

Priority Queues

Priority Queue ADT

- Stores a collection of (key, element) pairs
- Main methods
 - `insertItem(k, o)`: inserts an item with key `k` and element `o`
 - `removeMin()`: removes the item with smallest key and returns its element
 - `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:
 - Multithreading
 - Triage

Keys must be comparable

- Keys in a priority queue can be arbitrary objects on which a **total order relation** is defined
- A generic priority queue uses an auxiliary **Comparator ADT**
 - Encapsulates the action of comparing two objects according to a given total order relation
 - The comparator is external to the keys being compared
 - When the priority queue needs to compare two keys, it uses its comparator
 - `isLessThan(x,y)`
 - `isLessThanOrEqualTo(x,y)`
 - `isEqualTo(x,y)`
 - `isGreaterThan(x,y)`
 - `isGreaterThanOrEqualTo(x,y)`

Suppose you are given a priority queue implementation, so you have the following operations to work with:

`insertItem(k, o)`

`removeMin()`

`minKey()`

`minElement()`

`size()`

`isEmpty()`

How can you use it to sort a sequence S of numbers?

Sorting with a Priority Queue

We can use a priority queue to sort a set of comparable elements

1. Insert the elements one by one with a series of `insertItem(e, e)` operations
2. Remove the elements in sorted order with a series of `removeMin()` operations

Running time
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 $\neg S.isEmpty()$

e ← *S.remove(S.first())*

P.insertItem(e, e)

while $\neg P.isEmpty()$

e ← *P.removeMin()*

S.insertLast(e)

Sequence-based Priority Queue

Implementation with an **unsorted** sequence 

- Store the items of the priority queue in a list-based sequence, in arbitrary order
- **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** sequence 

- Store the items of the priority queue in a sequence, sorted by key
- **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

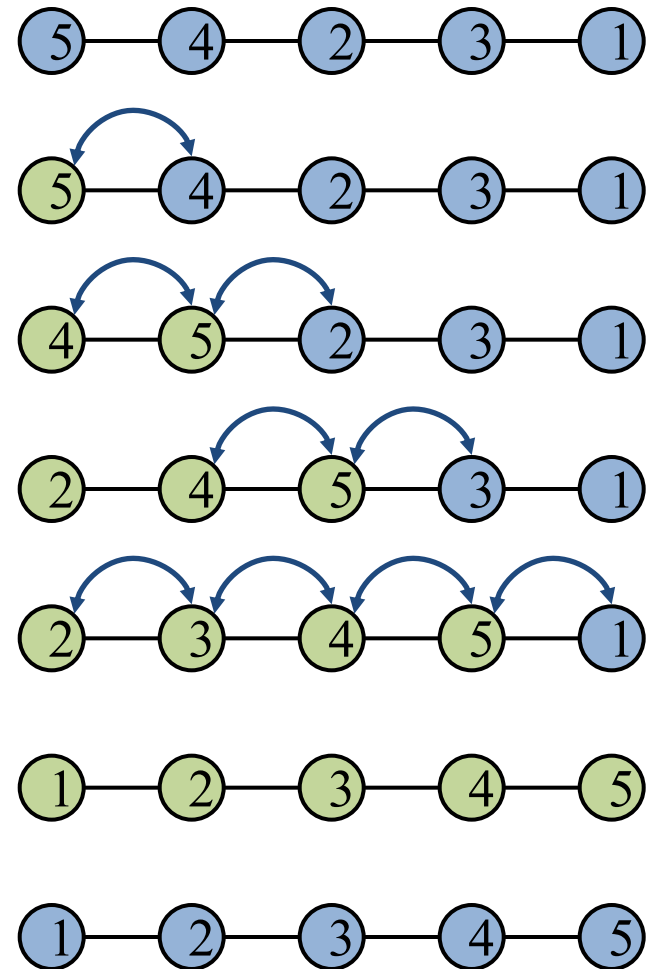
- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an **unsorted** sequence
- Running time of Selection-sort:
 1. Inserting the elements into the priority queue with n **insertItem** operations takes $O(n)$ time
 2. Removing the elements in sorted order from the priority queue with n **removeMin** operations takes time proportional to
$$1 + 2 + \dots + n$$
- 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:
 1. Inserting the elements into the priority queue with n **insertItem** operations takes time proportional to
$$1 + 2 + \dots + n$$
 2. Removing the elements in sorted order from the priority queue with a series of n **removeMin** operations takes $O(n)$ time
- Runs in $O(n^2)$ time

In-place Insertion-sort

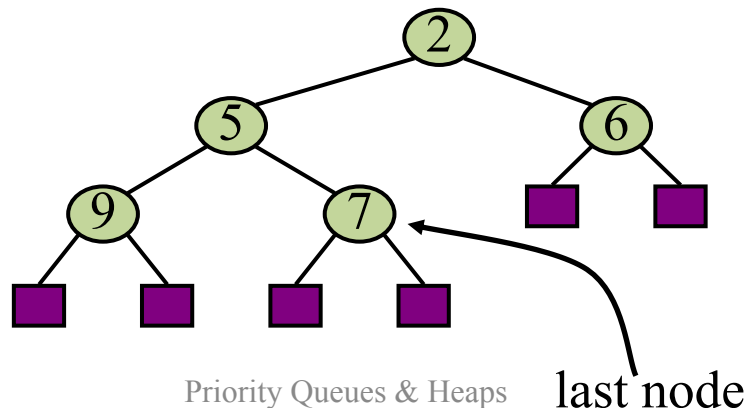
- Instead of using an external data structure, we can implement selection-sort and insertion-sort **in-place**
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
 - We keep sorted the initial portion of the sequence
 - We can use **swapElements** instead of modifying the sequence



Heap-Based Priority Queue

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 $i = 0, \dots, h - 2$, there are 2^i internal 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 $h - 1$

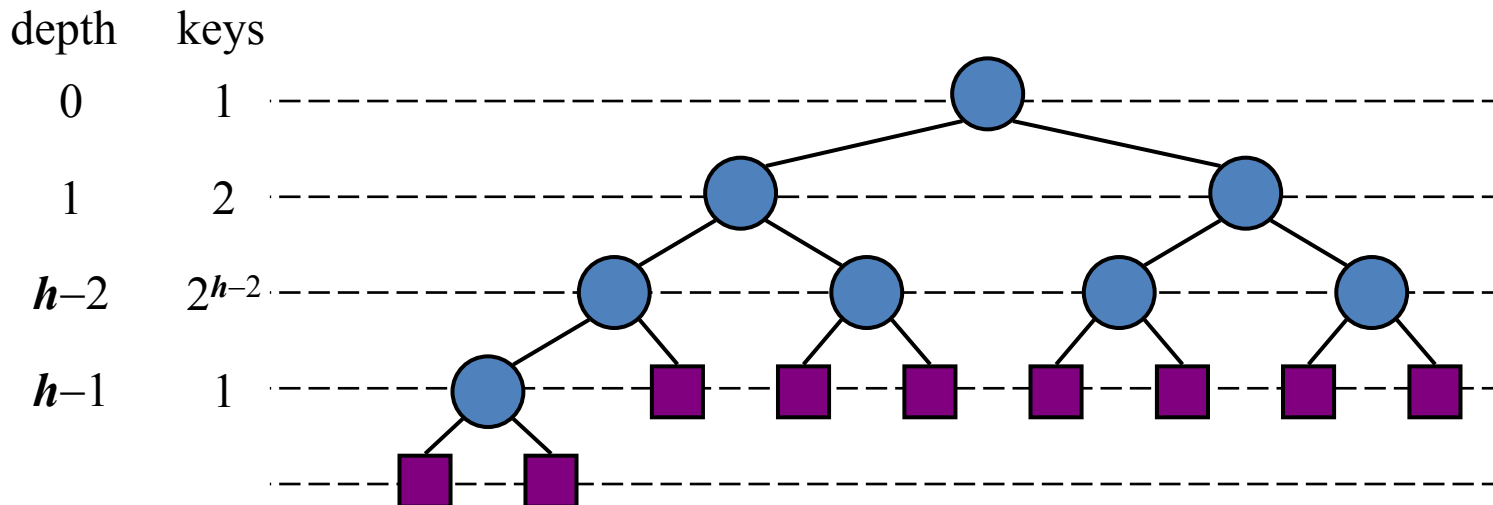


Height of a Heap

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

Proof: (we apply the complete binary tree property)

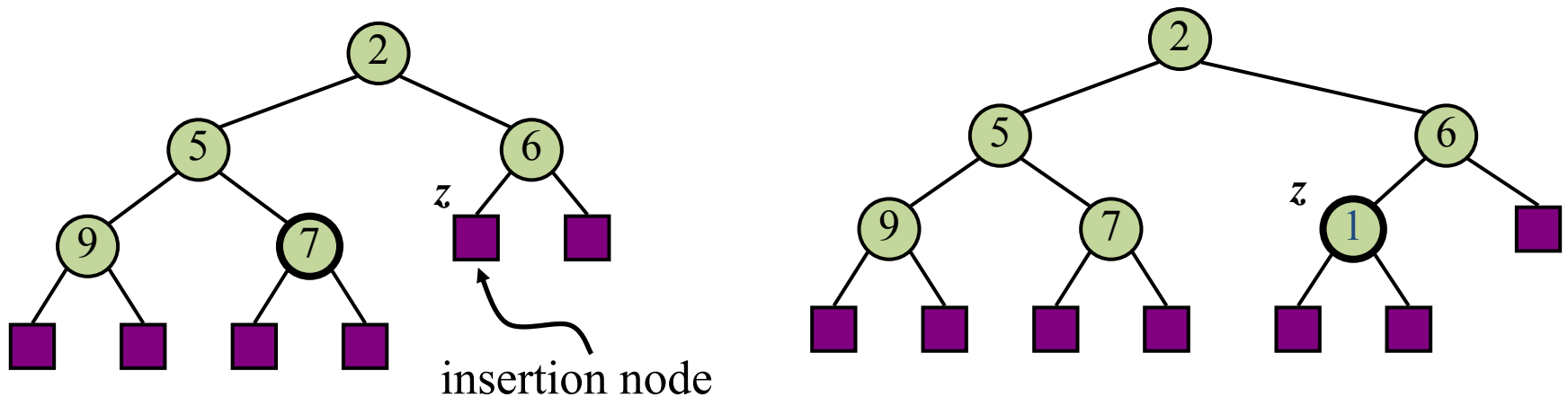
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \dots, h - 2$ and at least one key at depth $h - 1$, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-2} + 1$
- Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



Insertion into a Heap: insertItem(k, o)

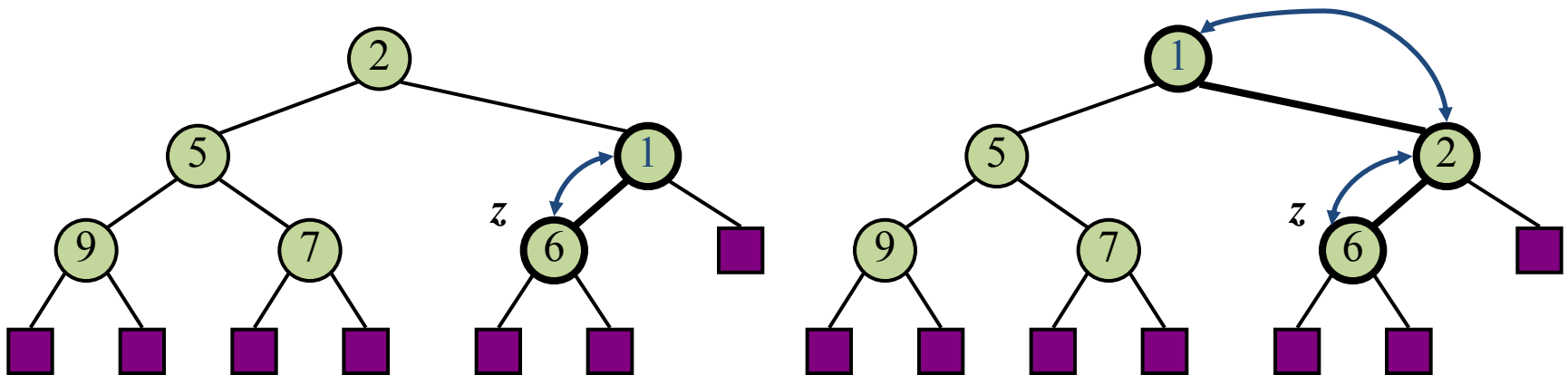
Consists of three steps:

- 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 Bubbling

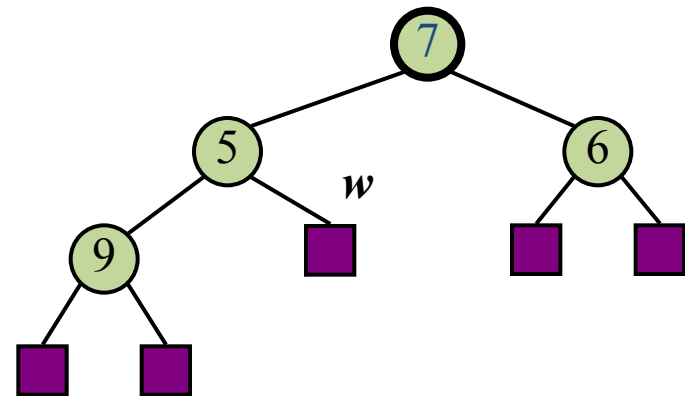
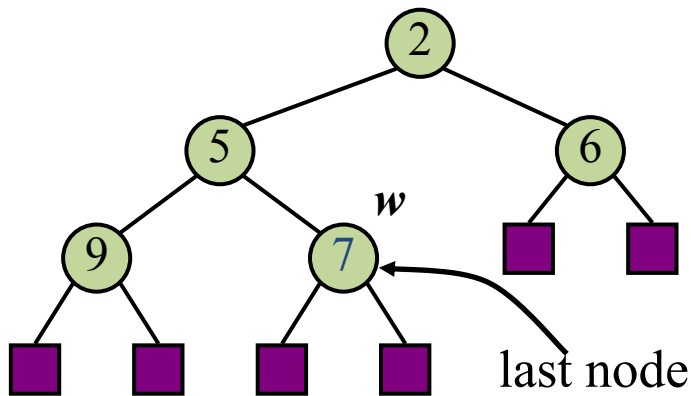
- After the insertion of a new key k , the heap-order property may be violated
- Algorithm **upheap** restores the heap-order property by swapping k along an **upward path** from the insertion node
- Terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to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



Removal from a Heap: removeMin()

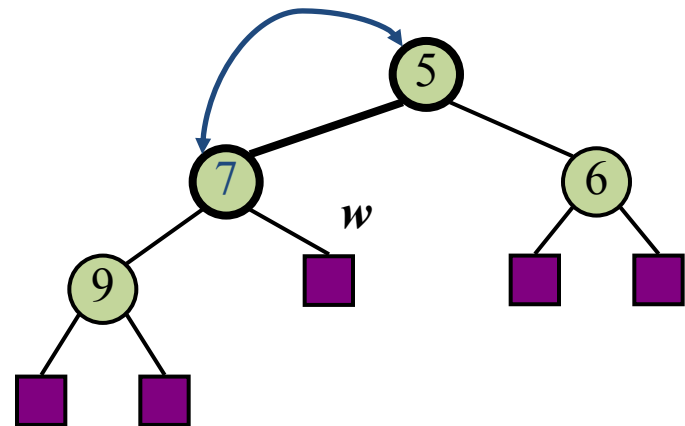
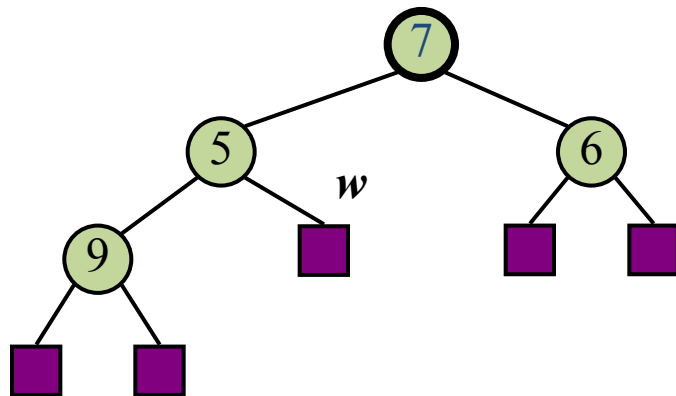
Consists of three steps

- 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)



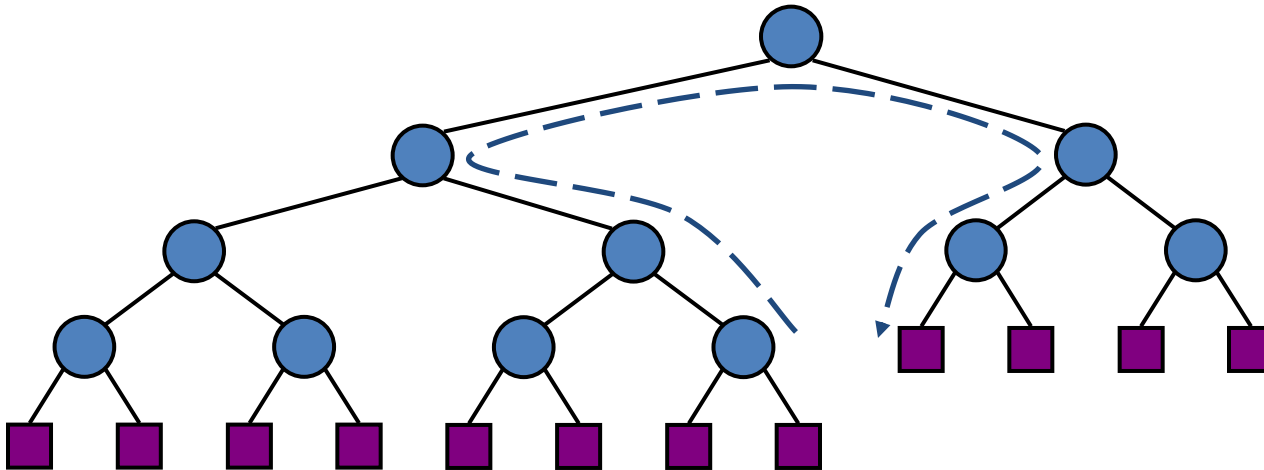
Downheap Bubbling

- 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 the **smallest key among children** along a **downward path** from the root
- 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



Finding the Last Node

- The last 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 of v , go to the right child of v
 - While the current node is internal, go to the left child
- Similar algorithm for updating the last node after a removal



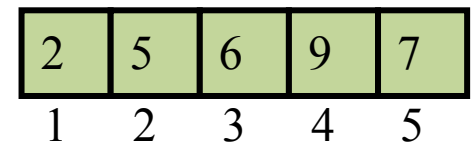
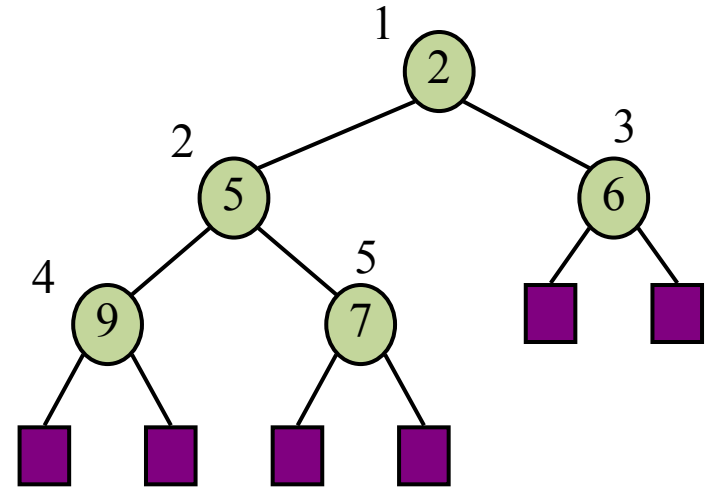
Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
 - the space used is $O(n)$
 - methods `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
 - much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Vector-based Heap Implementation

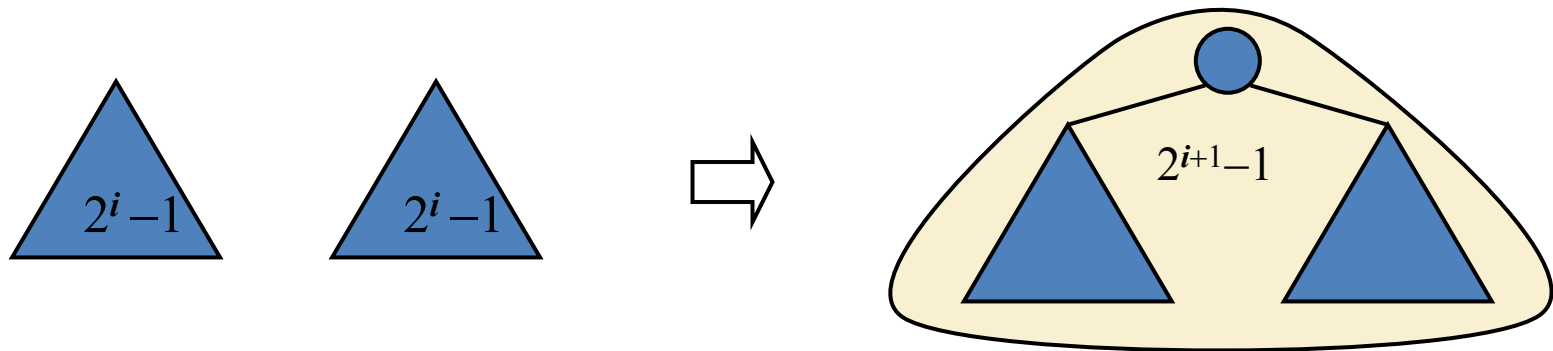
We can represent a heap with n keys by means of a vector of length $n + 1$

- For the node at rank i
 - left child is at rank $2i$
 - right child is at rank $2i + 1$
- What does not need to be stored:
 - links between nodes
 - leaves
- The cell at rank 0 is not used
- Last node is at rank n
 - `insertItem` inserts at rank $n + 1$
 - `removeMin` removes at rank n (after swapping root with last node)
- Yields in-place heap-sort



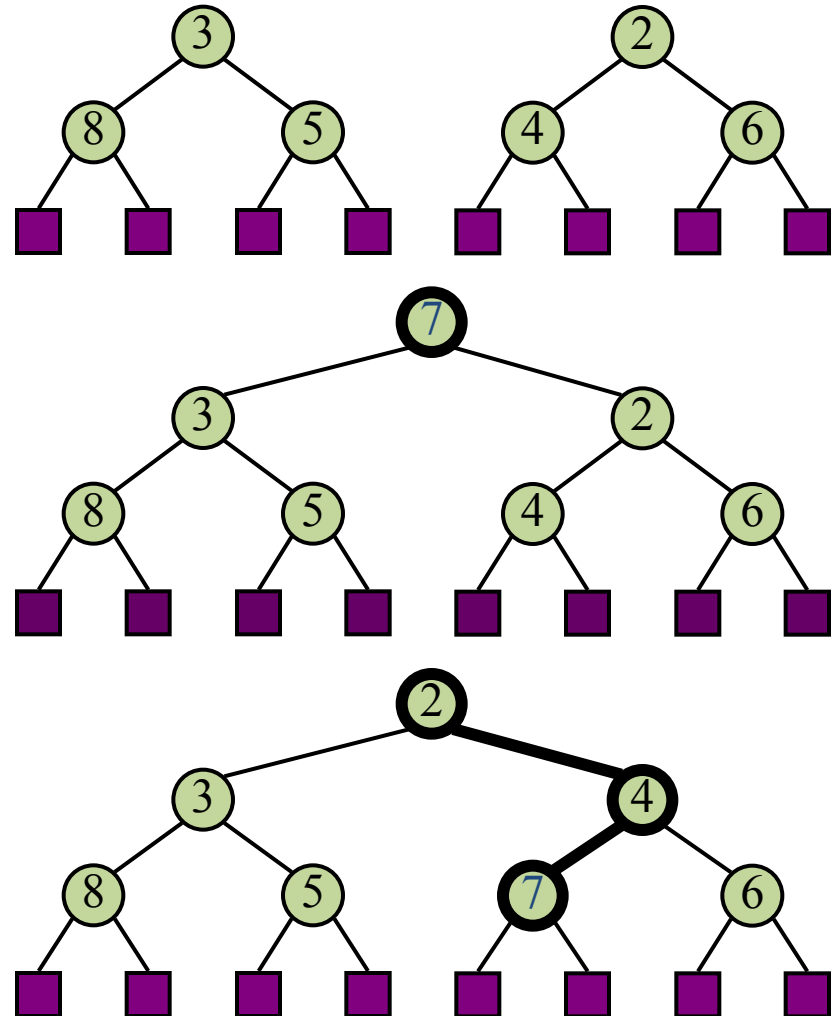
Bottom-up Heap Construction

- If **all keys are known in advance**, we can build a heap recursively
- For simplicity, assume number of keys $n = 2^h - 1$ so the heap is a complete binary tree with every level being full
- Given n keys, build heap 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

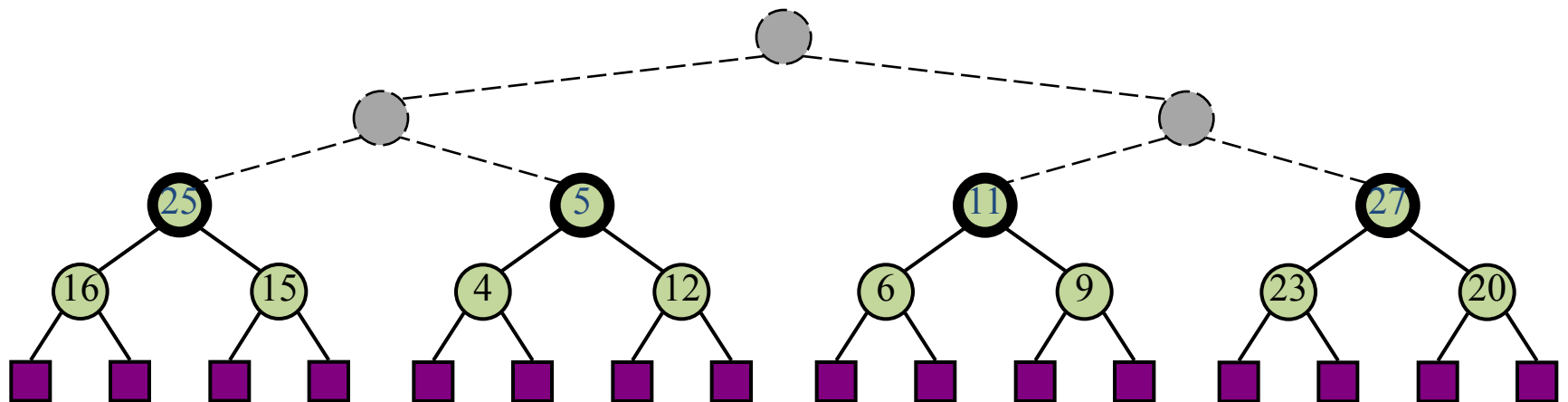
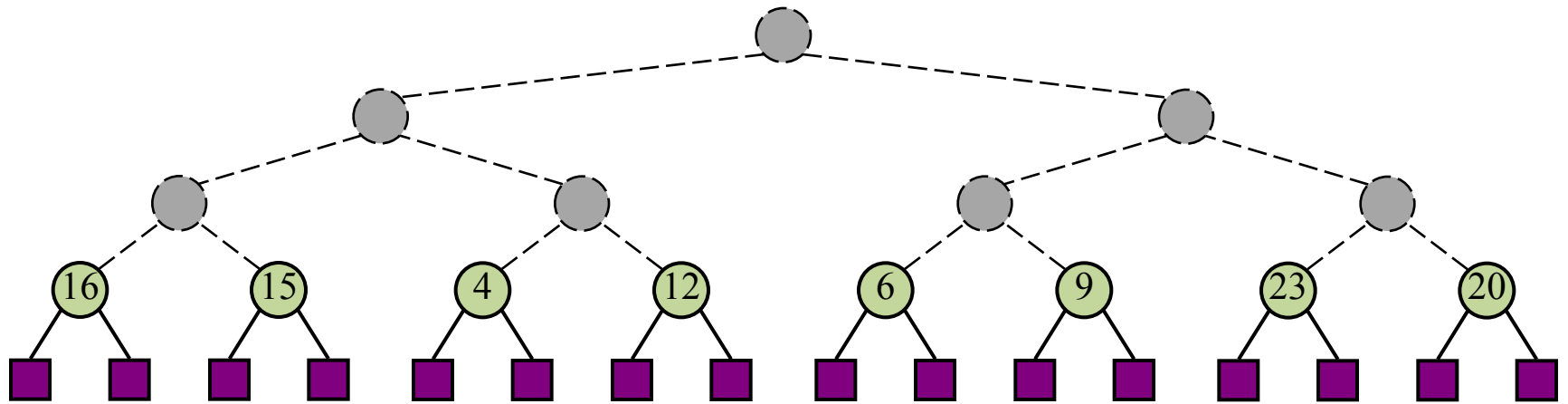


Merging Two Heaps

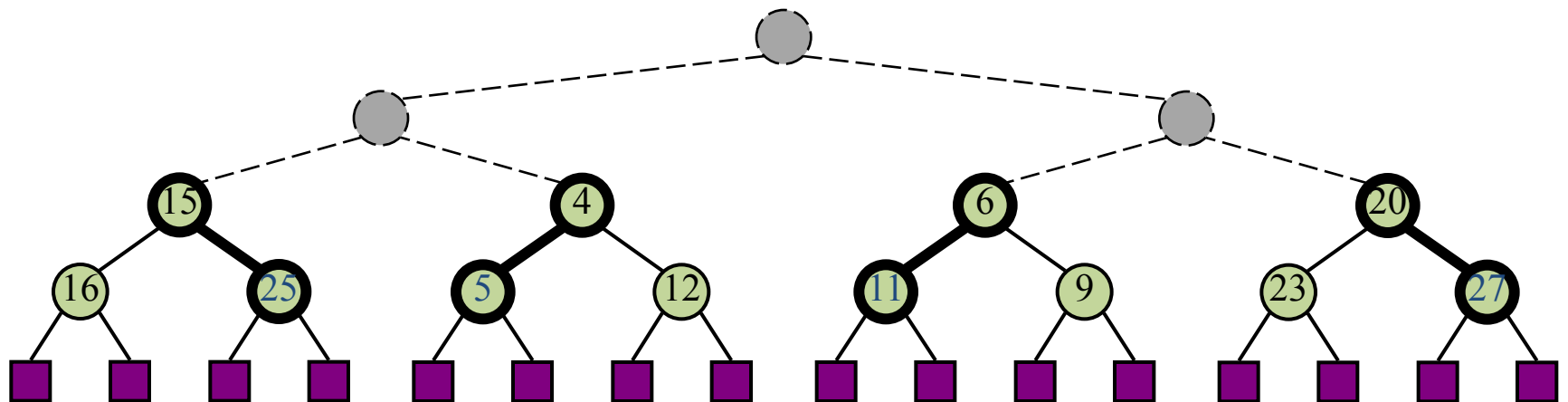
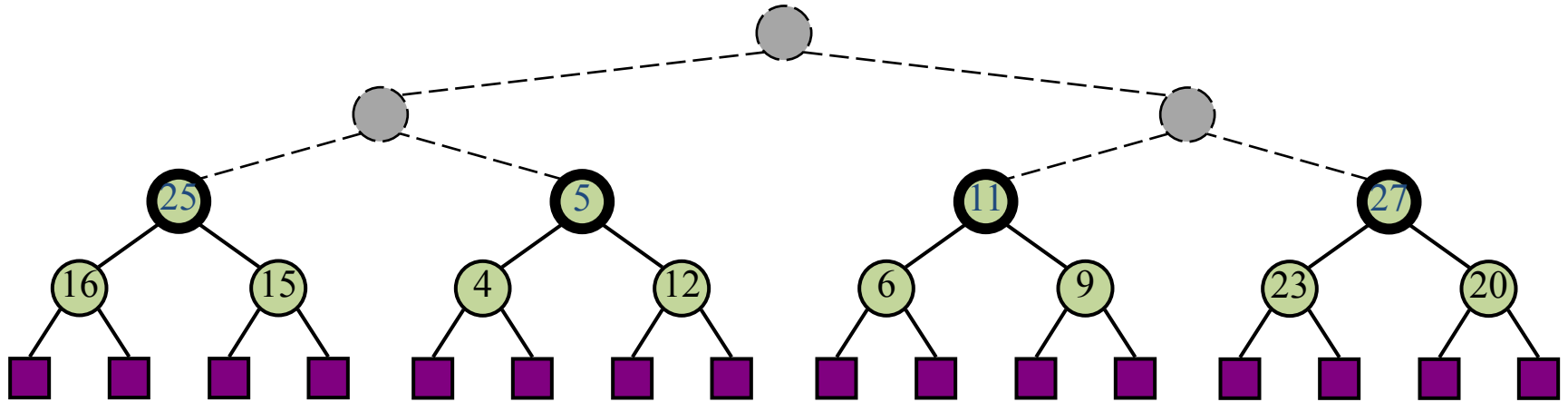
- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform **downheap** to restore the heap-order property



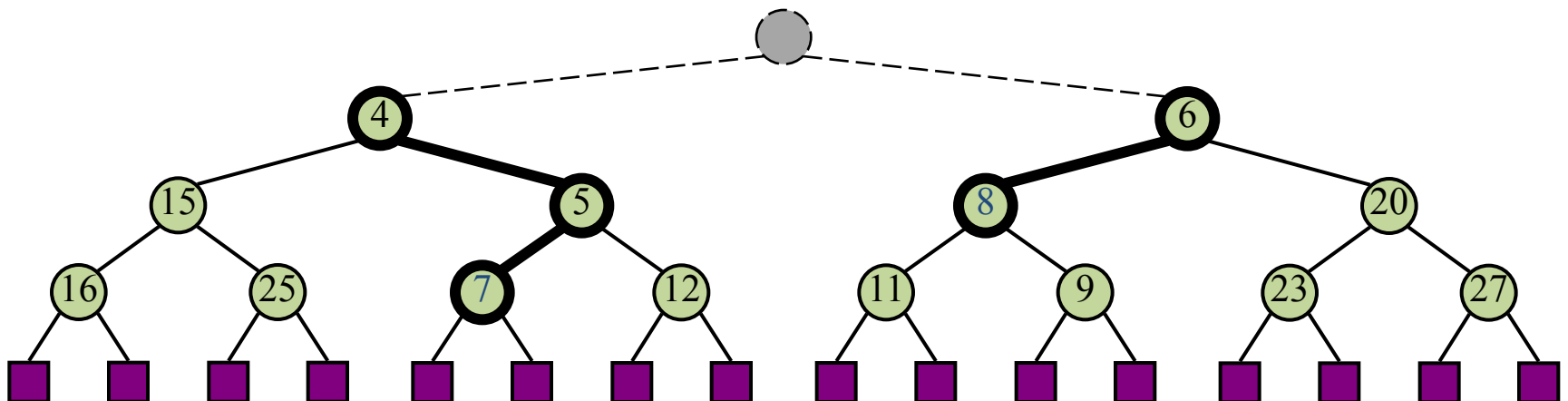
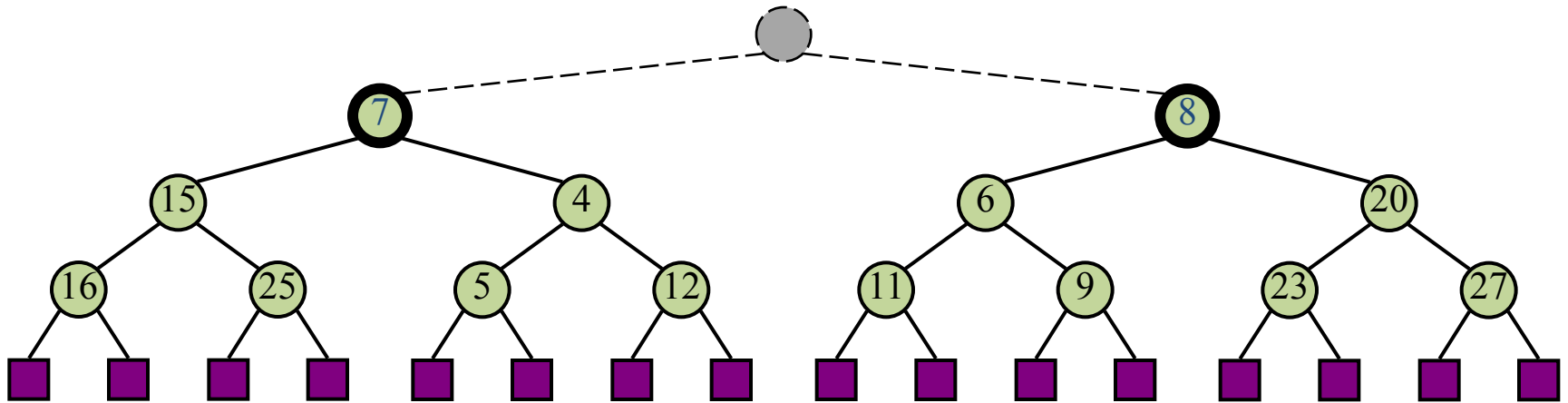
Example



Example (contd.)



Example (contd.)



Example (end)

