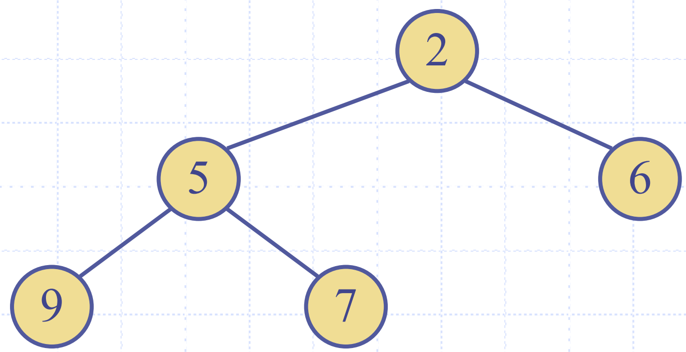
Heaps



Recall Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a pair (key, value)
- Main methods of the Priority Queue ADT
  - add(k, x)
    inserts an item with key k and value x
  - remove_min()
    removes and returns the item with smallest key
- Additional methods
  - min()
    returns, but does not remove, an item with smallest key
  - len(), is_empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Recall PQ Sorting

- We use a priority queue
  - Insert the elements with a series of add operations
  - Remove the elements in sorted order with a series of remove_min operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- Can we do better?

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 = priority queue with comparator ***C***

While not ***S.is_empty*** ()

  e = ***S.remove*** (***S. first*** ())

  P.add(***e***, ***e***)

While not ***P.is_empty***()

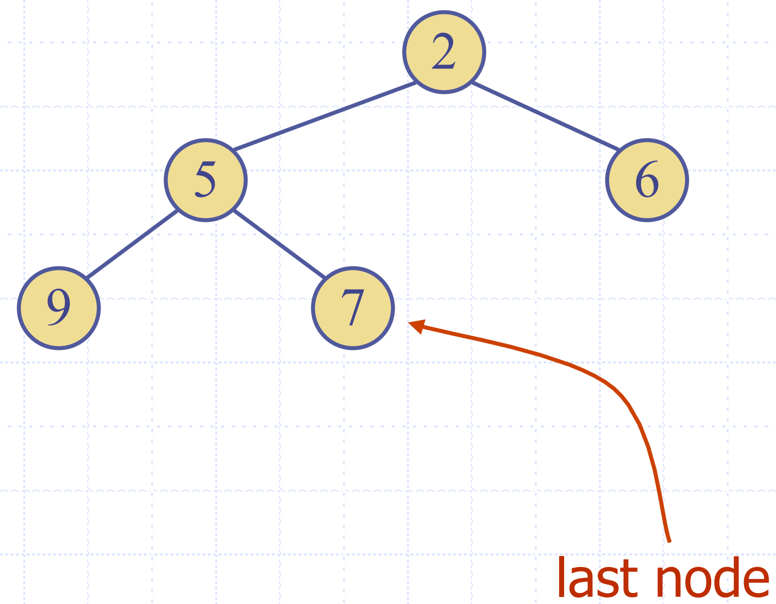
  e = ***P.remove_min***().***key***()

  S.add_last(***e***)

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, $\boldsymbol{key}(\boldsymbol{v}) \le \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
- Complete Binary Tree: let $\boldsymbol{h}$ be the height of the heap
  - for $\boldsymbol{i} = 0, \dots, \boldsymbol{h} - 1$, there are 2^{i} nodes of depth $\boldsymbol{i}$
  - at depth $\boldsymbol{h} - 1$, the internal nodes are to the left of the external nodes

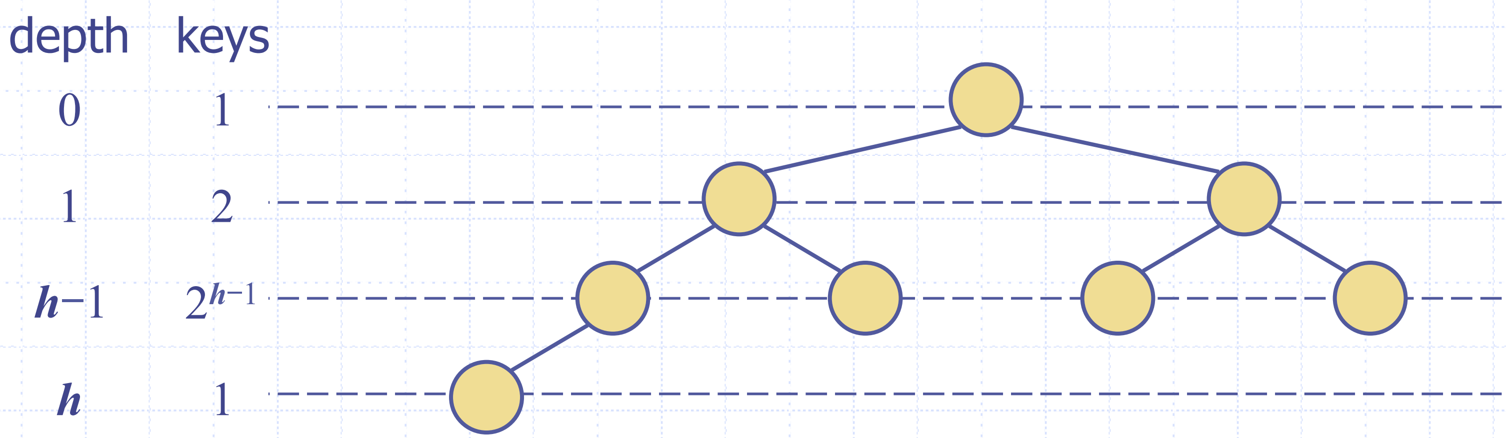
- The last node of a heap is the rightmost node of maximum depth



Height of a Heap

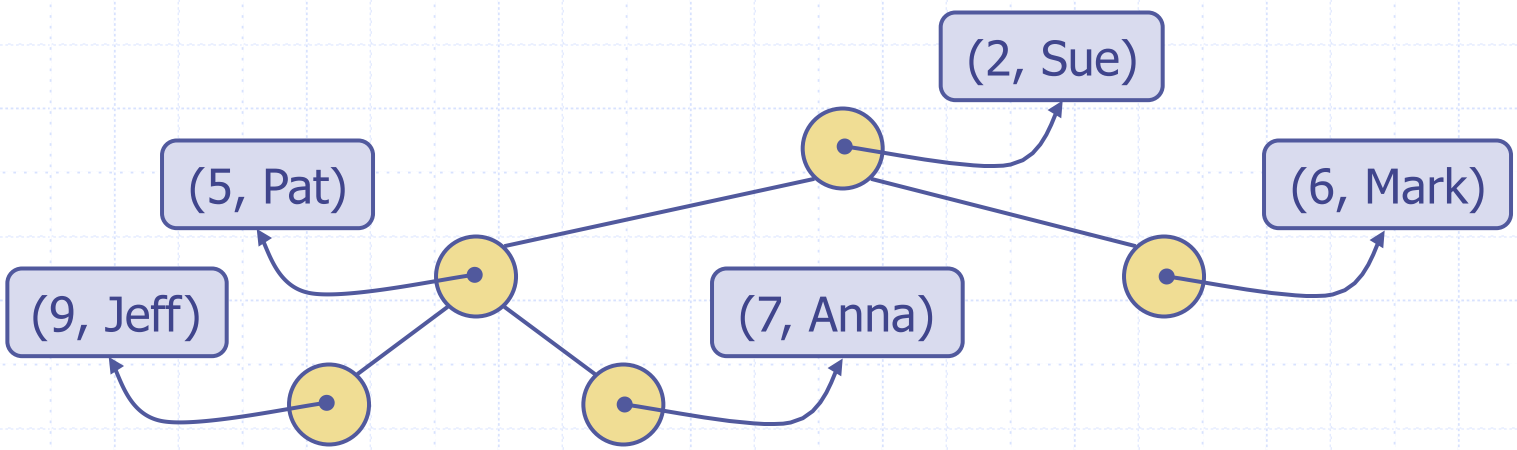
- **Theorem:** A heap storing n keys has height $O(\log n)$

  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 $i = 0, \ldots, h-1$ and at least one key at depth h, we have $1 + 2 + 4 + \ldots + 2^{h-1} + 1 <= n <= 2^h - 1$
  - Thus, $n < 2^h$, i.e., $h =$ floor($\log n$)



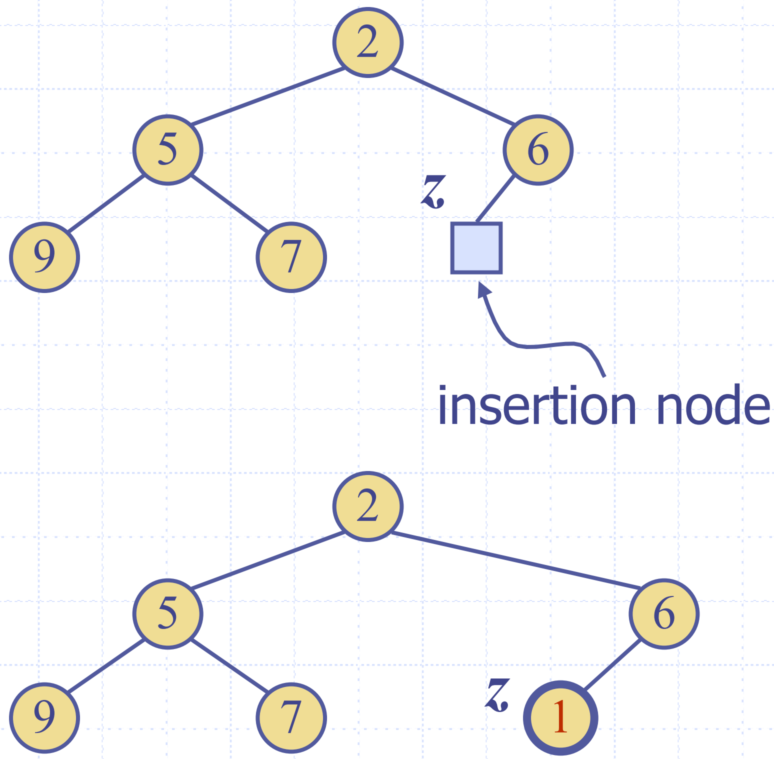
Heaps and Priority Queues

- 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



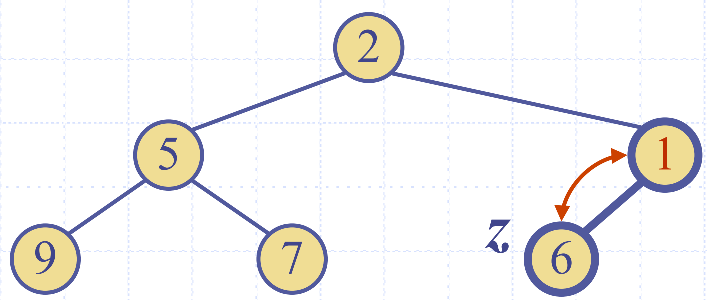
Insertion into a Heap

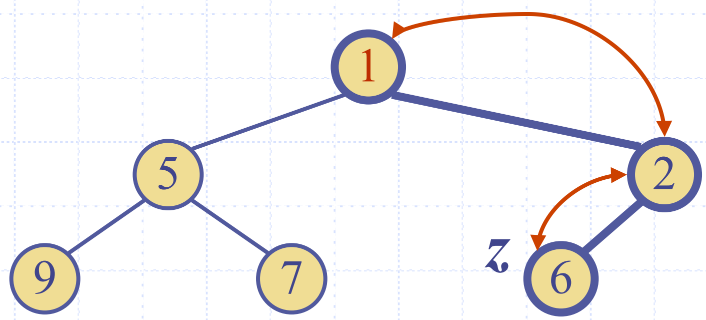
- Method add of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



Upheap (also called swim)

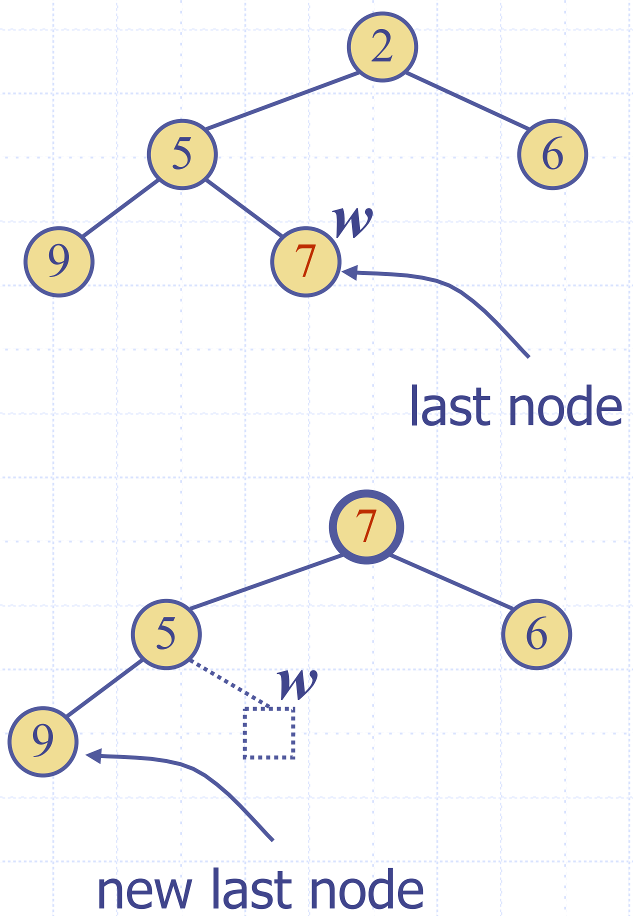
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k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





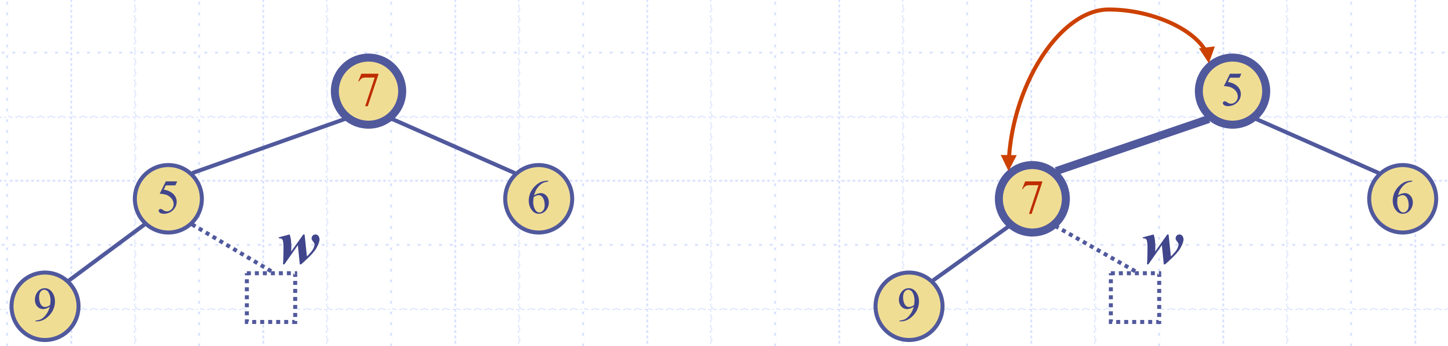
Removal from a Heap

- Method remove_min 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
  - Remove w
  - Restore the heap-order property (discussed next)



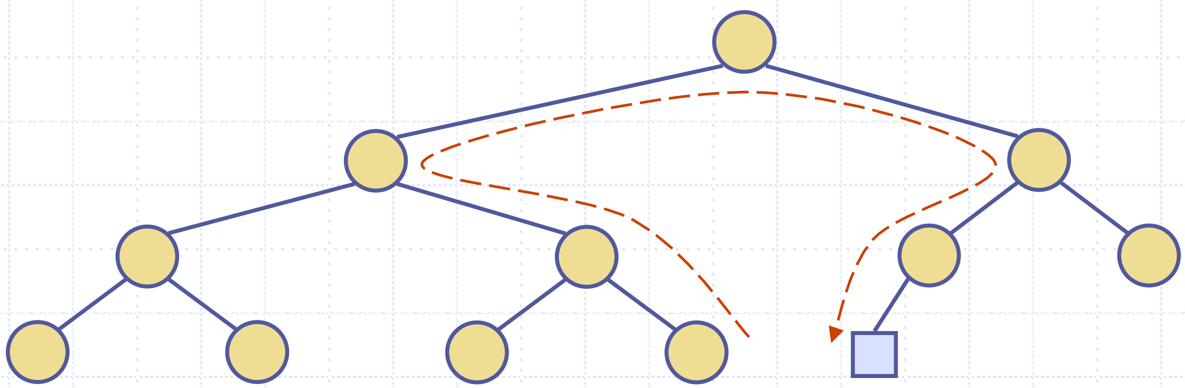
Downheap (also called sink)

- 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

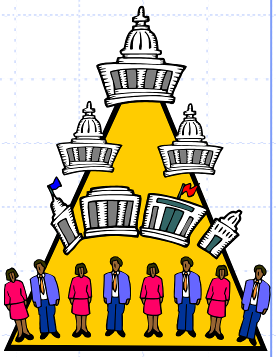


Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



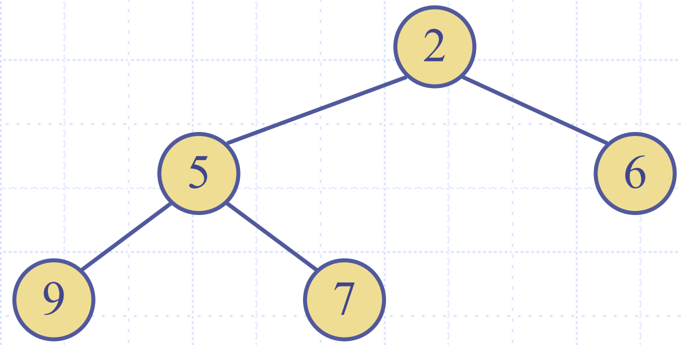
Heap-Sort



- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods add and remove_min take $O(\log n)$ time
  - methods len, is_empty, and min 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

Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank $n + 1$
- Operation remove_min corresponds to removing at rank n
- Yields in-place heap-sort



2	5	6	9	7
0	1	2	3	4

Python Heap Implementation

```
#ArrayHeapPriorityQueue
class ArrayHeapPriorityQueue( PriorityQueue ):

    def __init__( self ):
        self._Q = []

    def __len__( self ):
      return len( self._Q )

    def __getitem__( self, i ):
        return self._Q[i]

    def is_empty( self ):
        return len( self ) == 0
```

Python Heap...

```
def _parent( self, j ):
     return (j-1) // 2

def _left( self, j ):
     return 2*j + 1

def _right( self, j ):
     return 2*j + 2

def _has_left( self, j ):
     return self._left( j ) < len( self )

def _has_right( self, j ):
     return self._right( j ) < len( self )

def min( self ):
    if self.is_empty():
        return False
    #min is in the root
    return self._Q[0]
```

Python Heap…

```
def add( self, k, x ):
    #in O(log n)
    item = self._Item( k, x )
    self._Q.append( item )
    #swim the new item in O(log n)
    self._swim( len(self)-1 )
    #return the new item
    return item

def remove_min( self ):
    if self.is_empty():
        return False
    #min is at the root
    the_min = self._Q[0]
    #move the last item to the root
    self._Q[0] = self._Q[len(self)-1]
    #delete the last item
    del self._Q[len(self)-1]
    if self.is_empty():
        return the_min
    #sink the new root in O(log n)
    self._sink( 0 )
    #return the min
    return the_min
```

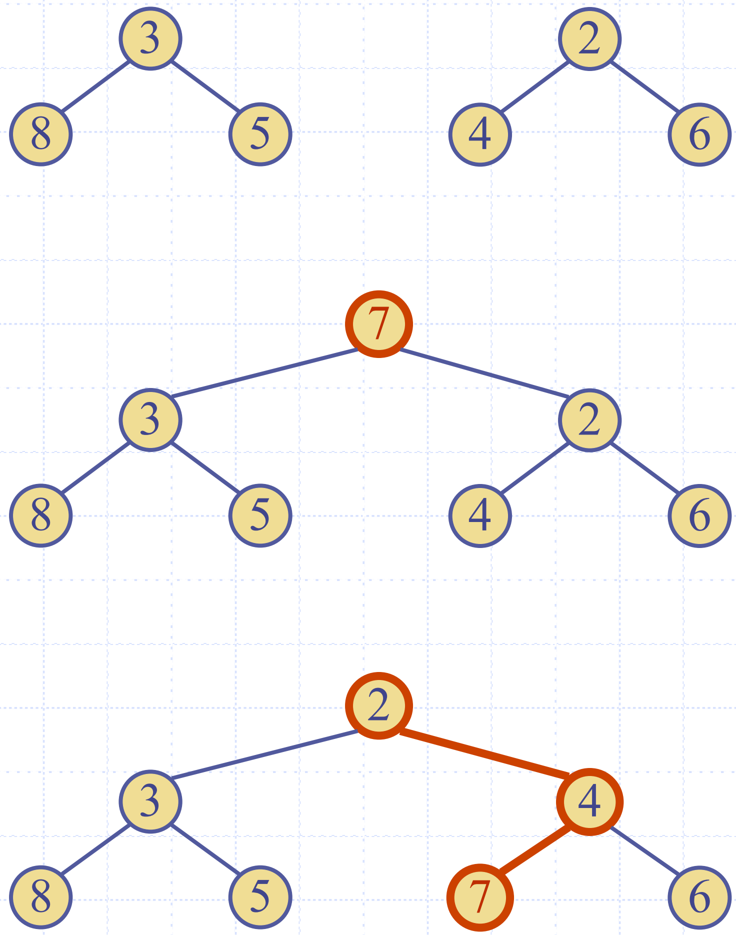
Python Heap

```
def _swim( self, j ):
    parent = self._parent( j )
    if j > 0 and self._Q[j] < self._Q[parent]:
        self._swap( j, parent )
        self._swim( parent )

def _sink( self, j ):
    if self._has_left( j ):
        left = self._left( j )
        small_child = left
        if self._has_right( j ):
            right = self._right( j )
            if self._Q[right] < self._Q[left]:
                small_child = right
        if self._Q[small_child] < self._Q[j]:
            self._swap( j, small_child )
            self._sink( small_child )
```

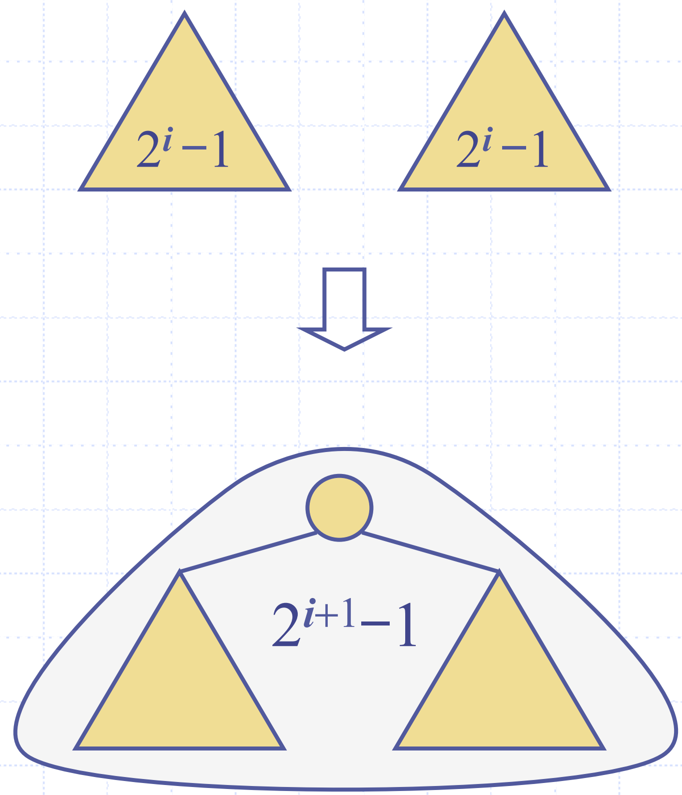
Merging Two Heaps

- 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

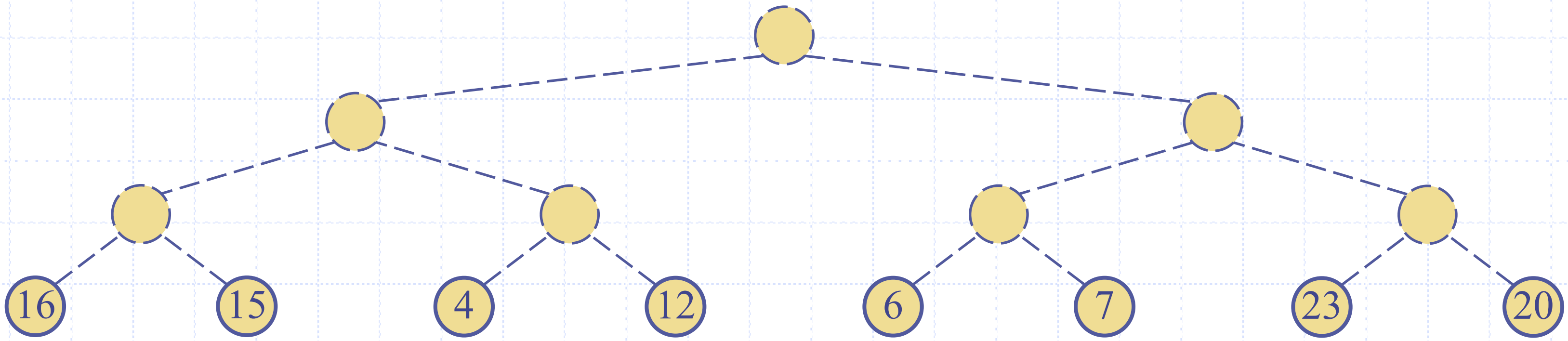


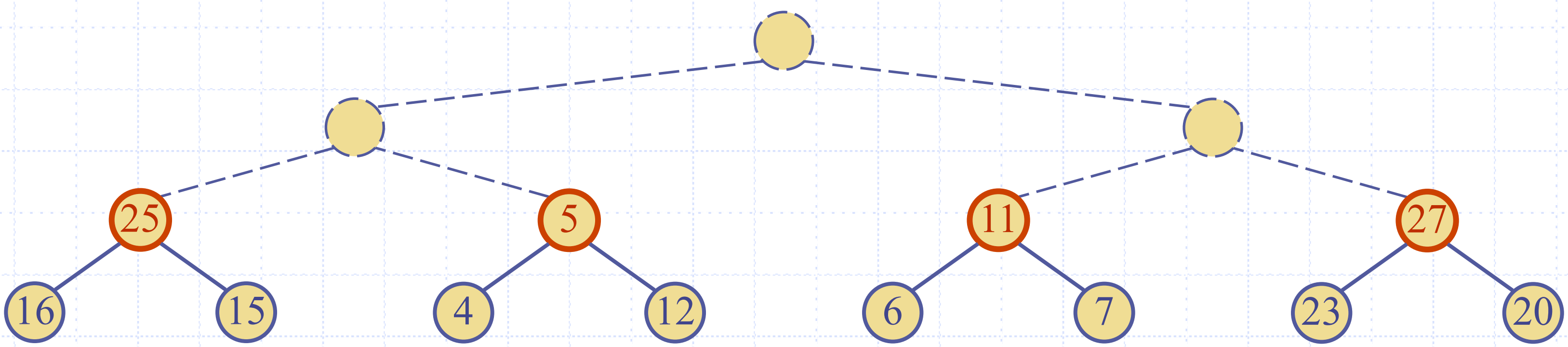
Bottom-up Heap Construction

- 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

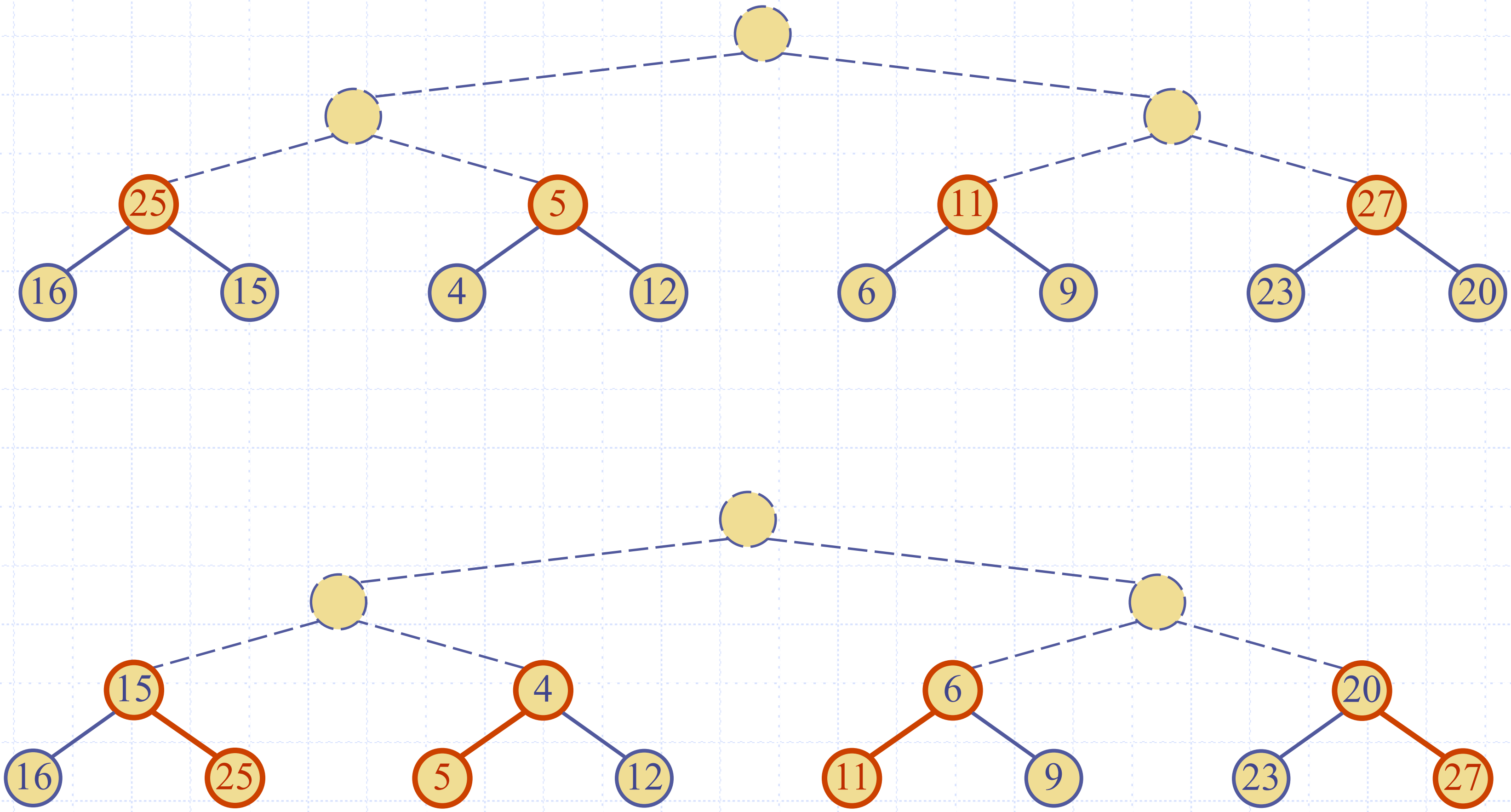


Example

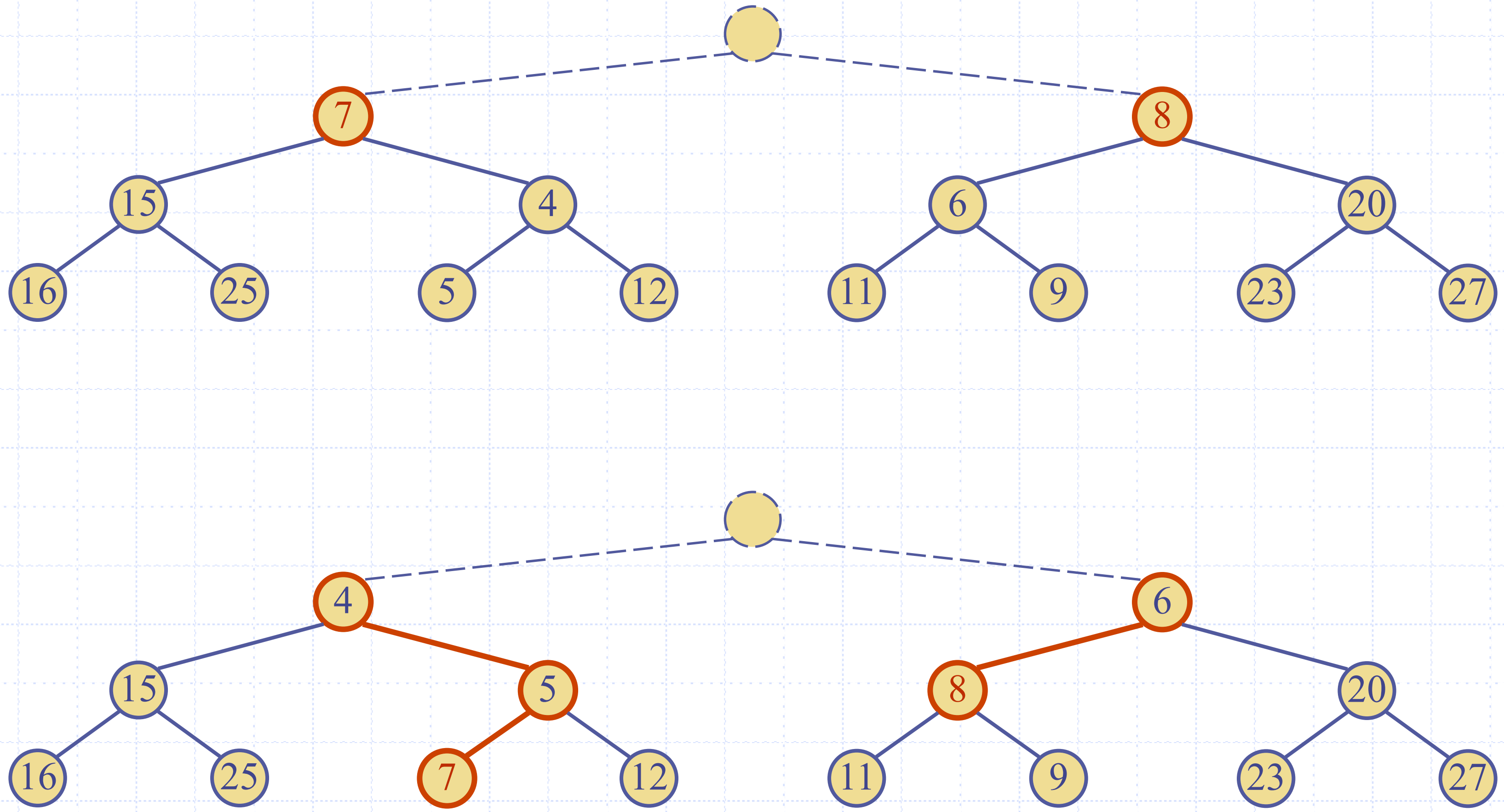




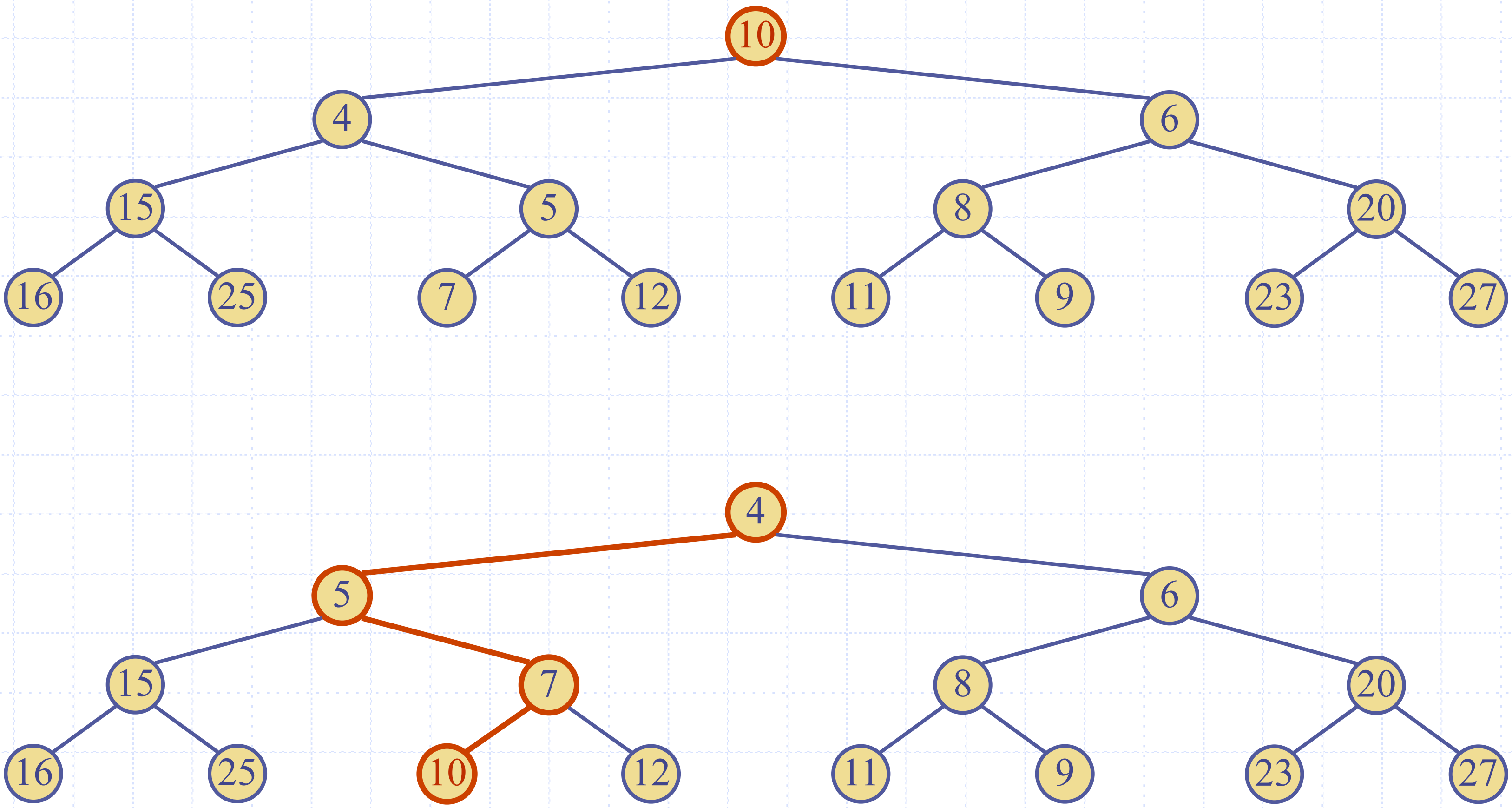
Example (contd.)



Example (contd.)



Example (end)



Analysis of Heap Construction

- 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