CS 2200: Algorithms
Heaps and Priority Queues

Bowdoin College
Fall, 2015

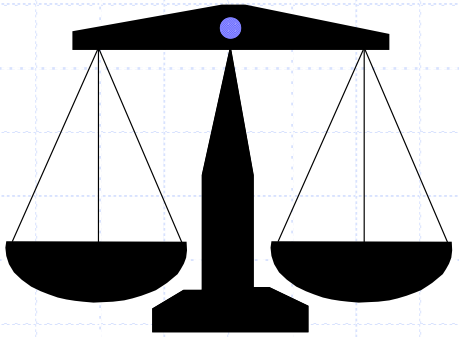
Queues and Priority Queues

- What's a queue?
- What's a priority queue?
  - Completely ordered?

Priority Queue ADT

- 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 $\leq$
  - **Totality** property:
    $x \leq y \vee y \leq x$
  - **Antisymmetric** property:
    $x \leq y \wedge y \leq x \Rightarrow x = y$
  - **Transitive** property:
    $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Sorting with a Priority Queue

- 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 = priority queue with comparator *C*
  while (!*S.isEmpty*)
    e = *S.remove* (*S. first* ())
    P.insertItem(*e*, *e*)
  while (!*P.isEmpty*)
    e = *P.removeMin*()
    S.insertLast(*e*)

Sequence-Based Priority Queue

- PQ implementation with an unsorted linked 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

- PQ implementation with a sorted linked list

- Performance:
  - insertItem takes $O(n)$ time for both since we have to find the right place 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

$$1 + 2 + \ldots + n$$

- 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

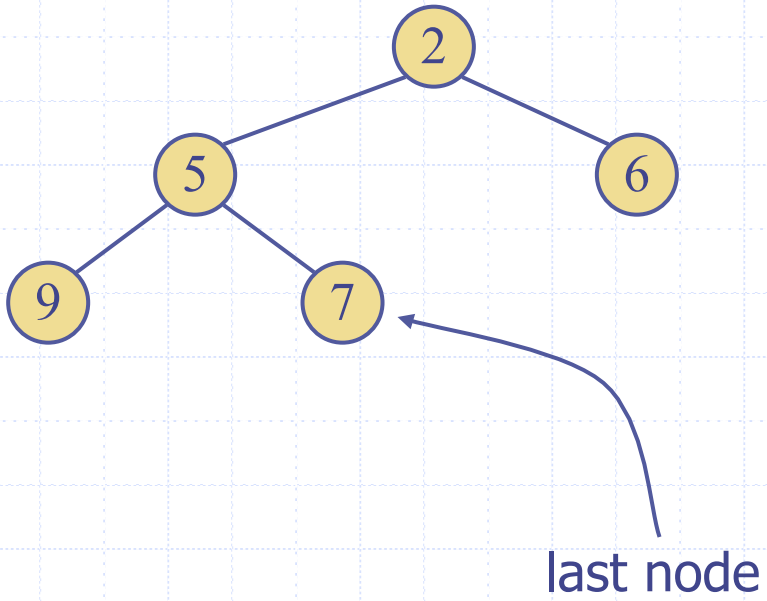
- Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n insertItem operations takes time proportional to

$$1 + 2 + \ldots + n$$

  - 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

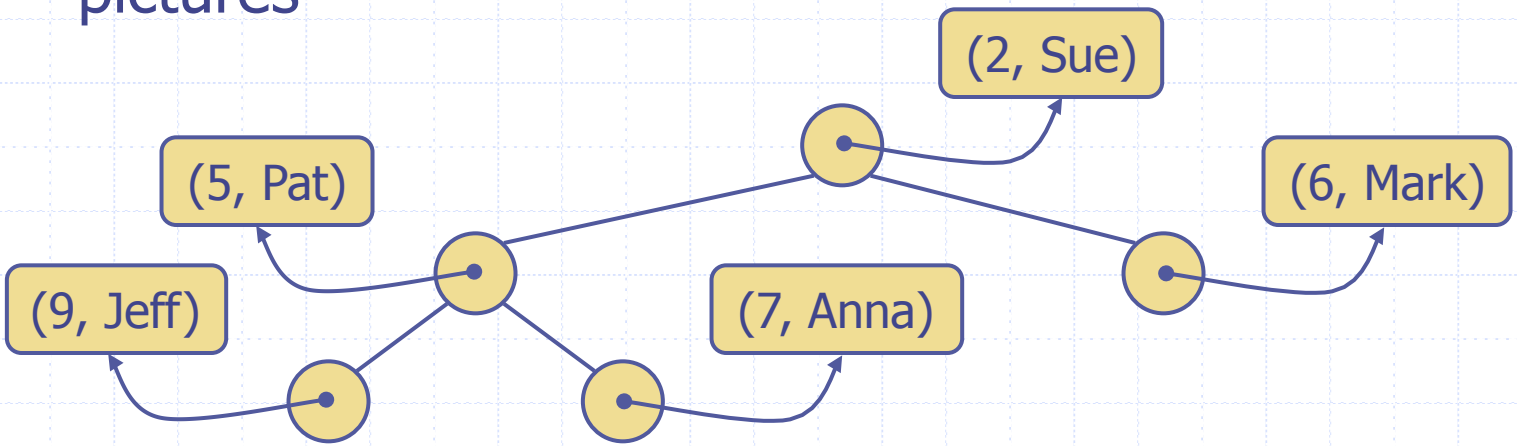
Heaps

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap-Order: for every 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
    - the last node of the heap is the rightmost node at depth h



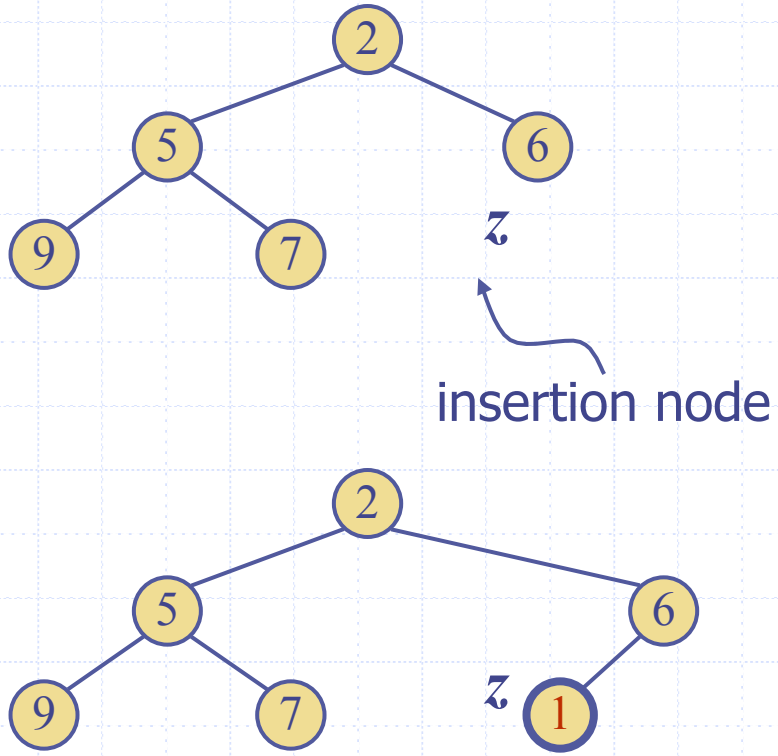
Heaps and Priority Queues

- 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 subsequent pictures



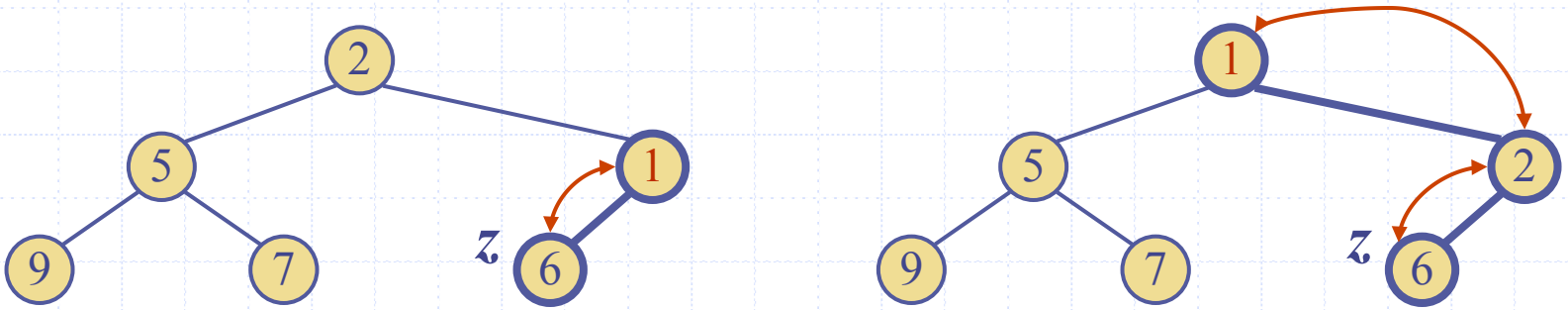
Insertion into Heap

- 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
  - Restore the heap-order property (discussed next)



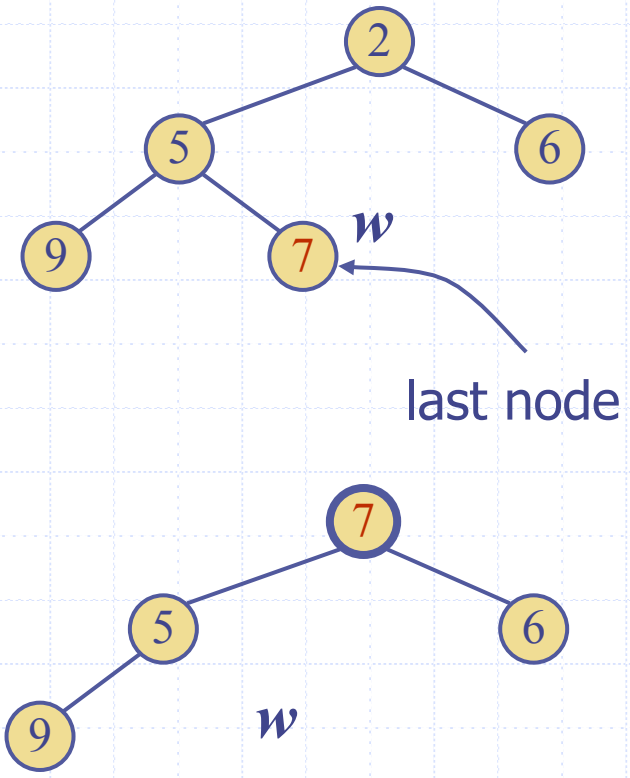
Upheap

- 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



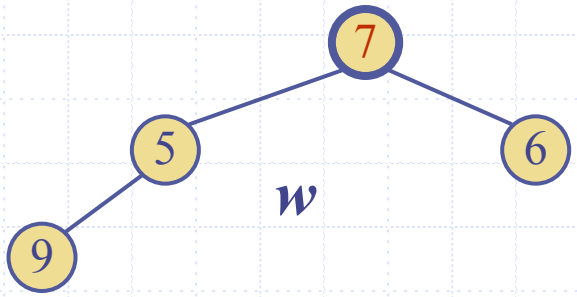
Removal from a Heap

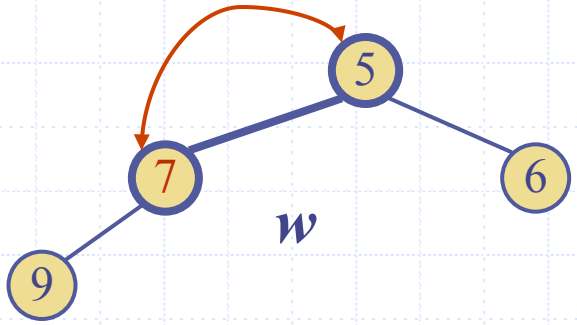
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps:
  - Remove the root key
  - Make the key of the last node w the root key
  - Restore the heap-order property (discussed next)



Downheap

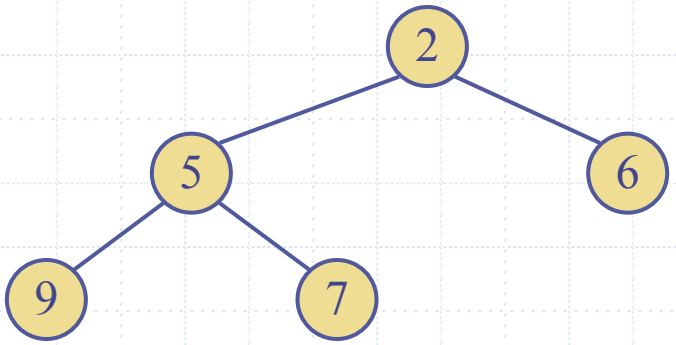
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k with one of its children along a downward path from the root
- Downheap terminates when key k reaches a node whose children have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

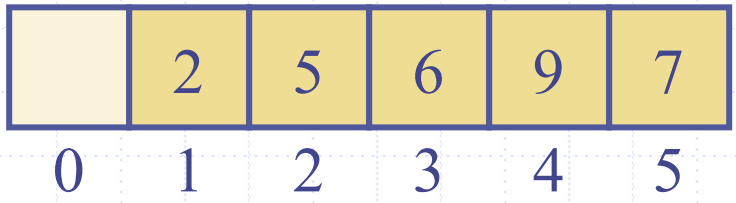




Vector-Based Heap Implementation

- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at index i
  - the left child is at index $2i$
  - the right child is at index $2i + 1$
  - the parent is at index floor(i/2)
- Links between nodes are not explicitly stored
- The cell at index 0 is not used
- Operation insertItem corresponds to inserting at index $n + 1$ and upheaping
- Operation removeMin corresponds to removing at index n and downheaping





Heap-Sort

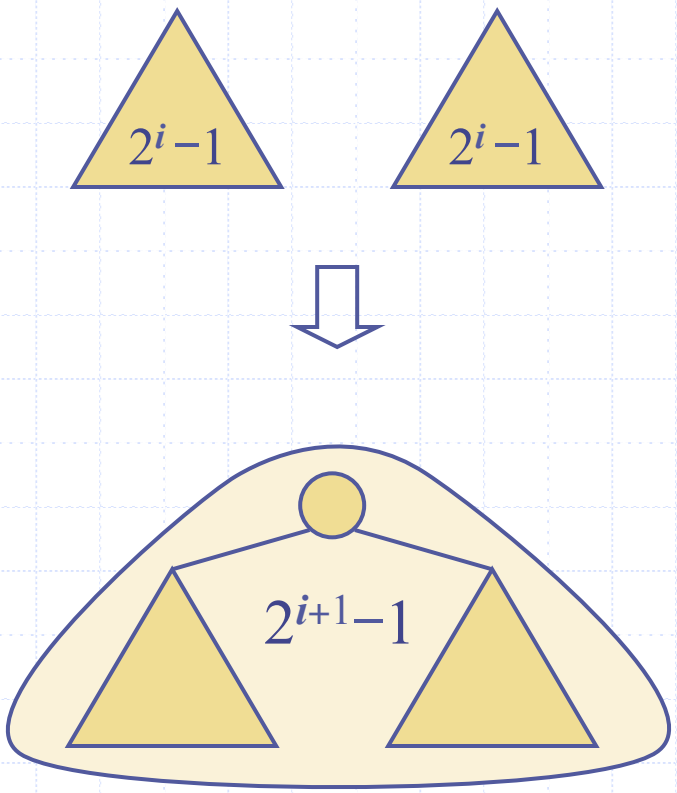
- Make a heap-based priority queue from the n keys to sort
- Remove the n keys from the heap one at a time

Top-Down Heap Construction

- Insert n keys one at a time
- How long does it take to insert key i? $\lg i$
- So: $\sum_{i=0}^{n} \lg i \approx n \lg n$

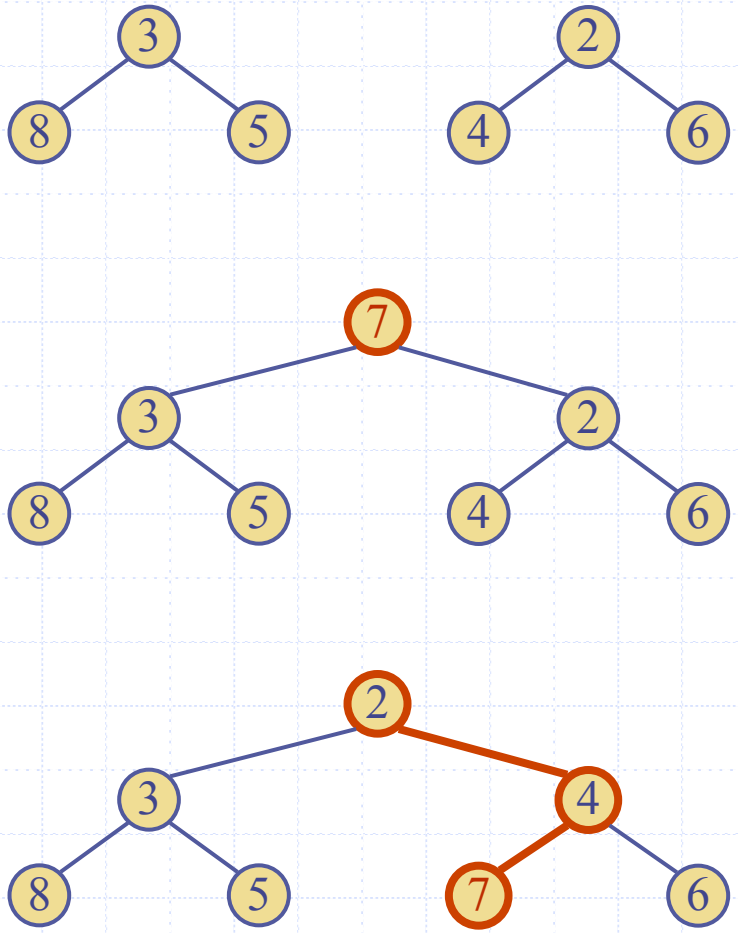
Bottom-Up Heap Construction

- 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

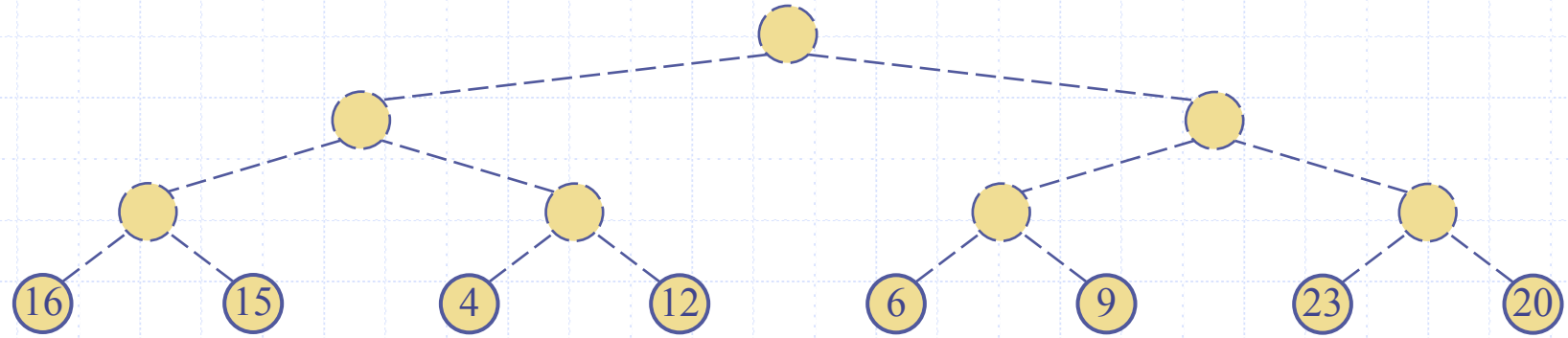


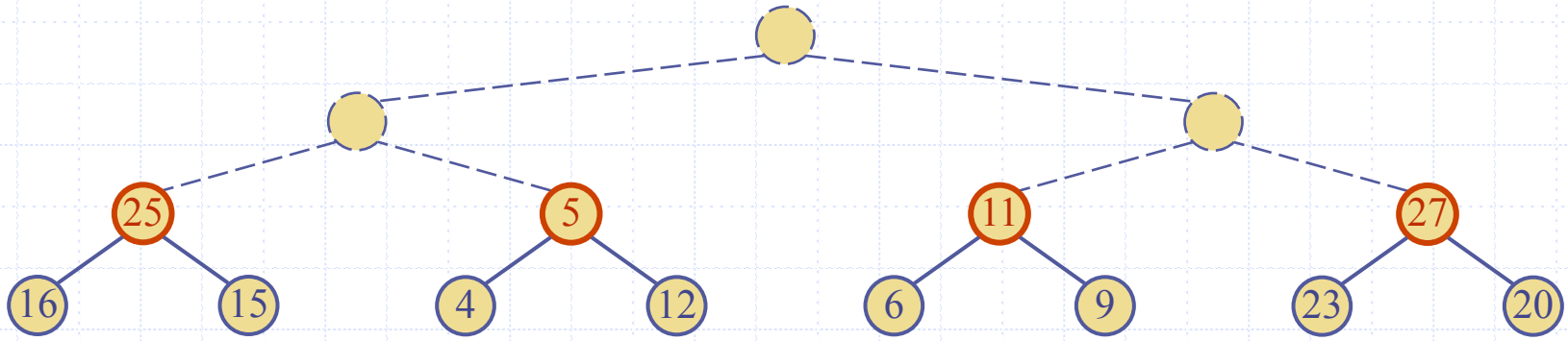
Merging Two Heaps

- 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

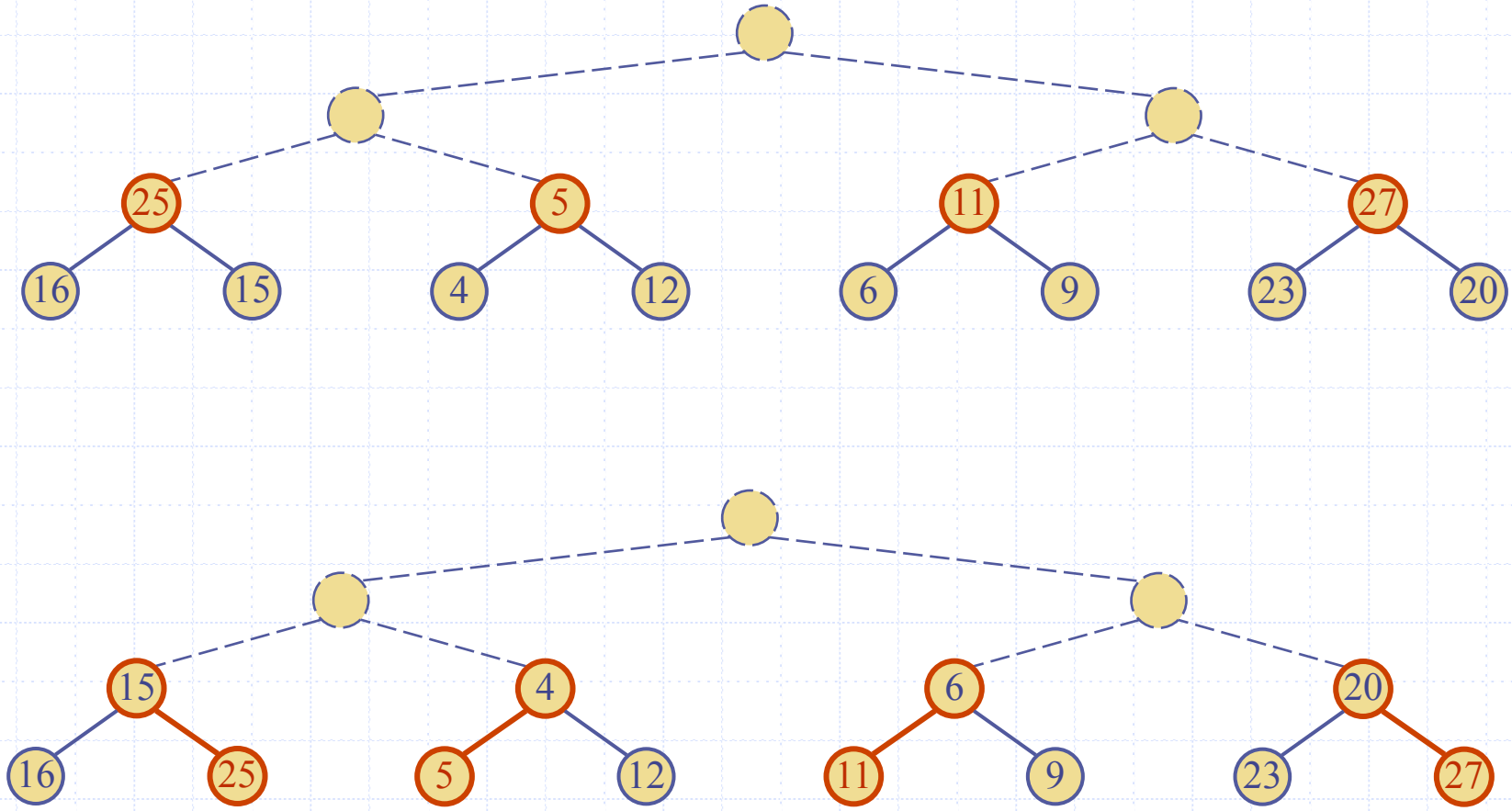


Example

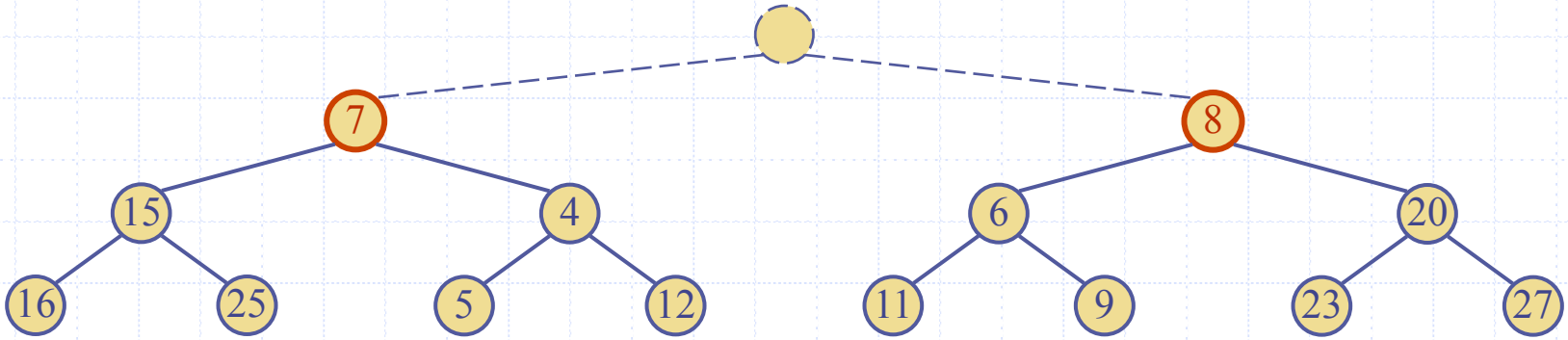


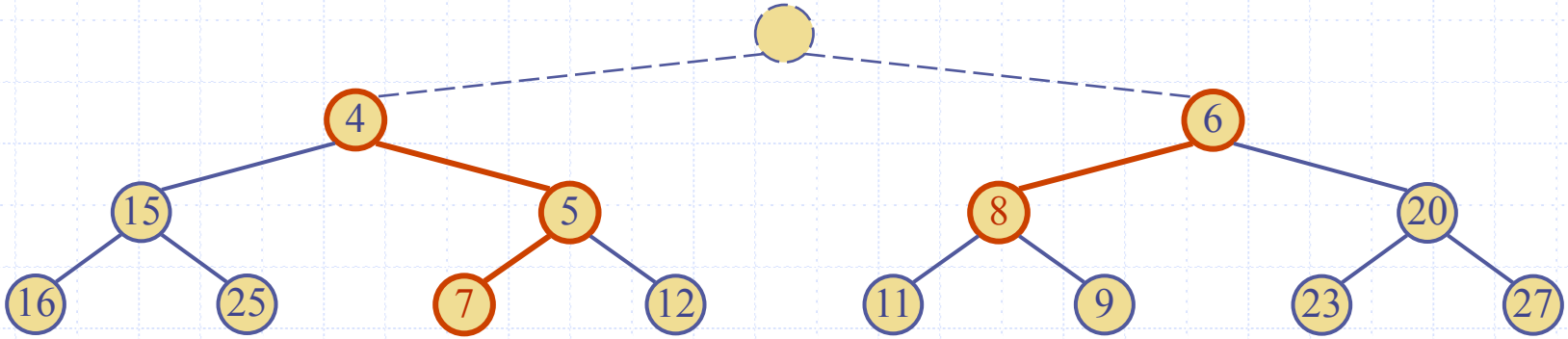


Example (contd.)

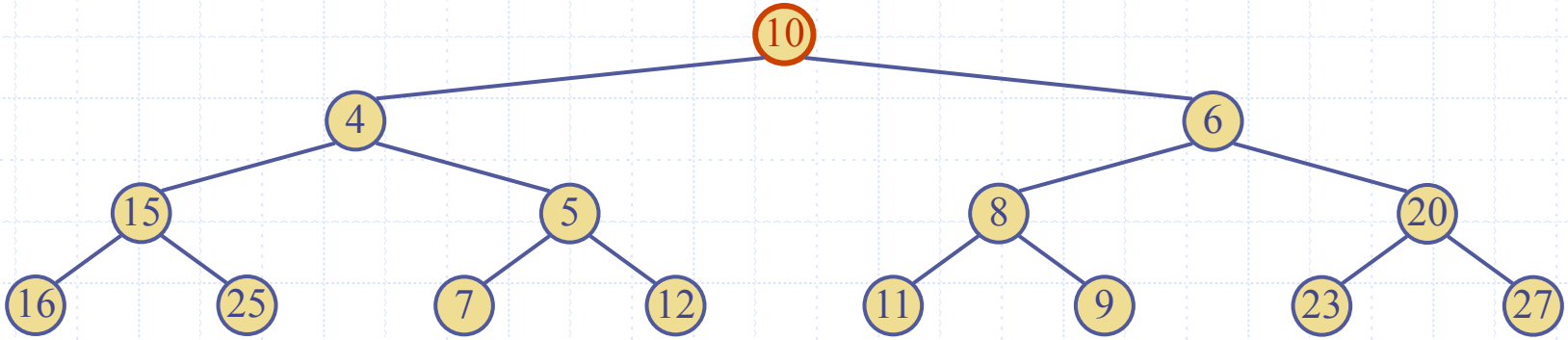


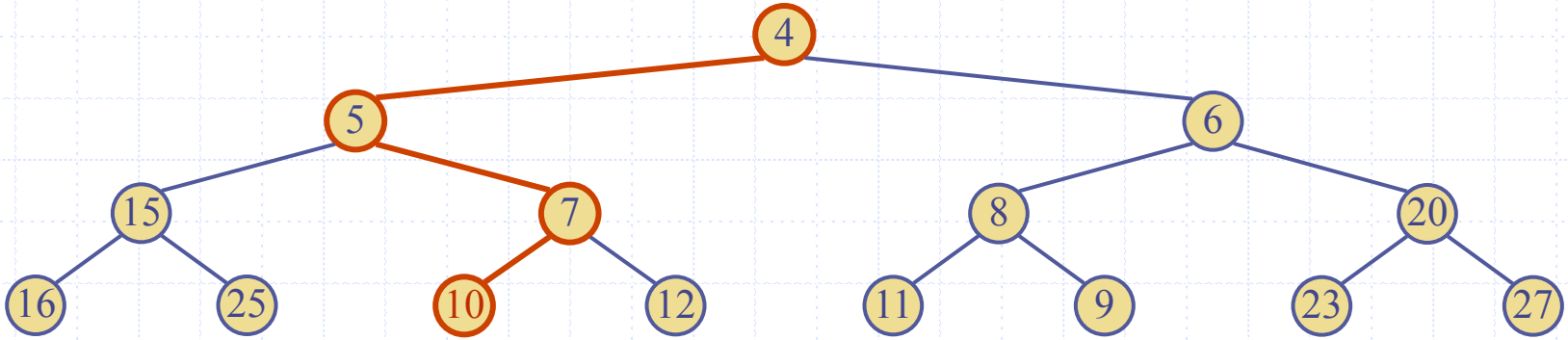
Example (contd.)





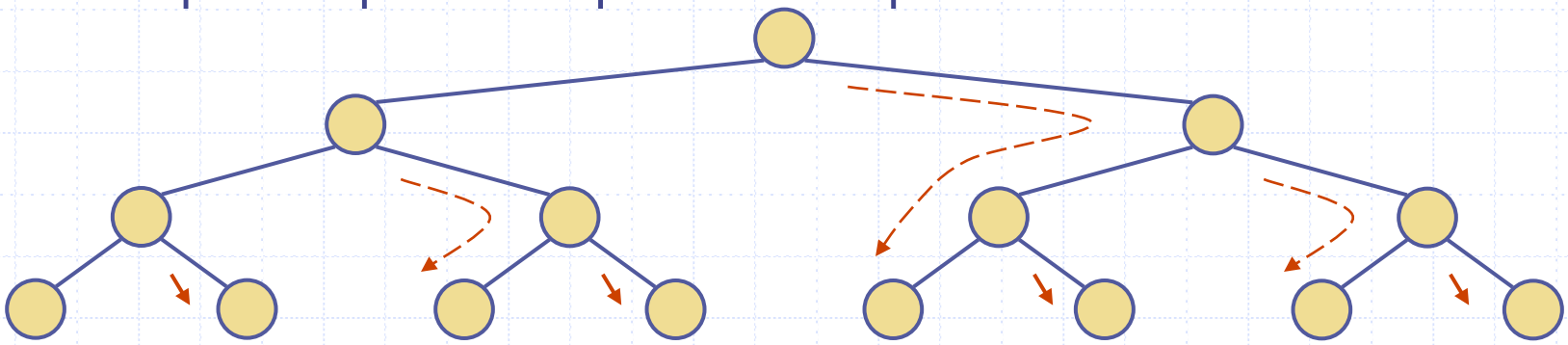
Example (end)





Visual 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 no edge is traversed more than once, 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



Non-Visual Analysis

- During phase i there are $n/2^{i+1}$ pairs of heaps being combined
- Each pair has a downheap path length of i
- So length of all downheap paths during phase i is $n(i/2^{i+1}) < n(i/2^i)$
- Sum these over the $\lg n$ phases:

$$\sum_{i=1}^{\lg n} n\frac{i}{2^i} = n\sum_{i=1}^{\lg n}\frac{i}{2^i} = n\sum_{i=1}^{\lg n} i\left(\tfrac{1}{2}\right)^i$$

- And use this fact:

$$\sum_{i=0}^{\infty} ix^i = \frac{x}{(1-x)^2}, \quad |x| < 1$$

- So:

$$n\sum_{i=1}^{\lg n} i\left(\tfrac{1}{2}\right)^i < n\sum_{i=0}^{\infty} i\left(\tfrac{1}{2}\right)^i = 2n$$

- And bottom-up heap construction is $O(n)$ 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
  - 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