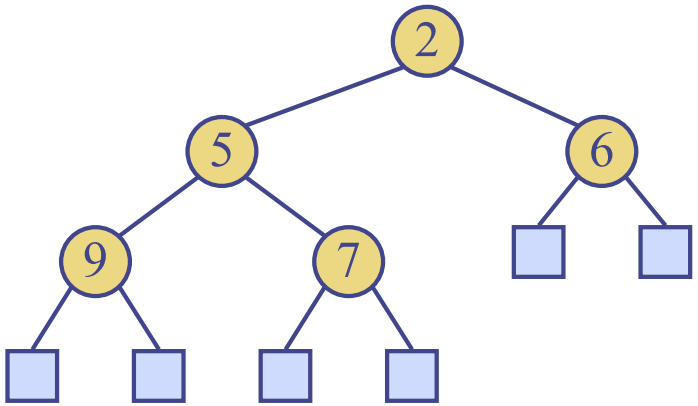
Heaps and HeapSort



Priority Queue Abstract Data Type (ADT)

- A priority queue stores a collection of items, each of which has a given priority.
- An item is a pair (key, element), where the key is the priority. Assume smaller key = higher priority.
- Main methods of the Priority Queue ADT
  - insertItem(k, e)
    inserts an item with key k and element e
  - 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

Sorting with Priority Queues

- We can use a priority queue to sort a set of elements
- How?

Sorting with Priority Queues

- We can use a priority queue to sort a set of 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
- What are some possible implementations?

```
PQ-Sort(S)
    P = priority queue
    while S.isEmpty == false
        e = S. get(0)
        S.remove(0)
        P.insertItem(e, e)
    while P.isEmpty() == false
        e = P.removeMin()
        S.add(e)
```

List-Based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insertItem takes $O(1)$ time since we can insert the item at the end of the list.
  - removeMin, minKey and minElement take $O(n)$ time since we may have to traverse the entire list to find the smallest key.

- Implementation with a sorted list

- Performance:
  - insertItem takes $O(n)$ time since we have to find where to insert the item and possibly move existing items
  - The time for minKey and minElement is $O(1)$.
  - The time for removeMin depends on how the list is implemented, even thought the min is the first item in the list.
    - $O(n)$ if it's an array
    - $O(1)$ if it's a linked list

PQ-Sort with Unsorted List PQ

- Running time:
  - 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:

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

- $O(n^2)$ time
- (An aside: Phase 2 is like SelectionSort.)

PQ-Sort with Sorted List PQ

- Running time:
  - Inserting the elements into the priority queue with n insertItem operations takes time:

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

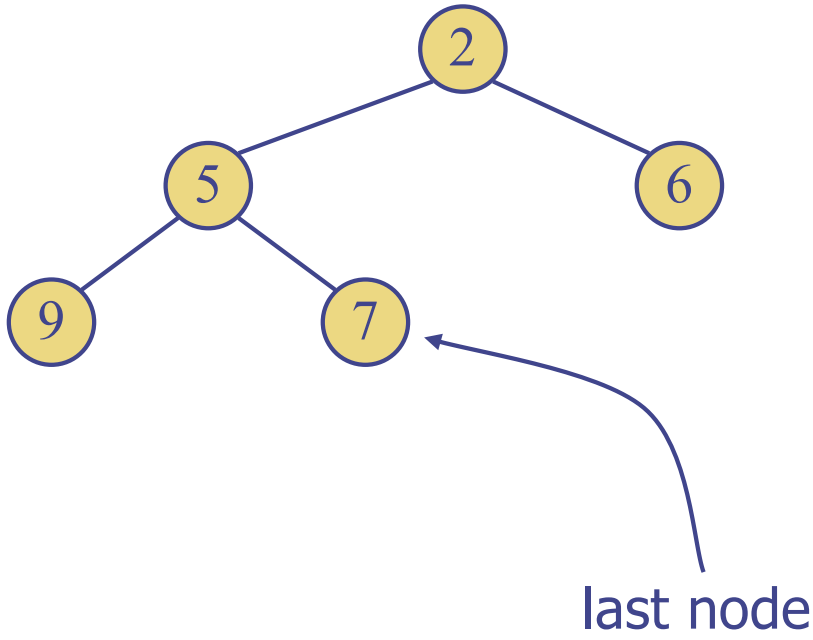
  - Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n^2)$ time if using an array, $O(n)$ time if using a linked list.
- In either case, $O(n^2)$ time.
- (An aside: Phase 1 is like Insertion Sort.)

Heaps

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
  - Heap Property for "Min Heap": for every node v other than the root:

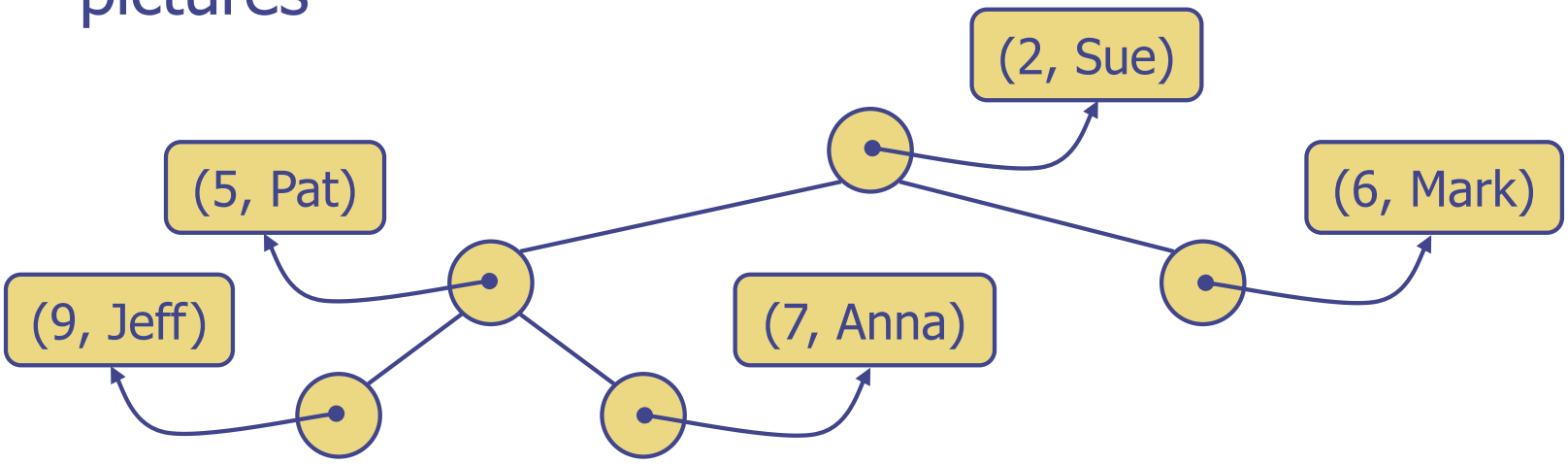
$$key(v) \geq key(parent(v))$$

  - Complete Binary Tree:
    - all levels are full except possibly the last one
    - all the nodes at the last level are as far left as possible
  - Important: A heap storing n keys has height $O(\log n)$



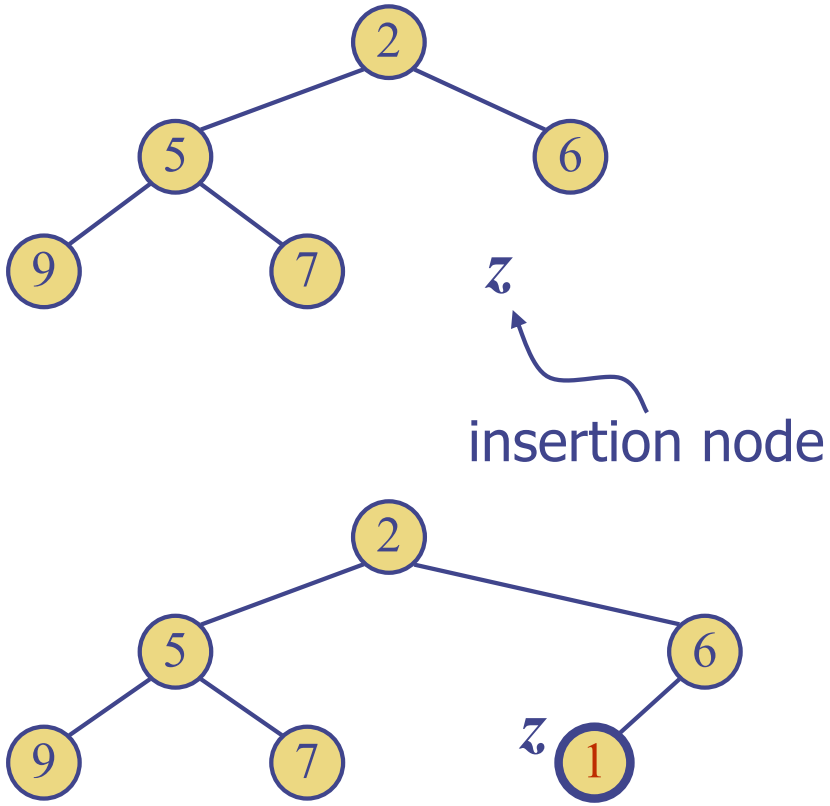
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a <key, value> pair at each node
- We keep track of the position of the last node
- For simplicity, we show only the keys in subsequent pictures



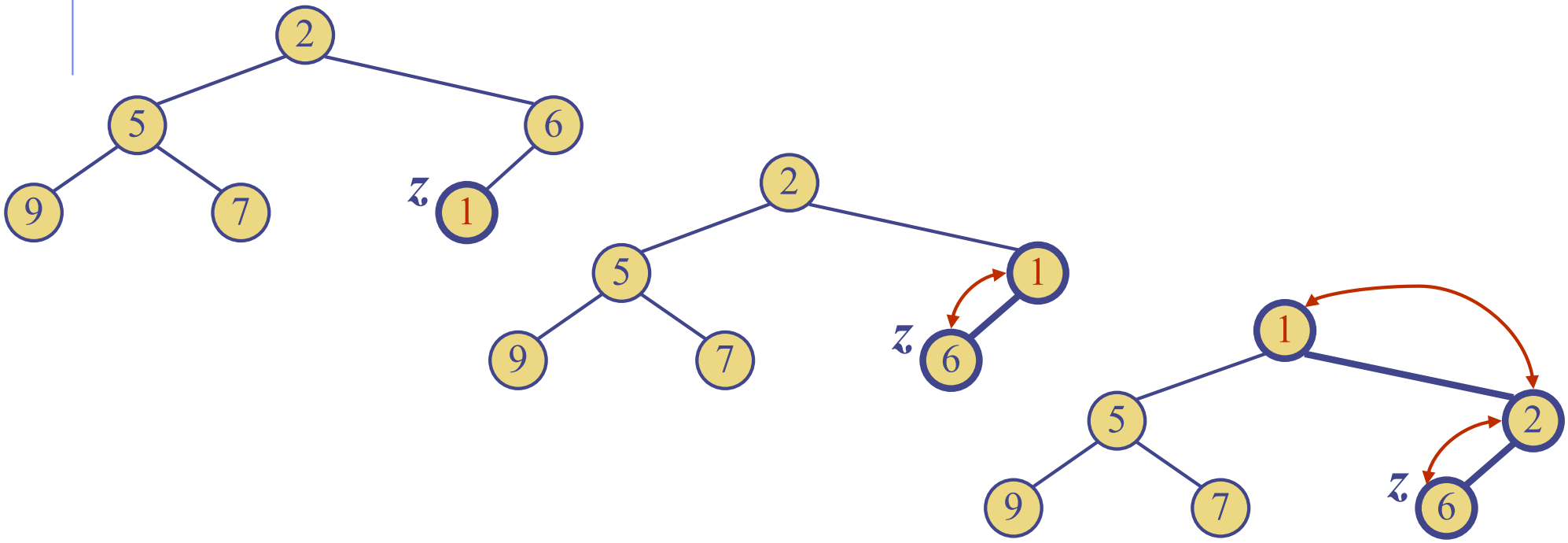
Insertion into Heap

- 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 point z (the new last node)
  - Store k at z
  - Restore the heap property (discussed next)



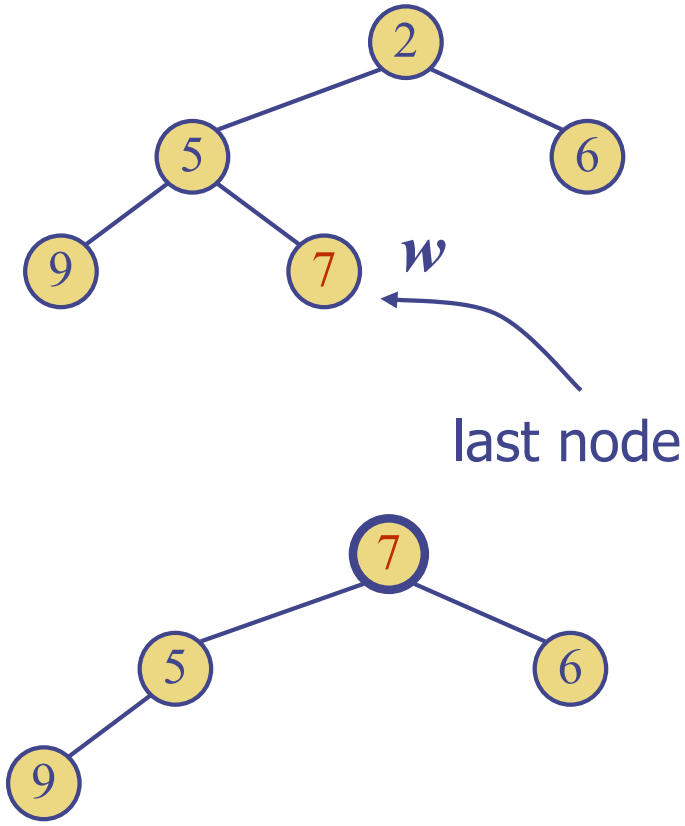
Upheap

- After the insertion of a new key k, the heap property may be violated
- Algorithm upheap restores the heap 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 less 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
  - Move the key of the last node w to the root
  - Restore the heap property (discussed next)

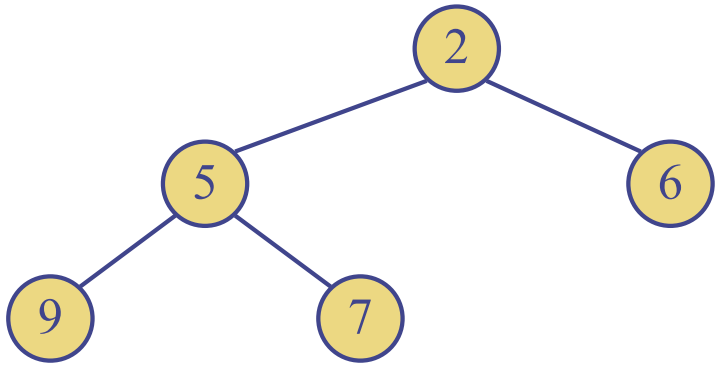


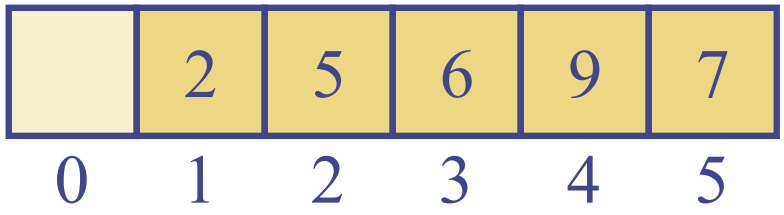
Downheap

- 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. Which one?
- 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

Heap Implementation Using ArrayLists or Arrays

- We can represent a heap with n keys by means of an array or ArrayList 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 moving the item at index n to index 1 and downheaping





HeapSort

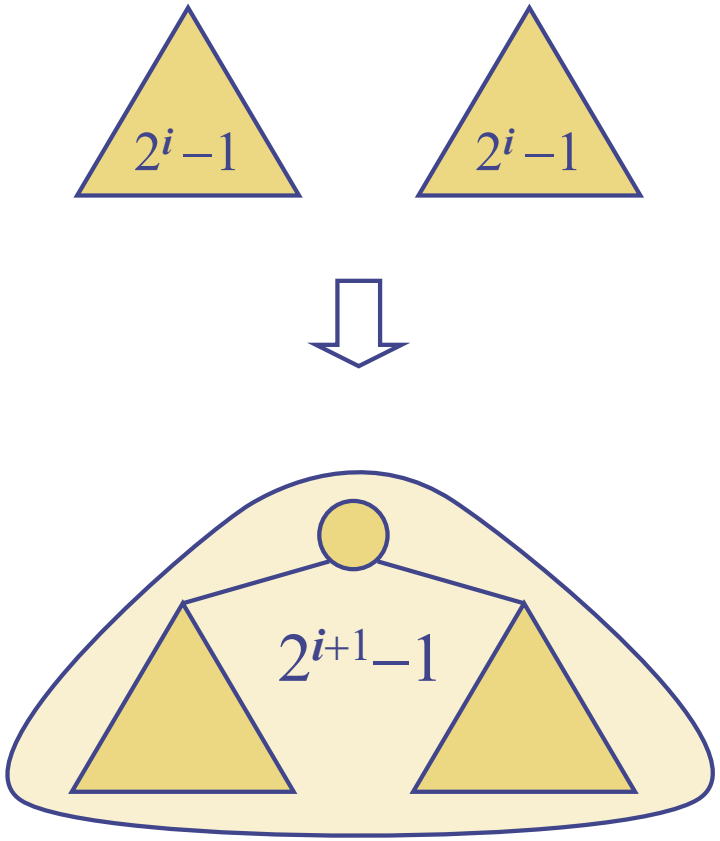
- Given a list S of n items (keys), remove the items in S and make a heap-based priority queue from the n keys
- Remove the n keys from the heap one at a time and add them to S

Top-Down Heap Construction

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

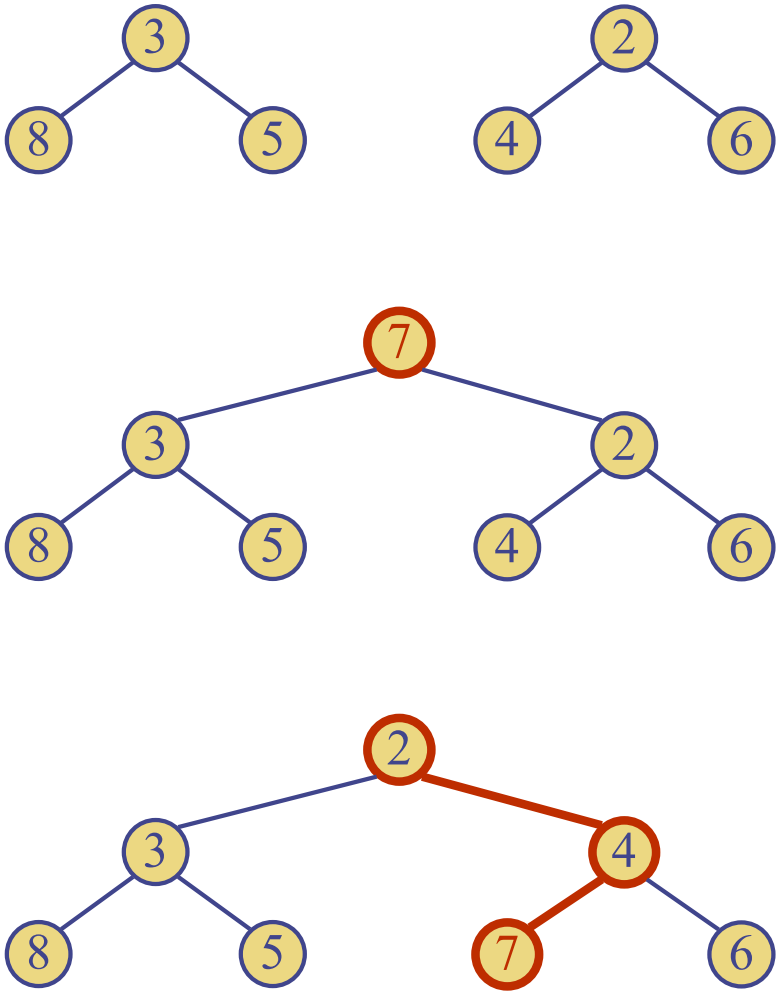
Bottom-Up Heap Construction

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

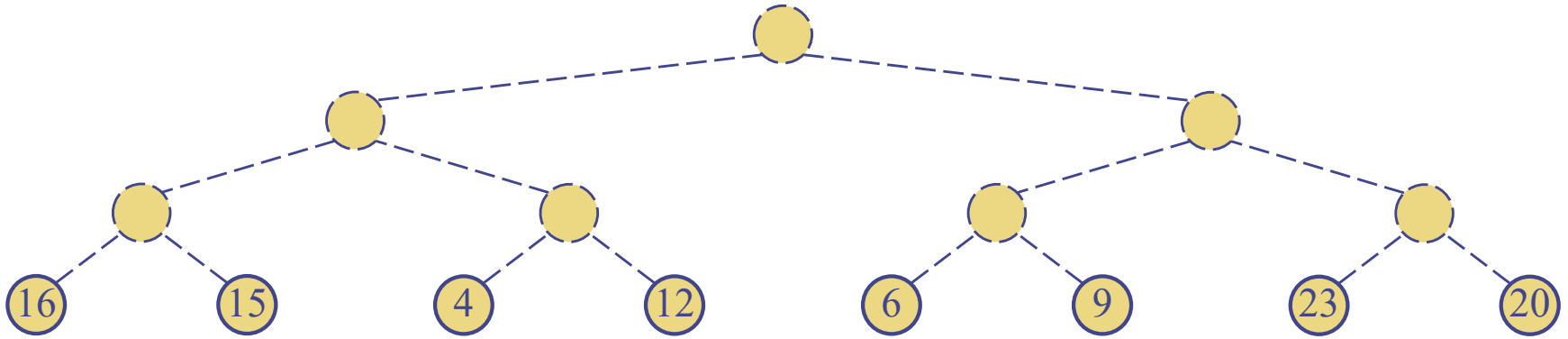


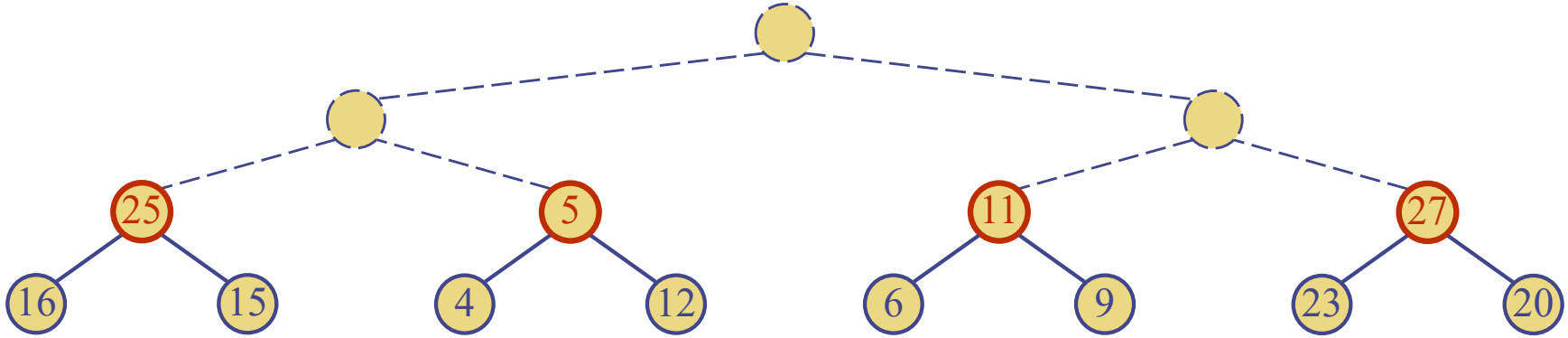
Merging Two Heaps

- 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

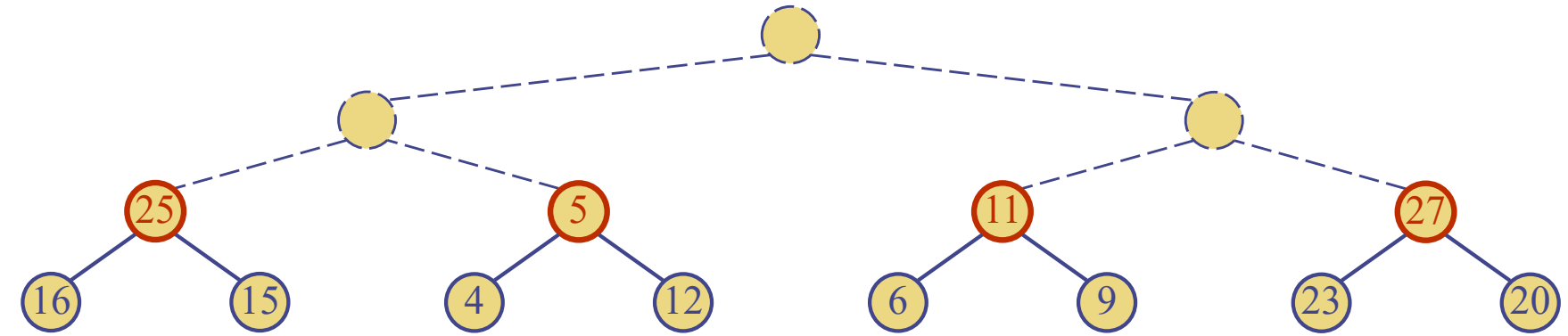


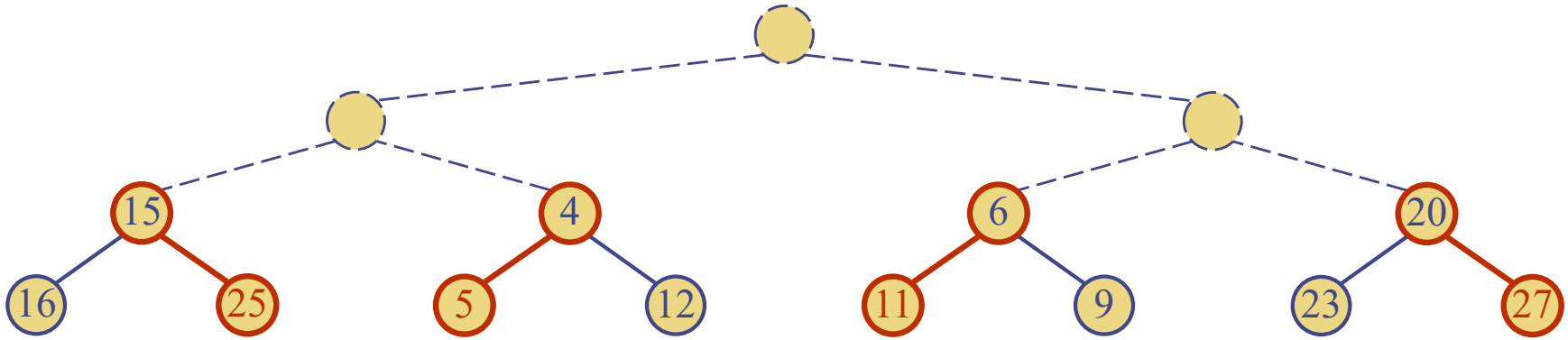
Example



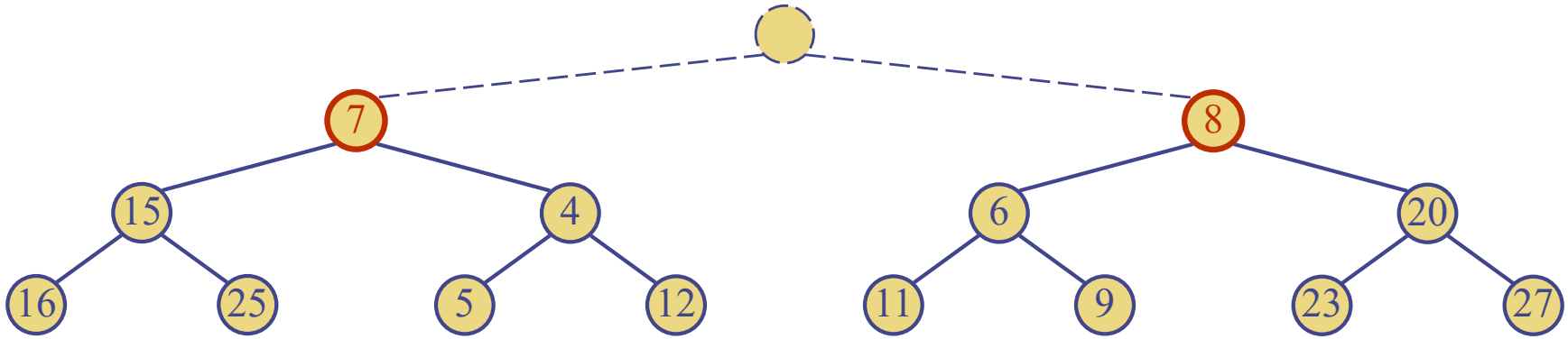


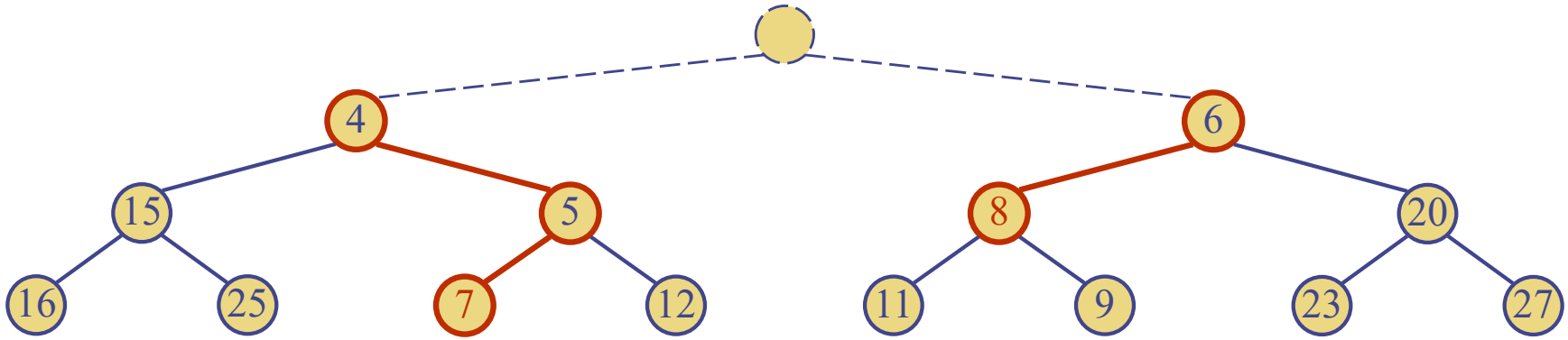
Example (contd.)



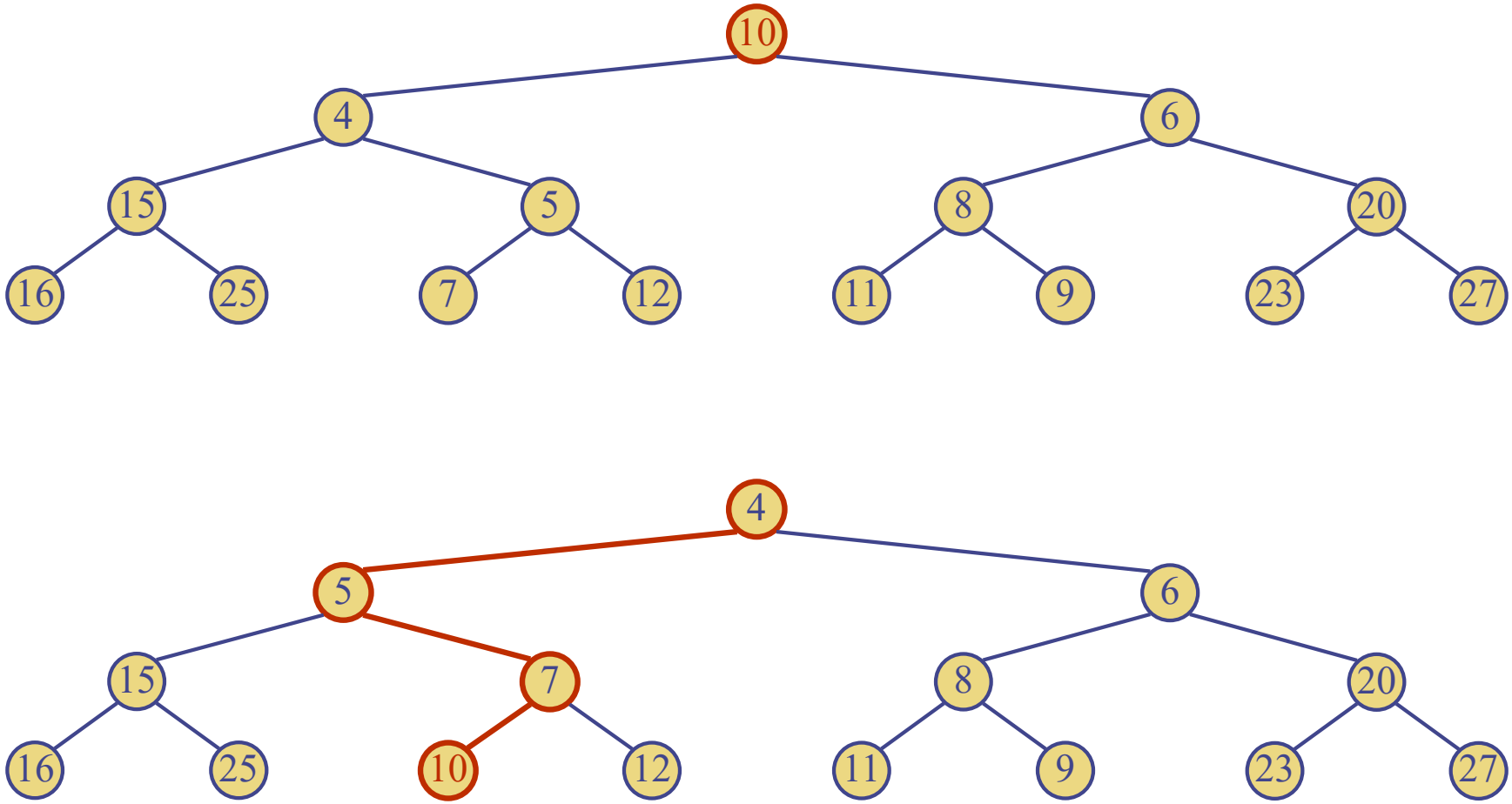


Example (contd.)



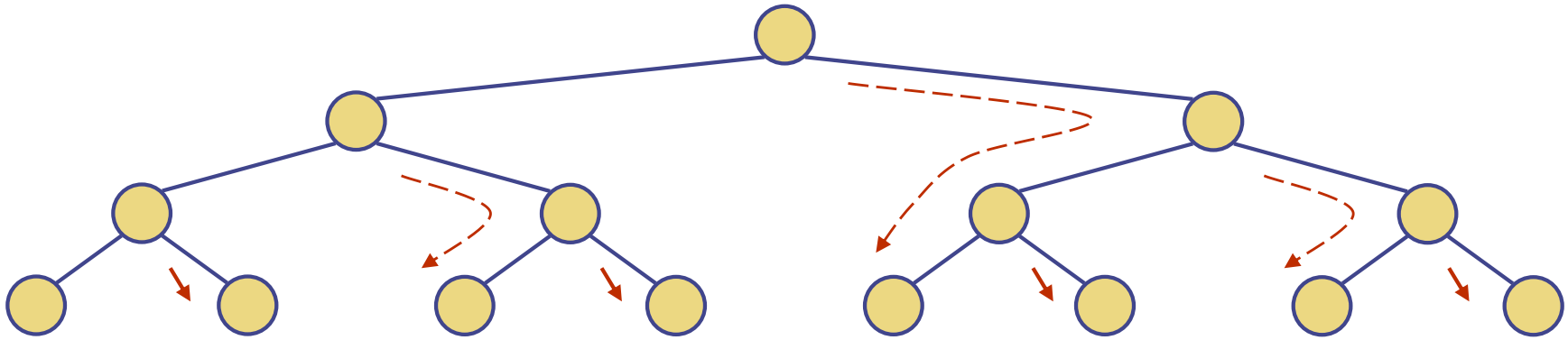


Example (end)



Visual Analysis

- We visualize the worst-case time using proxy paths that first go right and then repeatedly go left until the bottom of the heap is reached (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

- In phase i = 1, 2,… are $(n+1)/2^{i+1}$ pairs of heaps being combined
- Each pair has a downheap path length of i
- Length of all downheap paths during phase i is $(n+1)(i/2^{i+1}) < n(i/2^{i})$
- 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

HeapSort

- Given a list S of n items (keys), remove the items in S and make a heap-based priority queue from the n keys: $O(n)$ time

- Remove the n keys from the heap one at a time and add them to S: $O(?)$ time

$$\sum_{i=0}^{n} \lg i \approx n \lg n$$

So: $O(n \lg n)$ time

- Total: $O(n \lg n)$ time