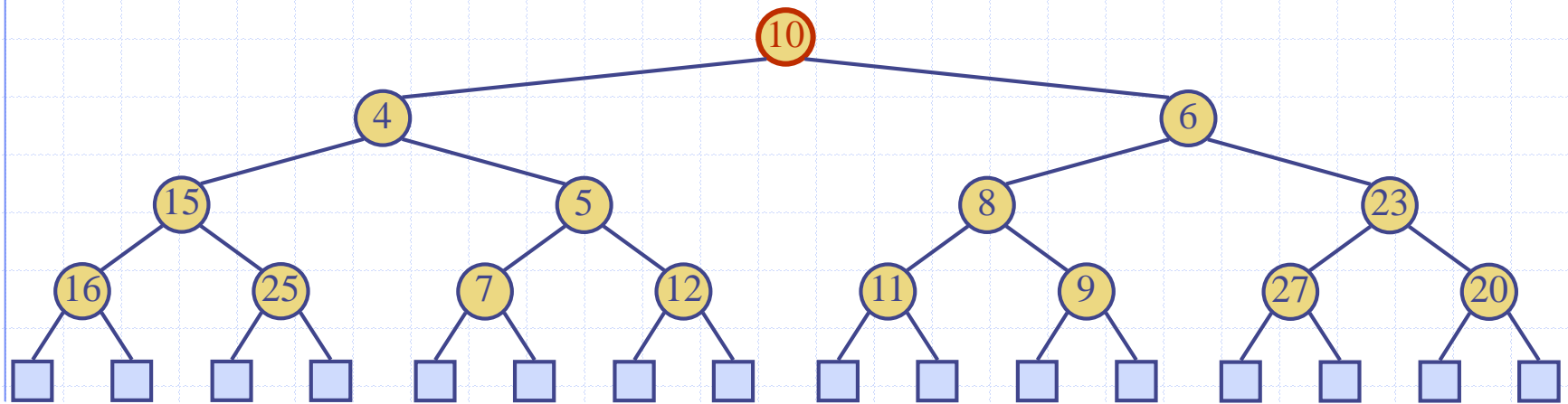
Example (contd.)



Example (contd.)



Example (end)



Building a heap

Running time

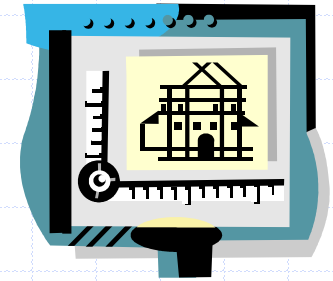
Easy: $O(n \log n)$ [Why?]

Better estimate: $O(n)$.

Proof: When HEAPIFY is called on a node of height h , it takes $O(h)$ time units.

Therefore, the total time is $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O(n)$.

Exercise: prove that $\sum_{h=1, \dots, \infty} h/2^h = O(1)$.



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 heap-sort

