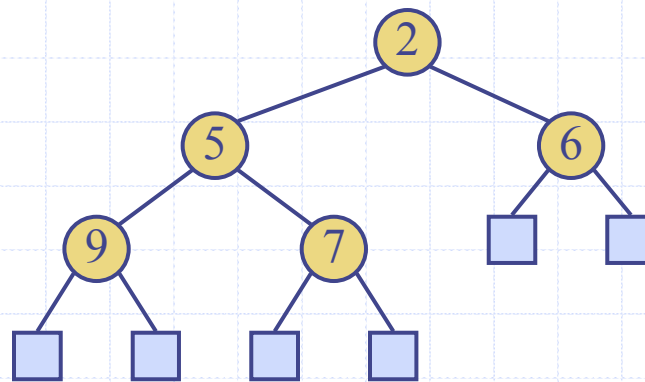


Heaps and Priority Queues

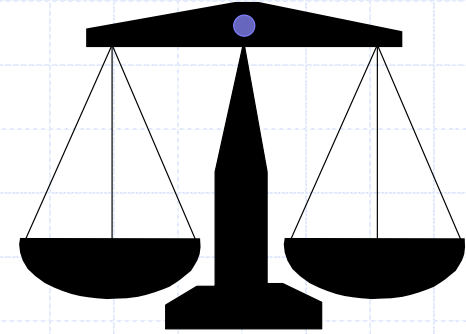


Priority Queue ADT (§ 2.4.1)



- ◆ A priority queue stores a collection of items
- ◆ An item is a pair (key, element)
- ◆ Main methods of the Priority Queue ADT
 - **insertItem(k, o)**
inserts an item with key k and element o
 - **removeMin()**
removes the item with smallest key and returns its element
- ◆ Additional methods
 - **minKey()**
returns, but does not remove, the smallest key of an item
 - **minElement()**
returns, but does not remove, the element of an item with smallest key
 - **size(), isEmpty()**
- ◆ Applications:
 - Standby flyers
 - Auctions
 - Stock market

Total Order Relation



- ◆ Keys in a priority queue can be arbitrary objects on which an order is defined
- ◆ Two distinct items in a priority queue can have the same key
- ◆ Mathematical concept of total order relation \leq
 - **Reflexive** property:
 $x \leq x$
 - **Antisymmetric** property:
 $x \leq y \wedge y \leq x \Rightarrow x = y$
 - **Transitive** property:
 $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT (§ 2.4.1)



- ◆ A comparator encapsulates the action of comparing two objects according to a given total order relation
 - ◆ A generic priority queue uses an auxiliary comparator
 - ◆ The comparator is external to the keys being compared
 - ◆ When the priority queue needs to compare two keys, it uses its comparator
- ◆ Methods of the Comparator ADT, all with Boolean return type
 - `isLessThan(x, y)`
 - `isLessThanOrEqualTo(x,y)`
 - `isEqualTo(x,y)`
 - `isGreaterThan(x, y)`
 - `isGreaterThanOrEqualTo(x,y)`
 - `isComparable(x)`

Sorting with a Priority Queue (§ 2.4.2)



- ◆ We can use a priority queue to sort a set of comparable elements
 - Insert the elements one by one with a series of `insertItem(e, e)` operations
 - Remove the elements in sorted order with a series of `removeMin()` operations
- ◆ The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort(S, C)*

Input sequence S , comparator C
for the elements of S

Output sequence S sorted in
increasing order according to C

$P \leftarrow$ priority queue with
comparator C

while $\neg S.isEmpty()$

$e \leftarrow S.remove(S.first())$

$P.insertItem(e, e)$

while $\neg P.isEmpty()$

$e \leftarrow P.removeMin()$

$S.insertLast(e)$

Sequence-based Priority Queue

- ◆ Implementation with an unsorted list



- ◆ Performance:

- **insertItem** takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
- **removeMin**, **minKey** and **minElement** take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

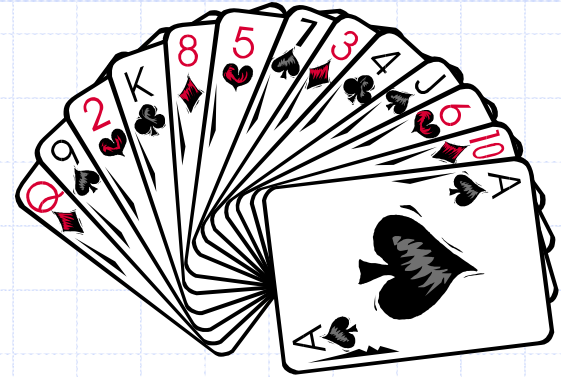
- ◆ Implementation with a sorted list



- ◆ Performance:

- **insertItem** takes $O(n)$ time since we have to find the place where to insert the item
- **removeMin**, **minKey** and **minElement** take $O(1)$ time since the smallest key is at the beginning of the sequence

Selection-Sort



- ◆ Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence

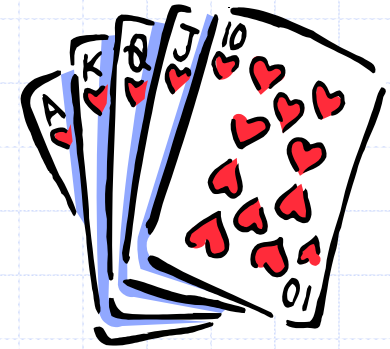


- ◆ Running time of Selection-sort:
 - Inserting the elements into the priority queue with n **insertItem** operations takes $O(n)$ time
 - Removing the elements in sorted order from the priority queue with n **removeMin** operations takes time proportional to

$$1 + 2 + \dots + n$$

- ◆ Selection-sort runs in $O(n^2)$ time

Insertion-Sort



- ◆ Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence



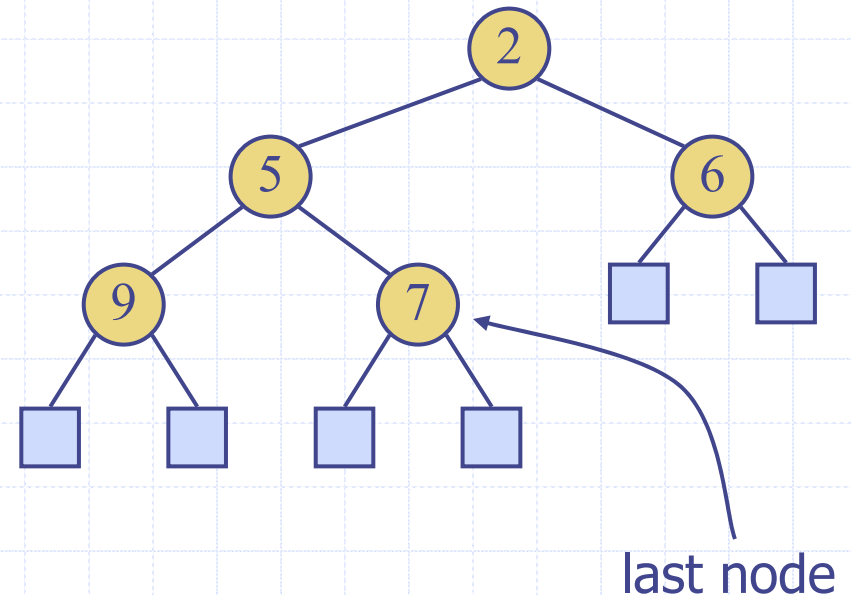
- ◆ Running time of Insertion-sort:
 - Inserting the elements into the priority queue with n **insertItem** operations takes time proportional to
$$1 + 2 + \dots + n$$
 - Removing the elements in sorted order from the priority queue with a series of n **removeMin** operations takes $O(n)$ time
- ◆ Insertion-sort runs in $O(n^2)$ time

What is a heap (§2.4.3)



- ◆ 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 $h - 1$



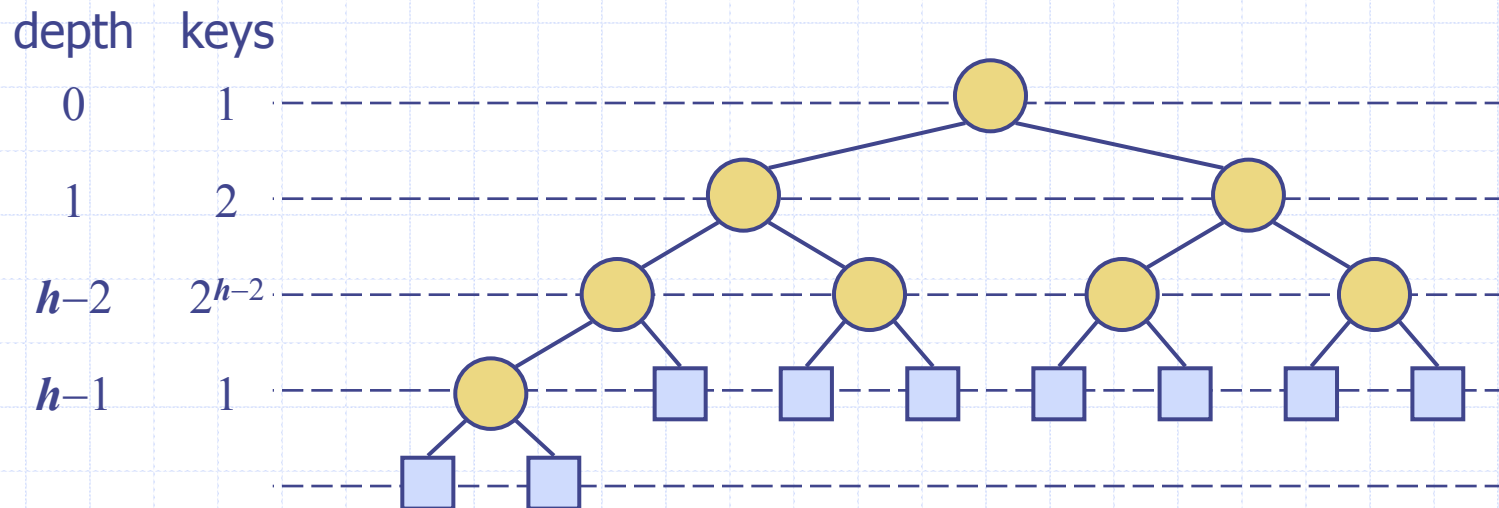
Height of a Heap (§2.4.3)



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

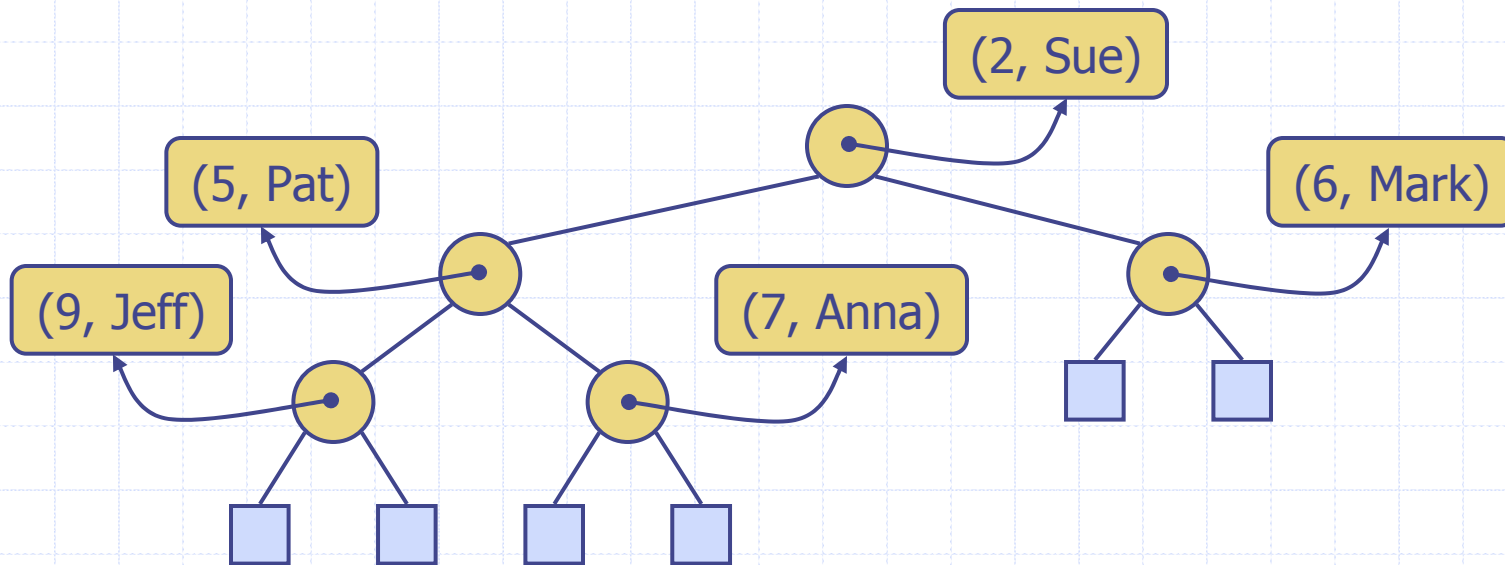
Proof: (we apply the complete binary tree property)

- 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

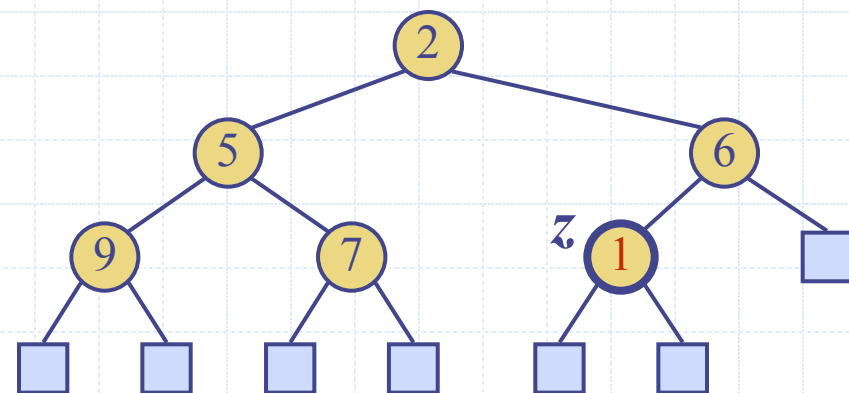
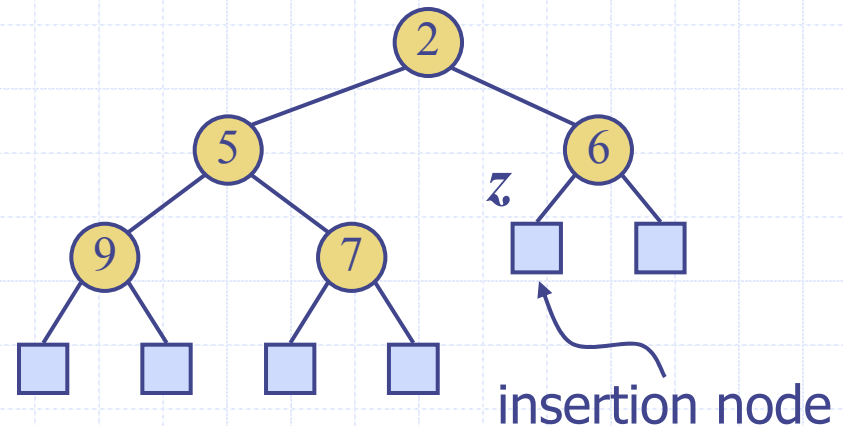
- ◆ We can use a heap to implement a priority queue
- ◆ We store a (key, element) item at each internal node
- ◆ We keep track of the position of the last node
- ◆ For simplicity, we show only the keys in the pictures



Insertion into a Heap (§2.4.3)

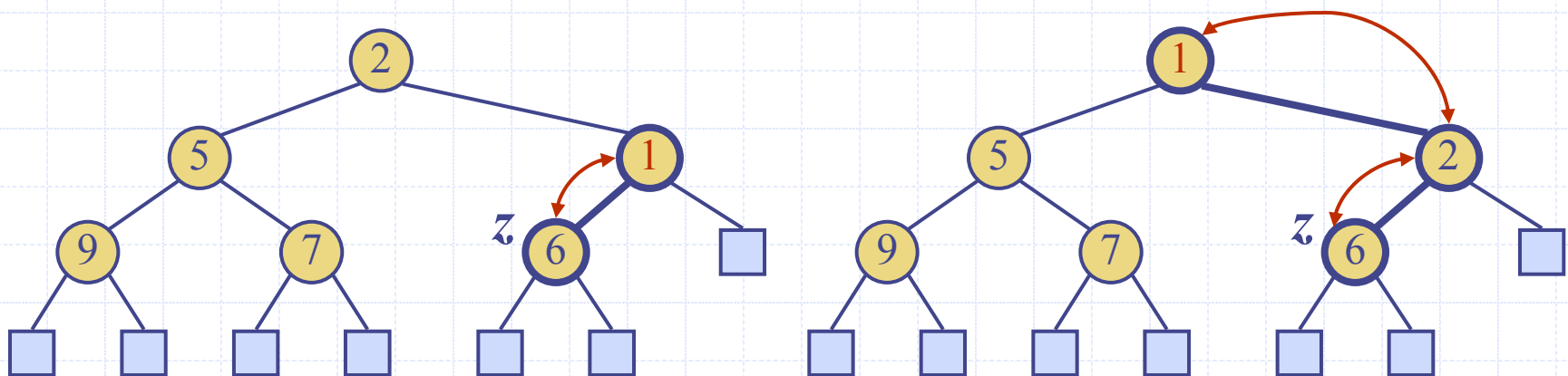


- ◆ Method `insertItem` of the priority queue ADT corresponds to the insertion of a key k to the heap
- ◆ The insertion algorithm consists of three steps
 - Find the insertion node z (the new last node)
 - Store k at z and expand z into an internal node
 - Restore the heap-order property (discussed next)



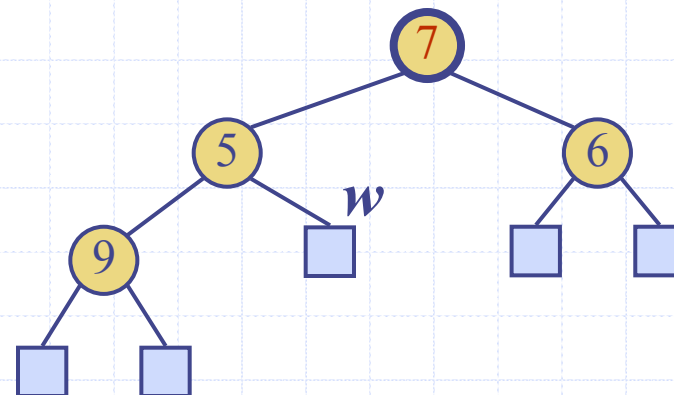
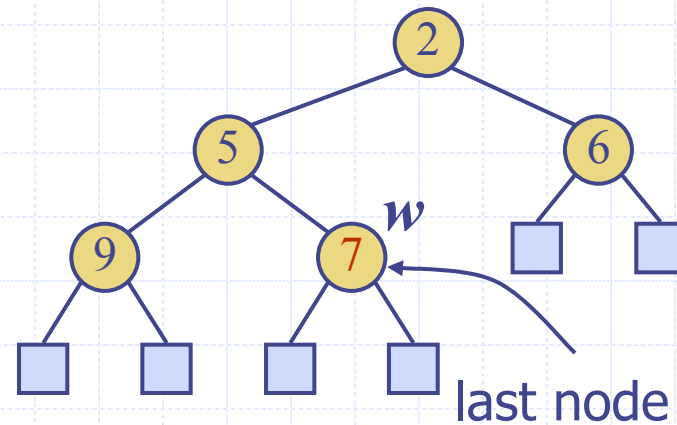
Upheap

- ◆ 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



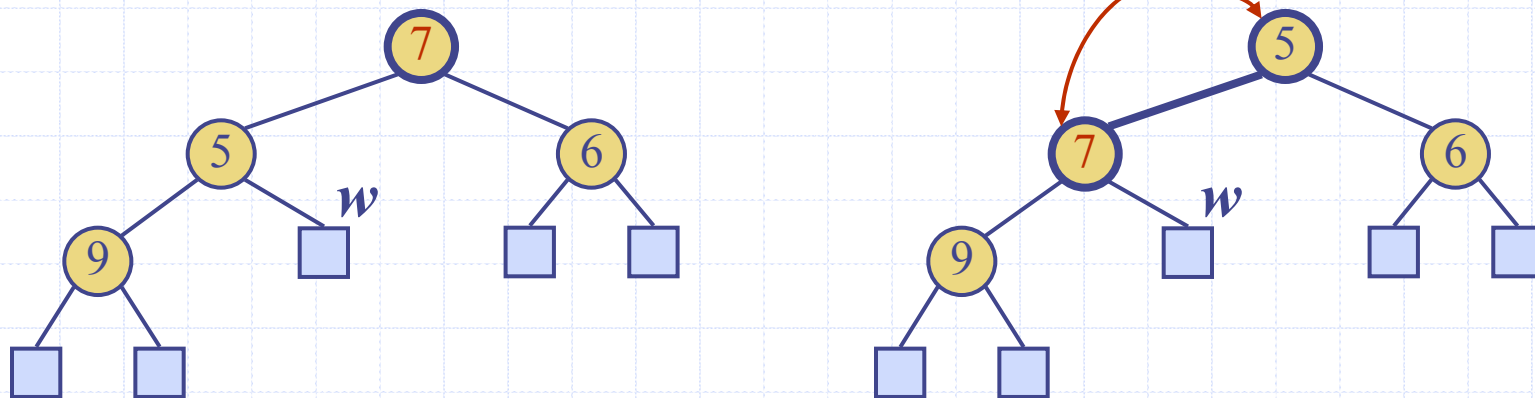
Removal from a Heap (§2.4.3)

- ◆ 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)



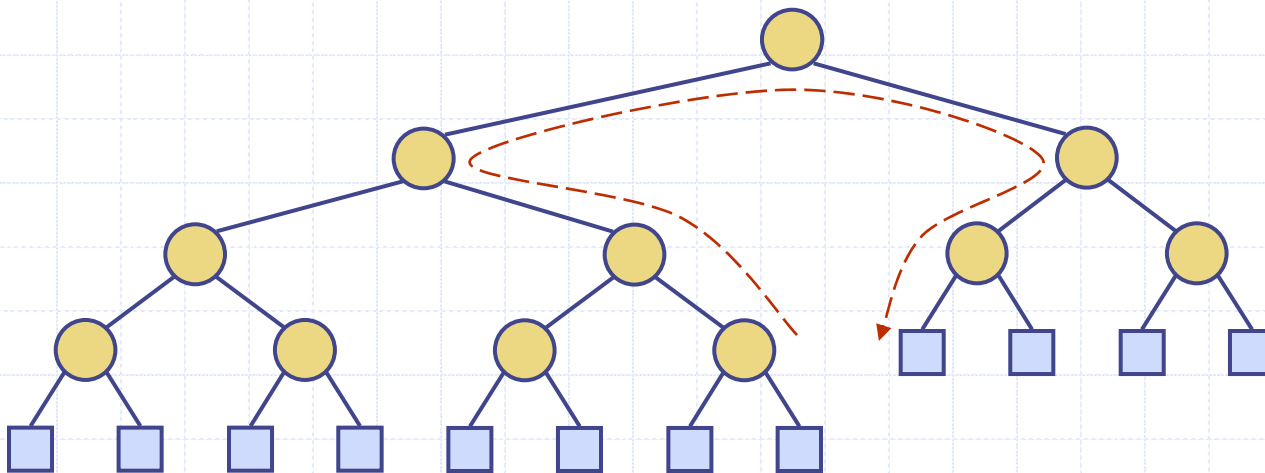
Downheap

- ◆ 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
- ◆ Upheap 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

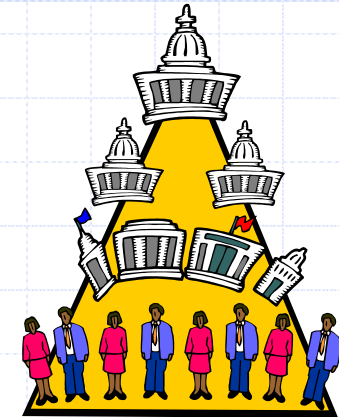


Updating the Last Node

- ◆ The insertion node can be found by traversing a path of $O(\log n)$ nodes
 - While the current node is a right child, go to the parent node
 - If the current node is a left child, go to the right child
 - While the current node is internal, go to the left child
- ◆ Similar algorithm for updating the last node after a removal



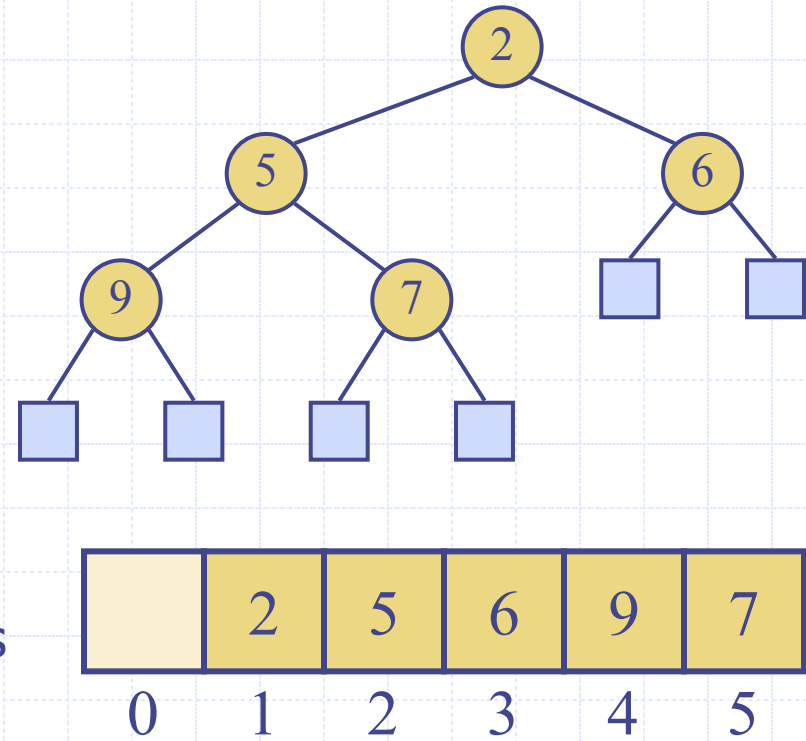
Heap-Sort (§2.4.4)



- ◆ 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
 - methods **size**, **isEmpty**, **minKey**, and **minElement** take time $O(1)$ 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

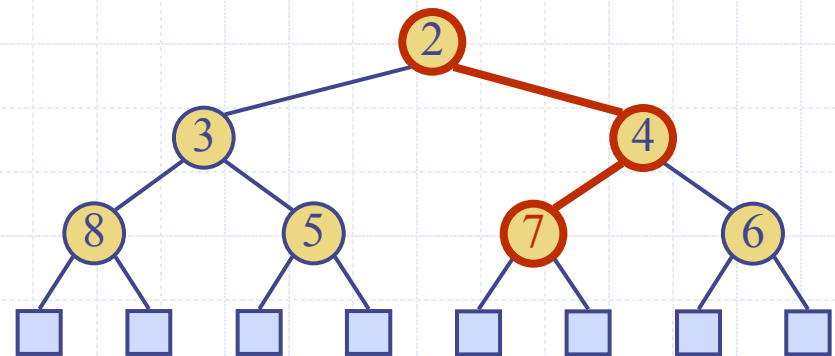
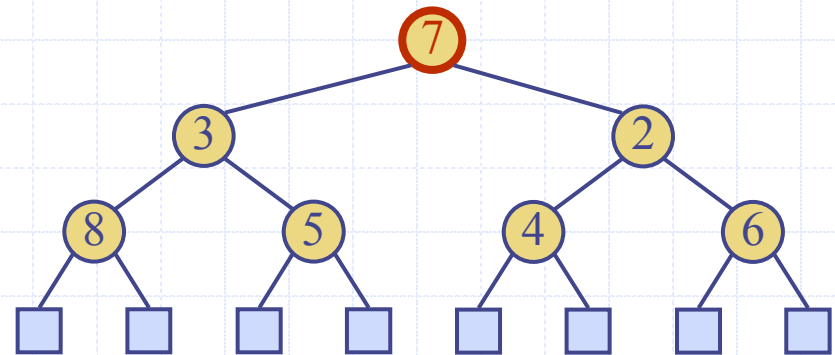
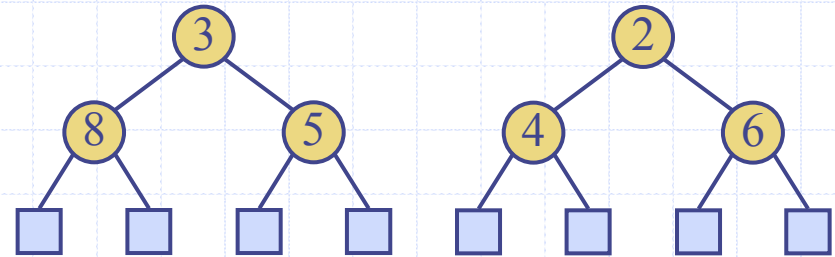
Vector-based Heap Implementation (§2.4.3)

- ◆ 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
- ◆ Operation `insertItem` corresponds to inserting at rank $n + 1$
- ◆ Operation `removeMin` corresponds to removing at rank n
- ◆ Yields in-place heap-sort



Merging Two Heaps

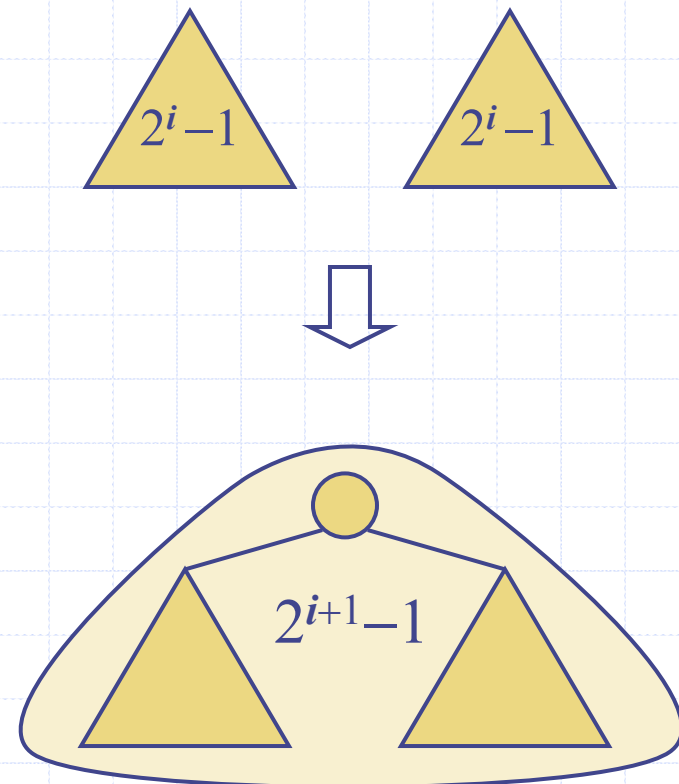
- ◆ We are given two two heaps and a key k
- ◆ We create a new heap with the root node storing k and with the two heaps as subtrees
- ◆ We perform downheap to restore the heap-order property



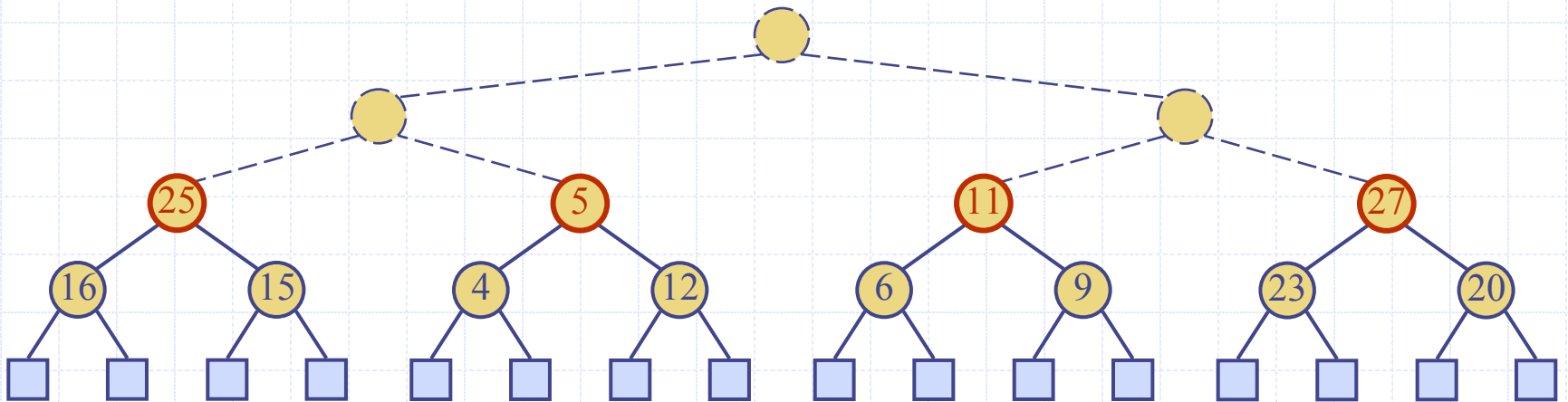
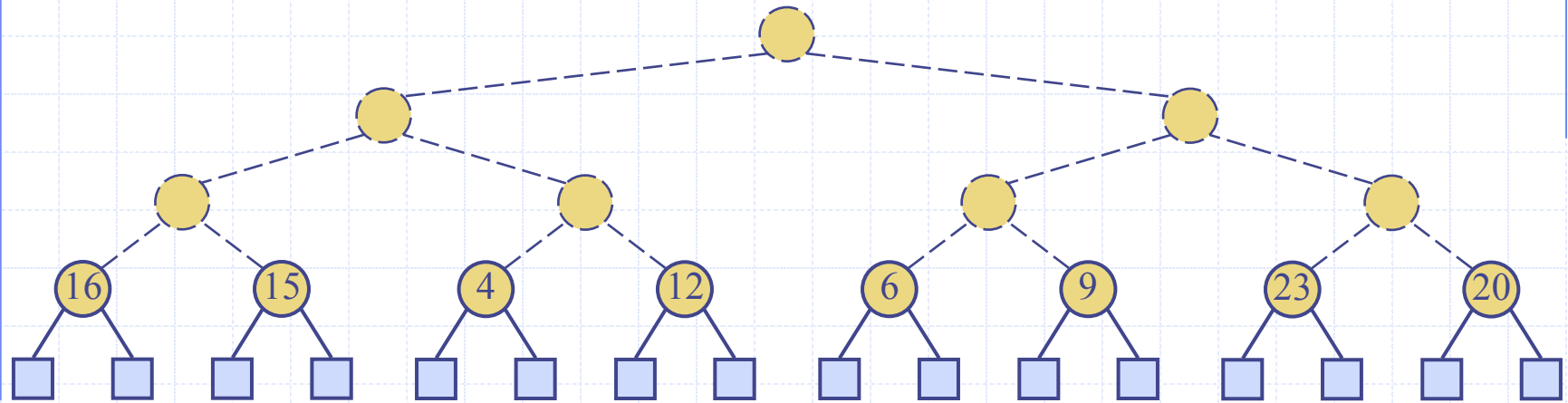
Bottom-up Heap Construction (§2.4.3)



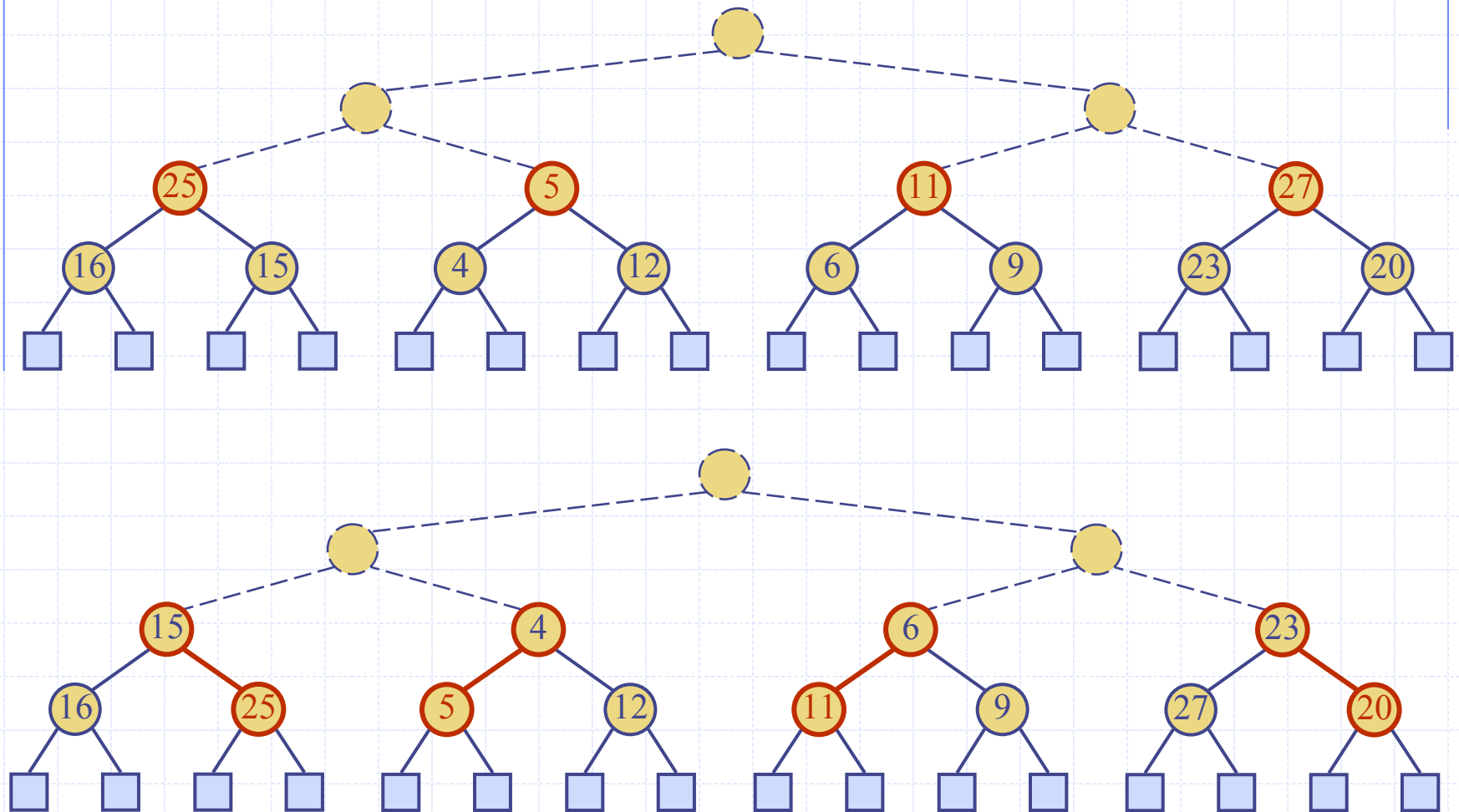
- ◆ We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- ◆ In phase i , pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys



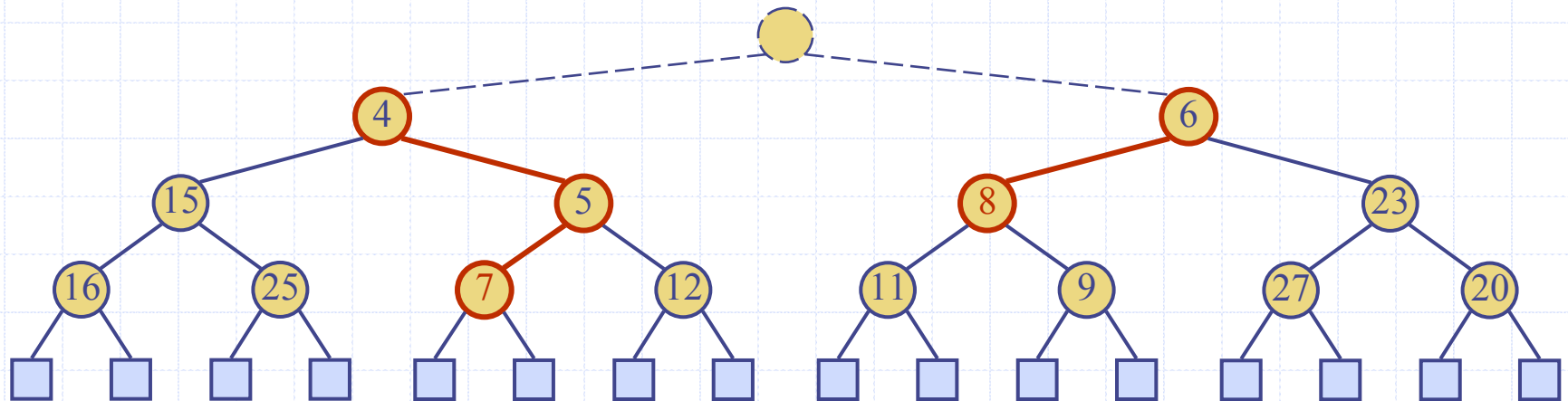
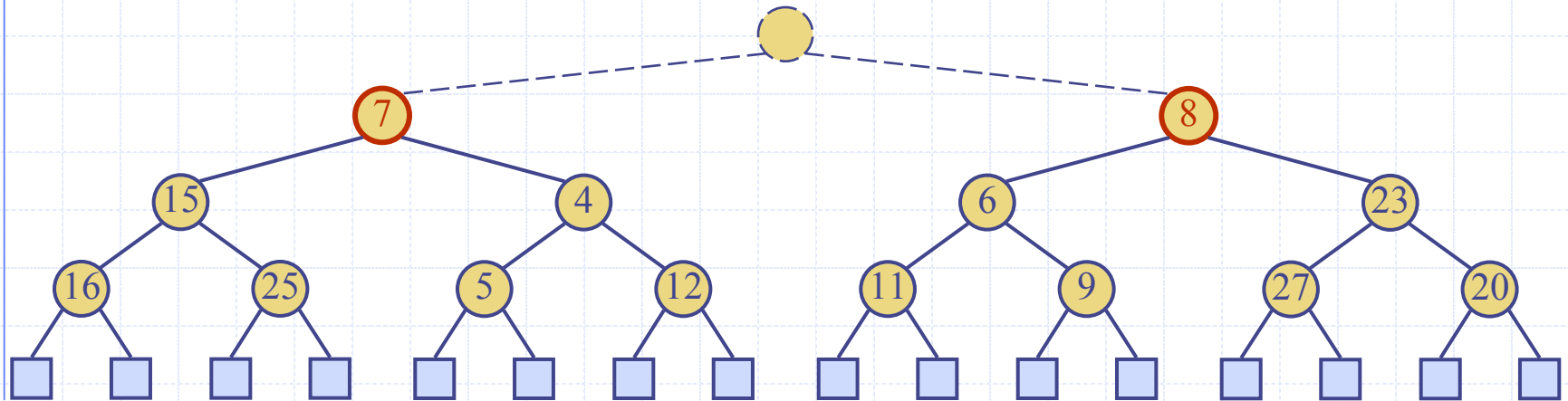
Example



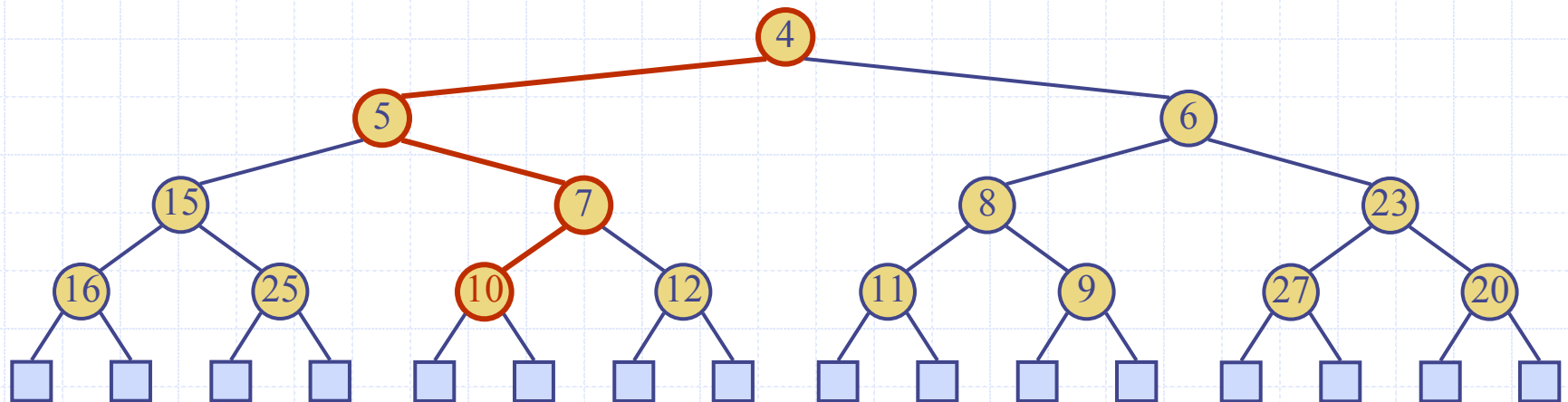
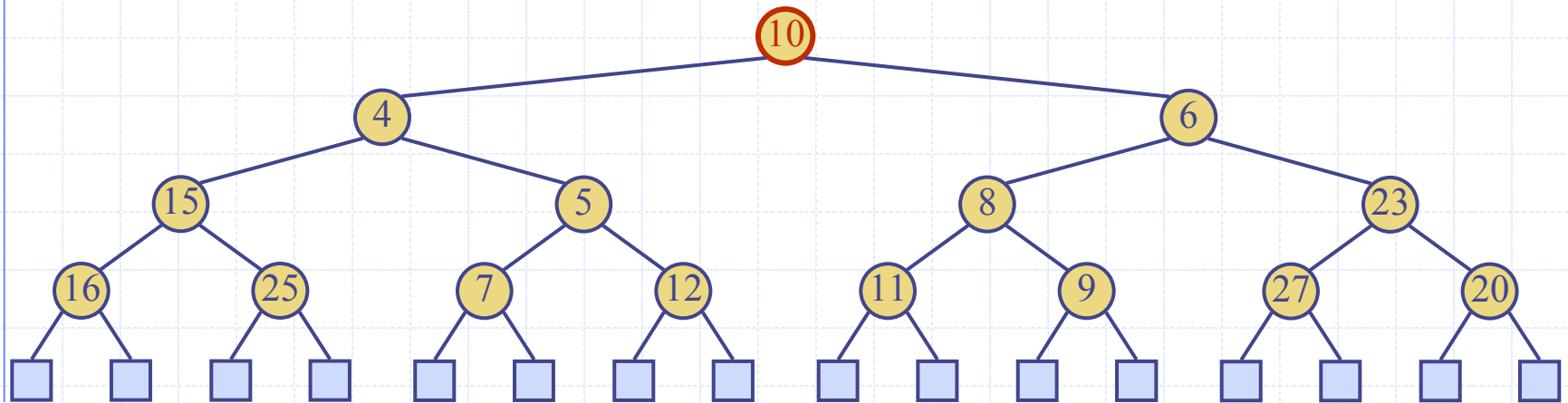
Example (contd.)



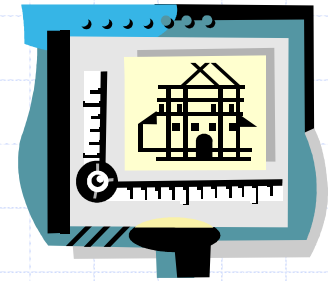
Example (contd.)



Example (end)



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

