Priority Queues
&
Sorting
&
Heap

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

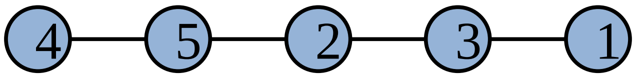
1. Insert the elements of a sequence S one by one with a series of insertItem(e, e) operations into an initially empty priority Queue P.
2. Remove the elements in sorted order from priority queue P with a series of removeMin() operations and put them back to into sequence S .

Running time depends on the priority queue implementation

Algorithm ***PQ-Sort***(***S***, ***C***)
    Input sequence ***S***, a priority Queue P
    Output sequence ***S*** sorted in increasing order
    while ¬ ***S.isEmpty*** ()
        e ← ***S.remove*** (***S. first*** ())
        P.insertItem(***e***, ***e***)
    while ¬ ***P.isEmpty***()
        e ← ***P.removeMin***()
        S.insertLast(***e***)

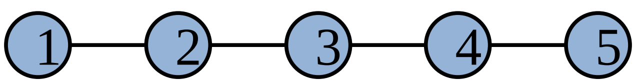
Sequence-based Priority Queue

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

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

- 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 $\boldsymbol{n}$ insertItem operations takes time proportional to

     $$1 + 2 + \ldots + \boldsymbol{n}$$

  2. Removing the elements in sorted order from the priority queue with a series of $\boldsymbol{n}$ removeMin operations takes $\boldsymbol{O}(\boldsymbol{n})$ time

- Runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

In-Place
Selection and Insertion
Sorting algorithms

Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place.

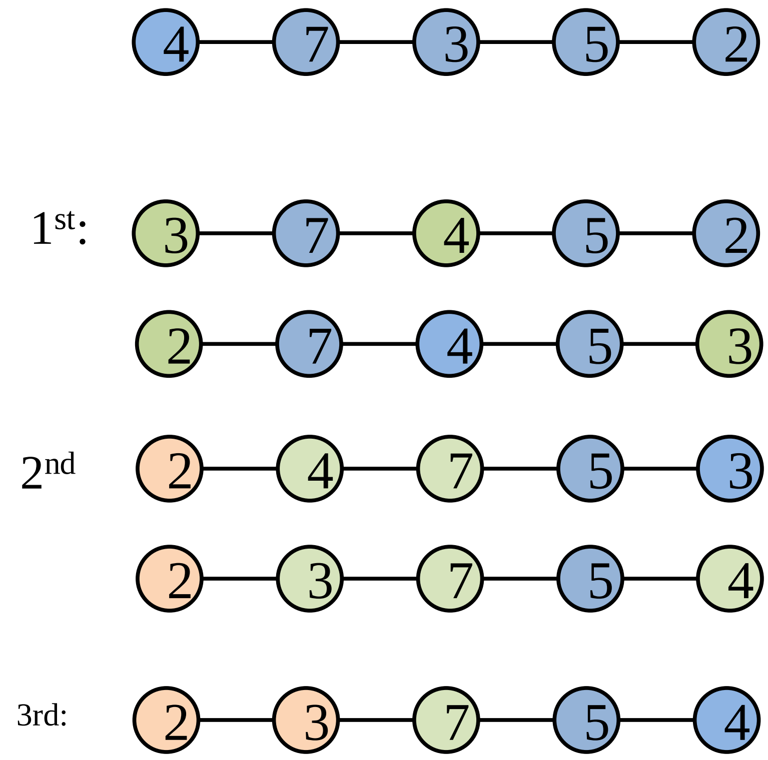
In-Place Selection Sort

- The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.
- The algorithm maintains two subarrays in a given array:,
  1) the sorted part at the left end and
  2) the unsorted part at the right end.

In-Place Selection Sort Pseudocode

```
Selection-Sort(A)
    Input: Unsorted Array A of size n
    Output: Sorted Array A.
    for i=1 to n-1 do
        for j=i+1 to n do
            if A[i]>A[j] then
                tmp=A[i]
                A[i]=A[j]
                A[lj=key
```

Runs in $O(n^2)$ time



Why does it need to run for only the first n-1 elements, rather than for all n elements?

In-place Insertion-sort

- Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

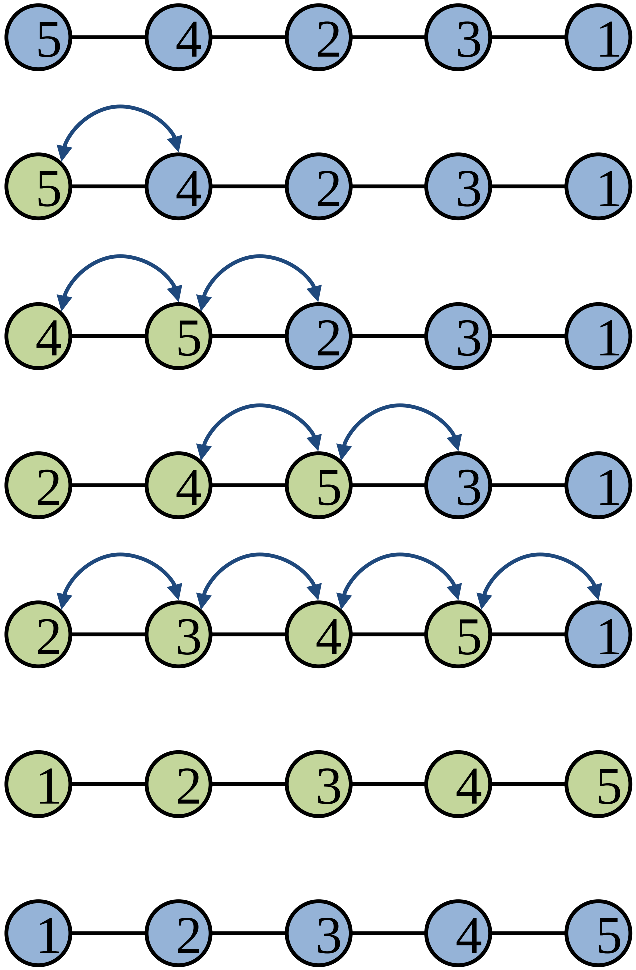
```
Insertion-Sort(A)
    Input: Unsorted Array A of size n
    Output: Sorted Array A.
    for i=2 to n do
        key=A[i]
        j=i-1
        while j>0 and A[j]>key
            A[j+1]=A[j]
            j=j-1
        A[j+1]=key
```

Runs in $O(n^2)$ time

In-place Insertion-sort

- 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

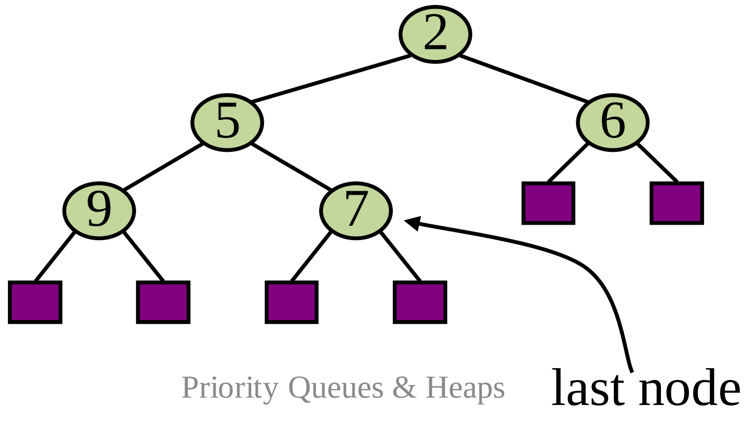


Heap-Based Priority Queue

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, $\boldsymbol{key}(\boldsymbol{v}) \geq \boldsymbol{key}(\boldsymbol{parent}(\boldsymbol{v}))$
  - Complete Binary Tree: let $\boldsymbol{h}$ be the height of the heap
    - for $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 2$, there are $2^{\boldsymbol{i}}$ internal 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 internal node of depth $\boldsymbol{h} - 1$

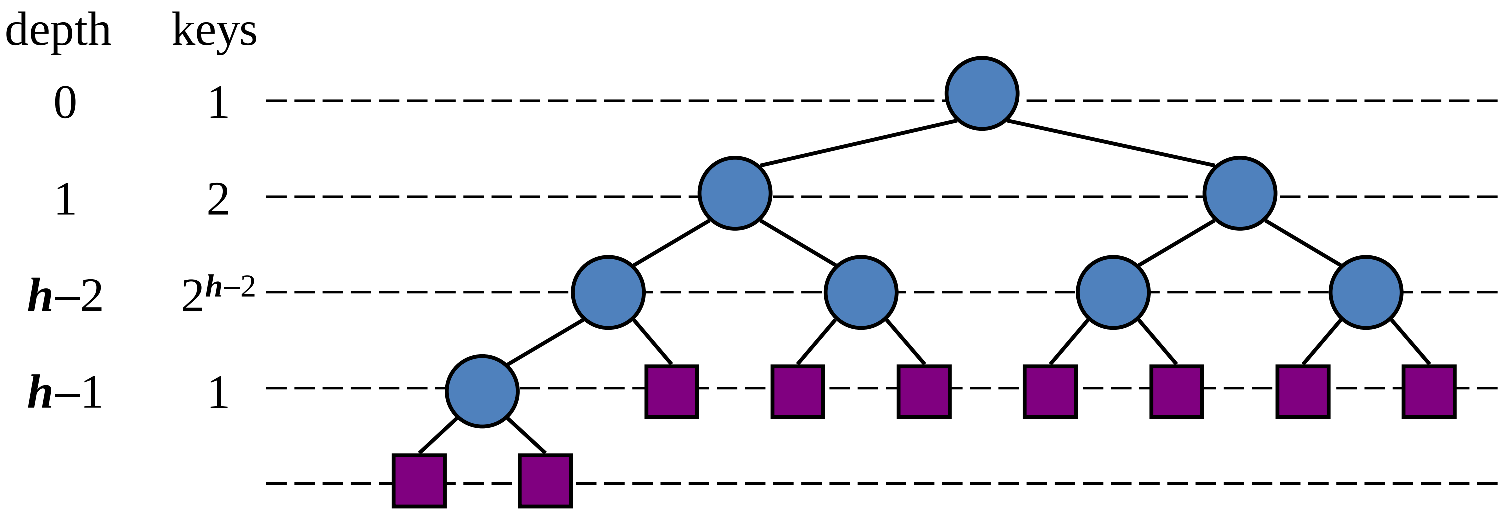


Height of a Heap

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

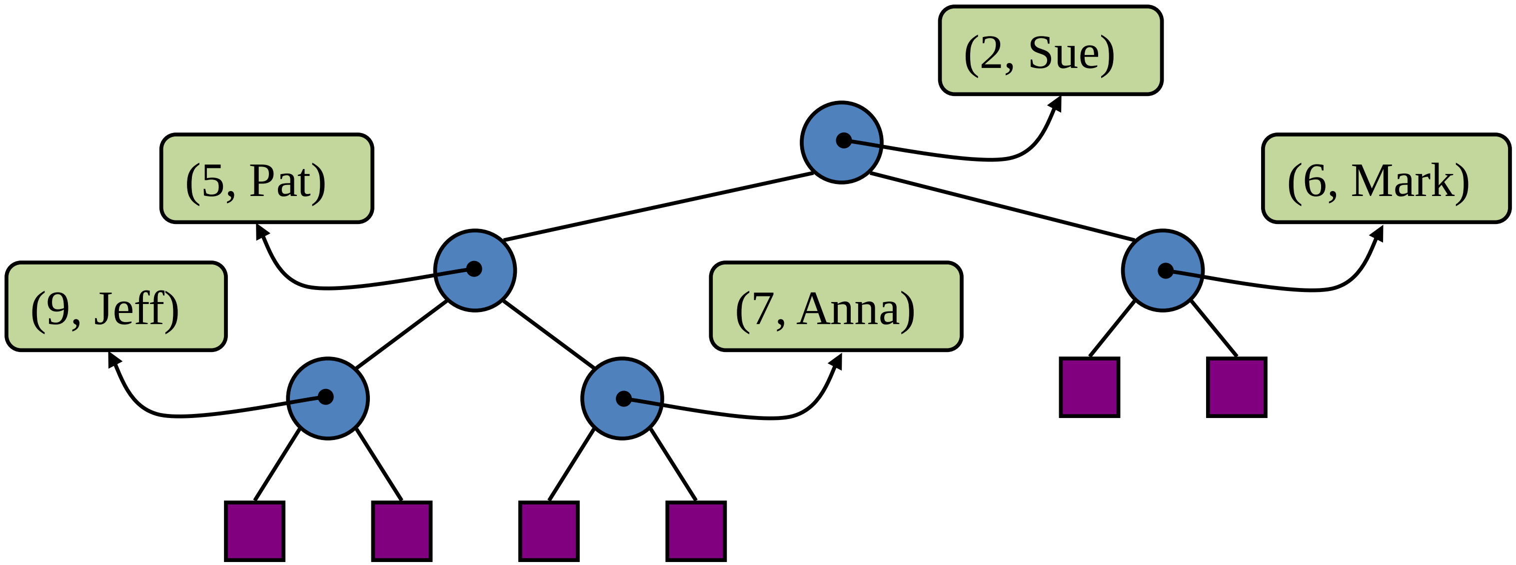
Proof: (we apply the complete binary tree property)

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



Heaps and Priority Queues

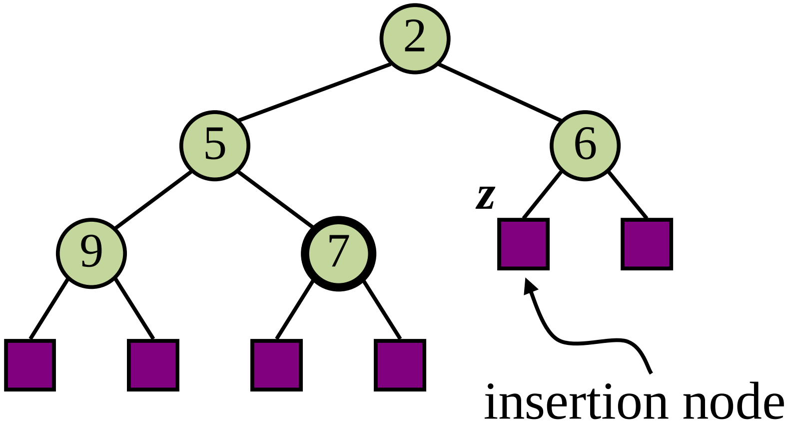
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures

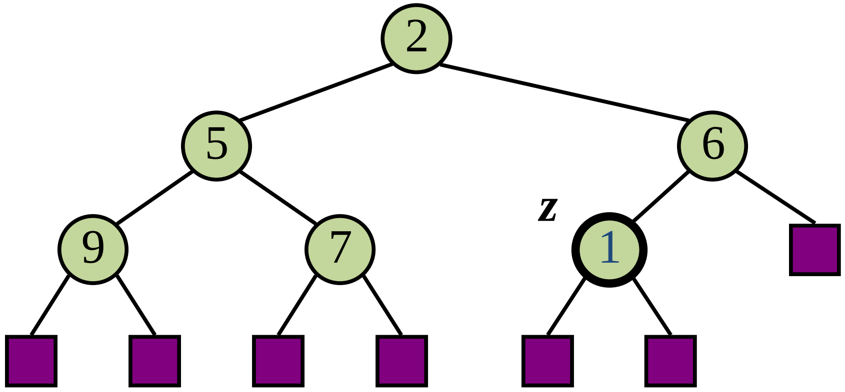


Insertion into a Heap: insertItem(*k*,*o*)

Consists of three steps:

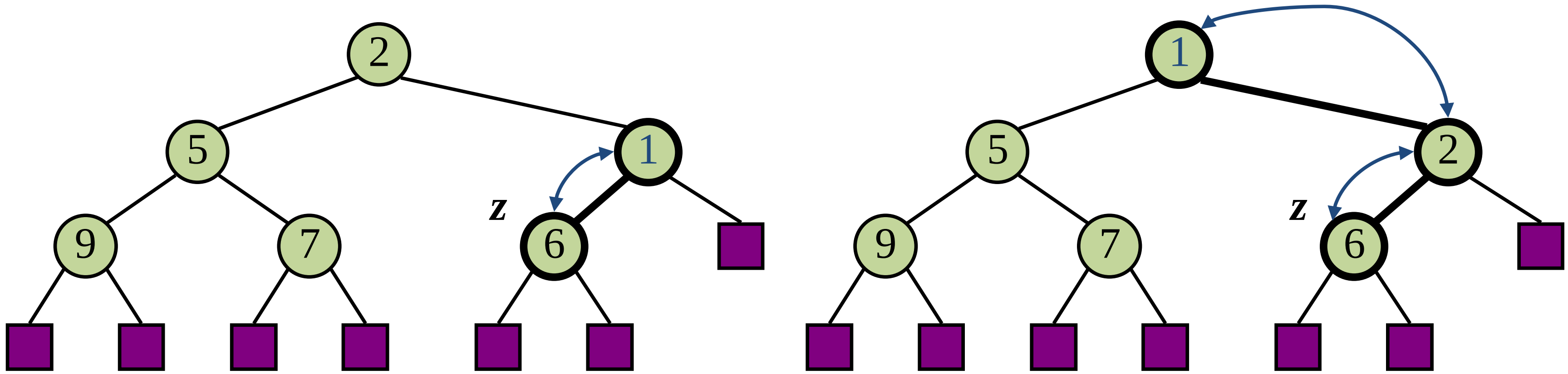
- Find the insertion node ***z*** (the new last node)
- Store ***k*** at ***z*** and expand ***z*** into an internal node
- Restore the heap-order property (discussed next)





Upheap Bubbling

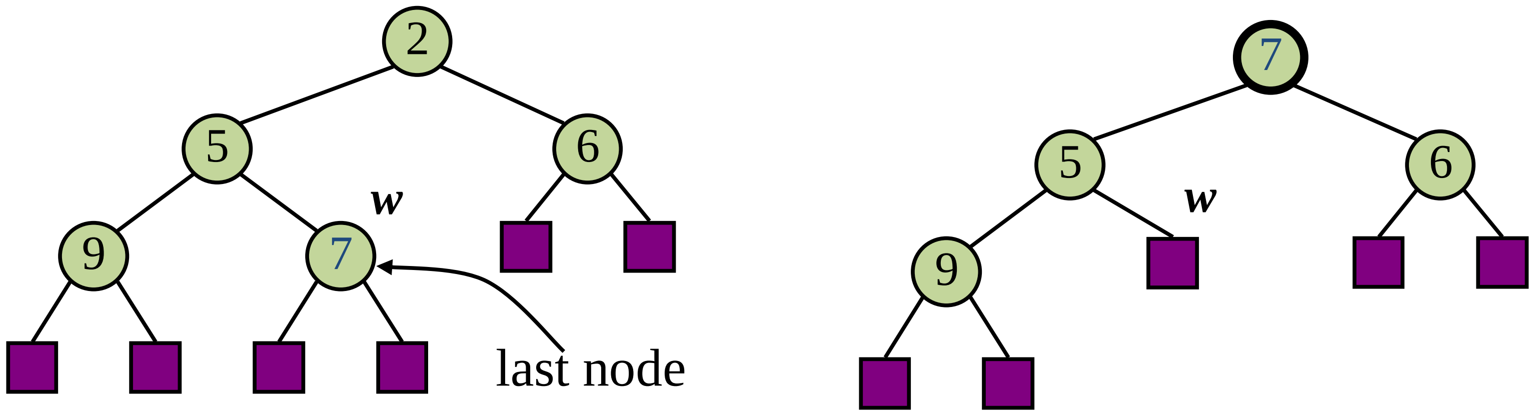
- After the insertion of a new key ***k***, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping ***k*** along an upward path from the insertion node
- Terminates when the key ***k*** reaches the root or a node whose parent has a key smaller than or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



Removal from a Heap: removeMin()

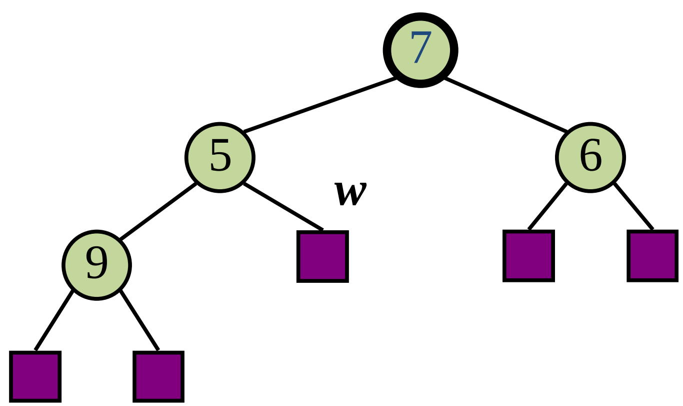
Consists of three steps

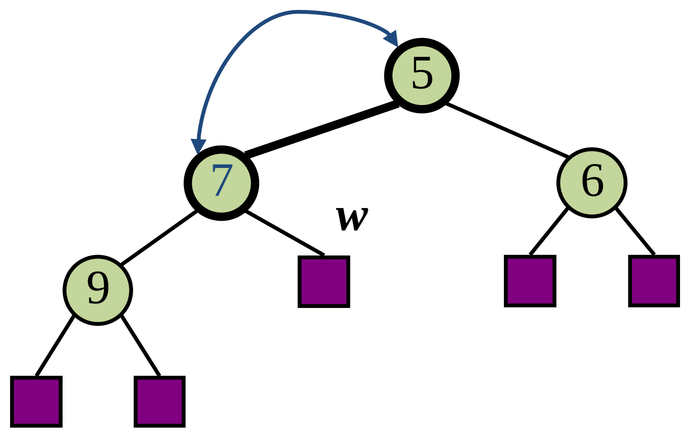
- Replace the root key with the key of the last node ***w***
- Compress ***w*** and its children into a leaf
- Restore the heap-order property (discussed next)



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





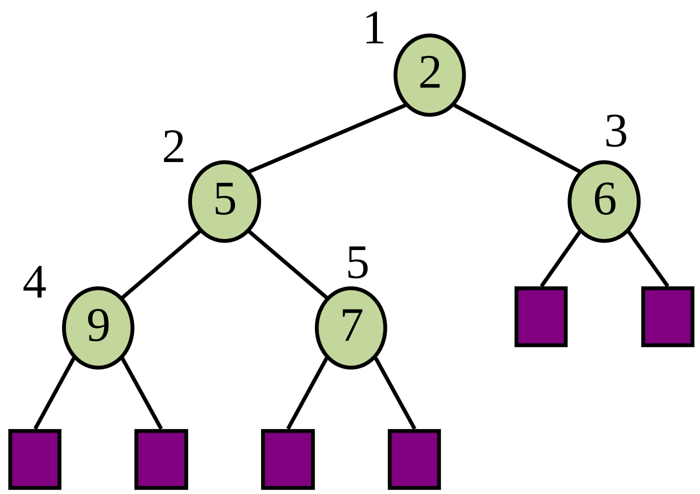
Heap-Sort

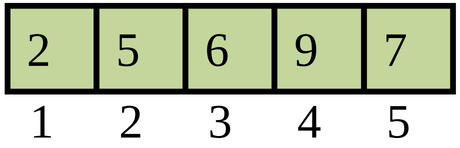
- Consider a priority queue with ***n*** items implemented by means of a heap
  - the space used is $O(n)$
  - methods insertItem and removeMin take $O(\log n)$ time
  - methods size, isEmpty, minKey, and minElement take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of ***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

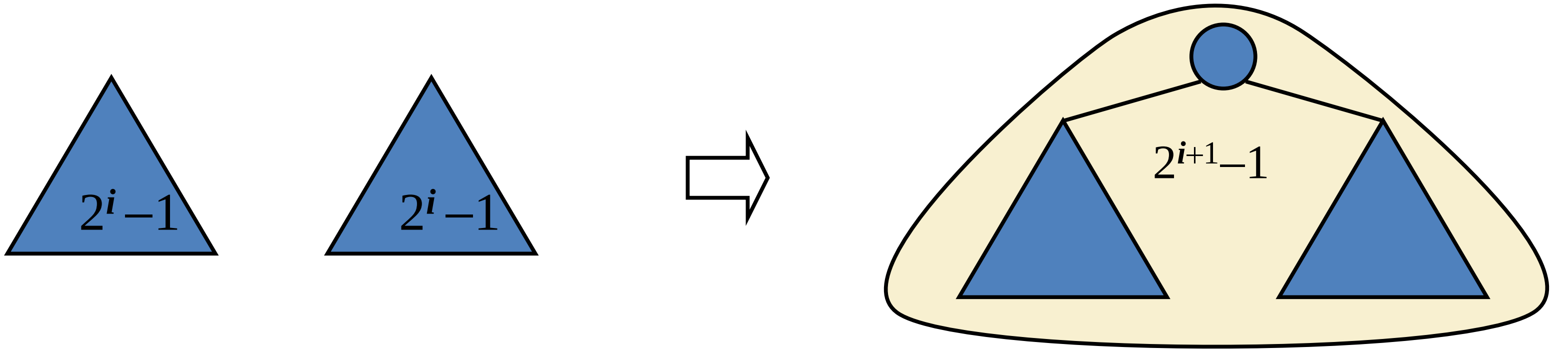
- For the node at rank ***i***
  - left child is at rank 2***i***
  - right child is at rank 2***i*** + 1
- What does not need to be stored:
  - links between nodes
  - leaves
- The cell at rank 0 is not used
- Last node is at rank ***n***
  - insertItem inserts at rank ***n*** + 1
  - removeMin removes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





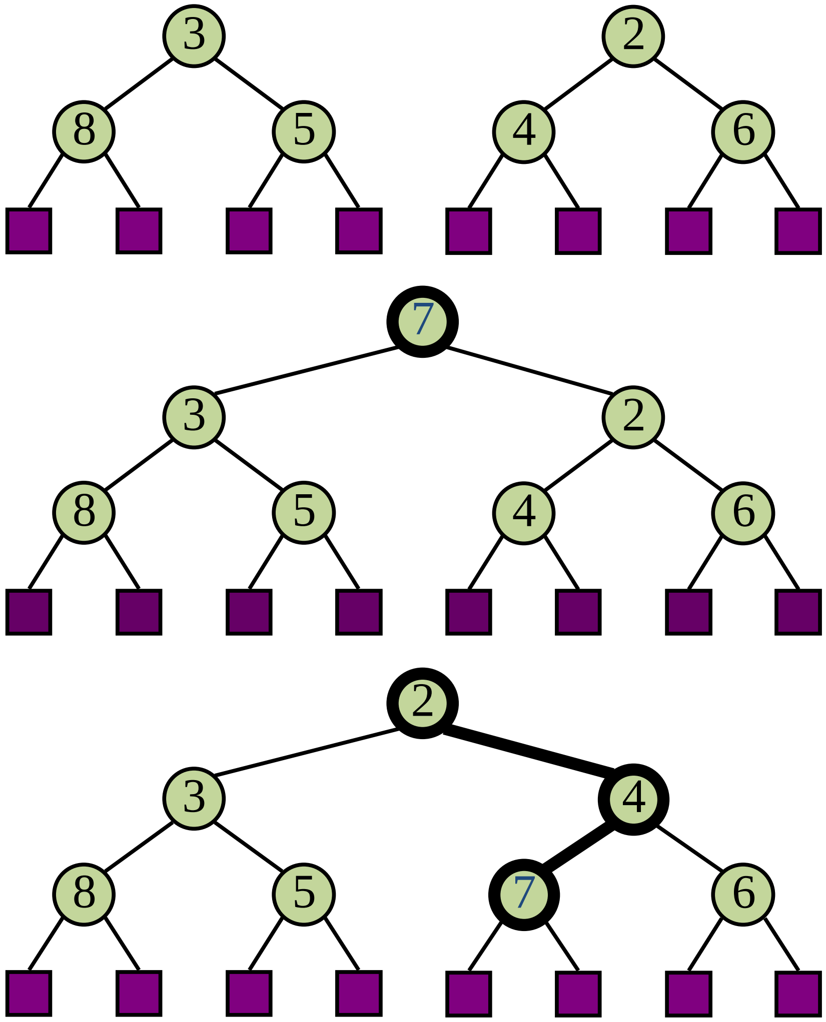
Bottom-up Heap Construction

- If all keys are known in advance, we can build a heap recursively
- For simplicity, assume number of keys $\boldsymbol{n} = 2^h - 1$ so the heap is a complete binary tree, so each depth $\boldsymbol{i} = 0, \ldots, \boldsymbol{h} - 2$ contains $2^{\boldsymbol{i}}$ containing internal nodes
- Given $\boldsymbol{n}$ keys, build heap using a bottom-up construction with log $\boldsymbol{n}$ phases
- In phase $\boldsymbol{i}$, pairs of heaps with $2^{\boldsymbol{i}}-1$ keys are merged into heaps with $2^{\boldsymbol{i}+1}-1$ keys

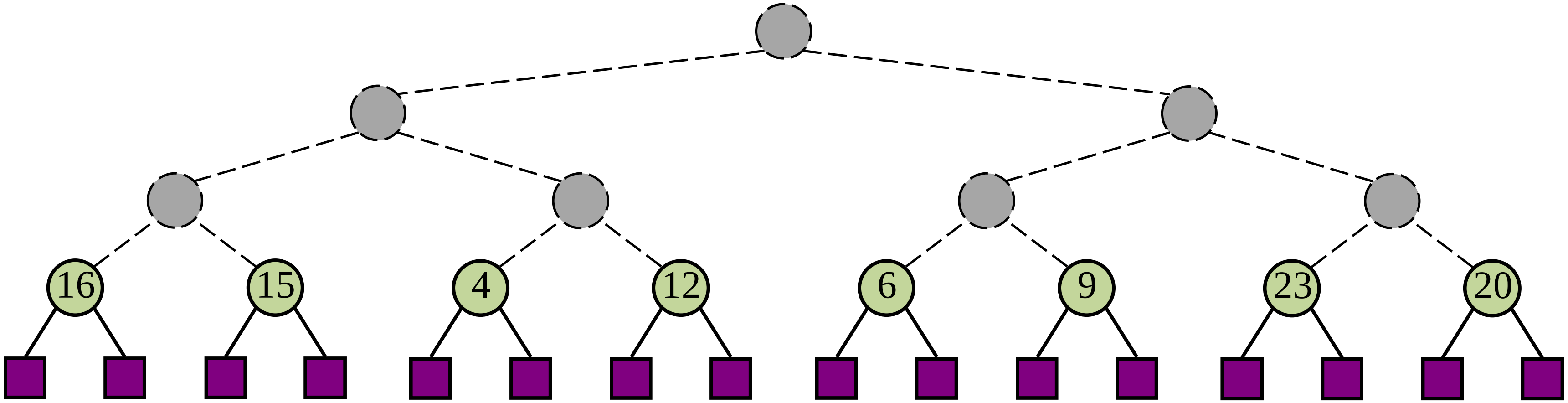


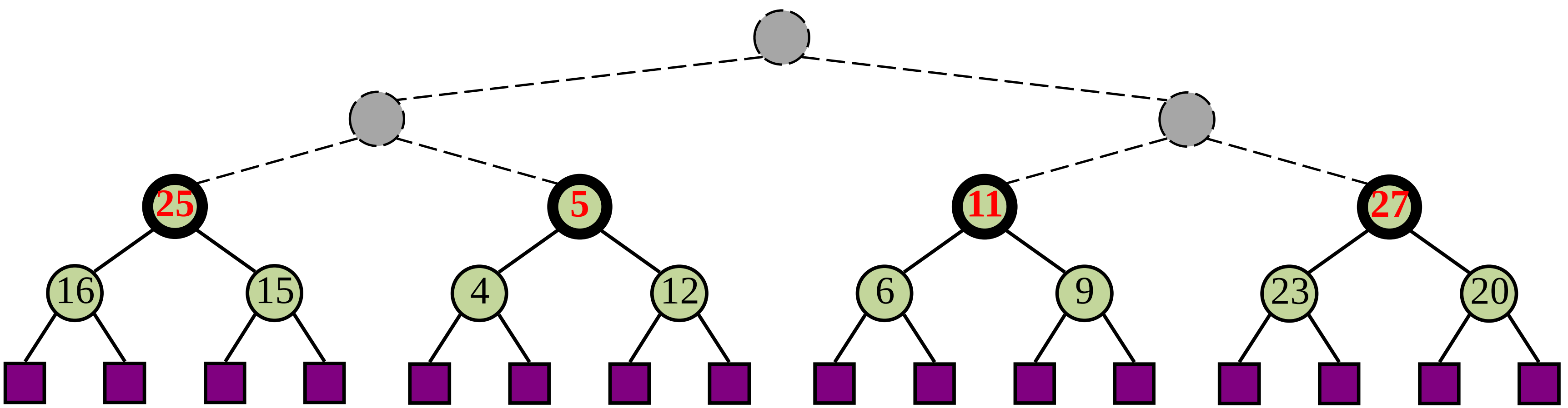
Merging Two Heaps

- 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

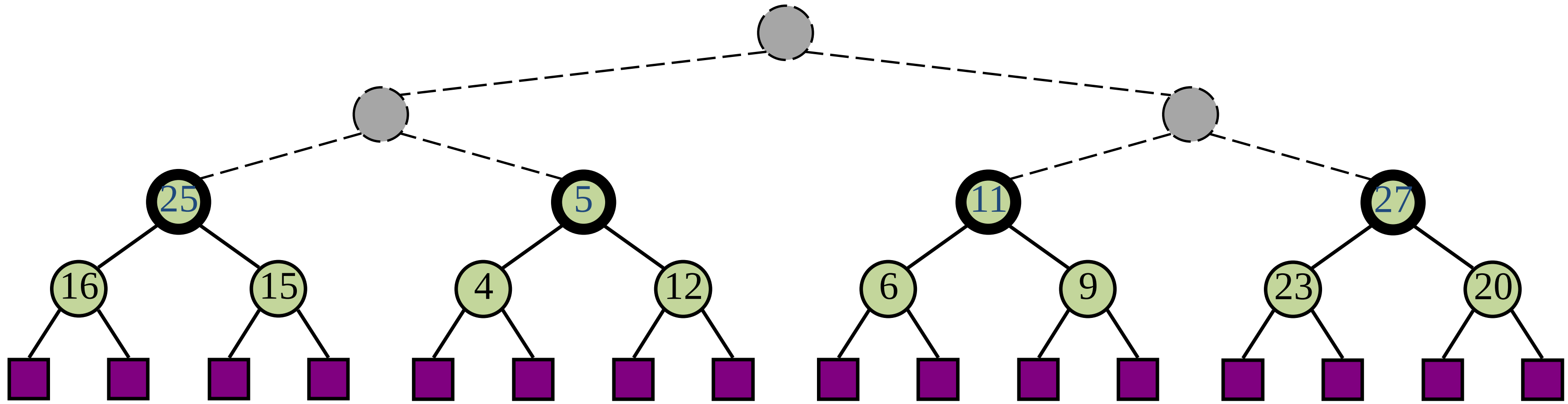


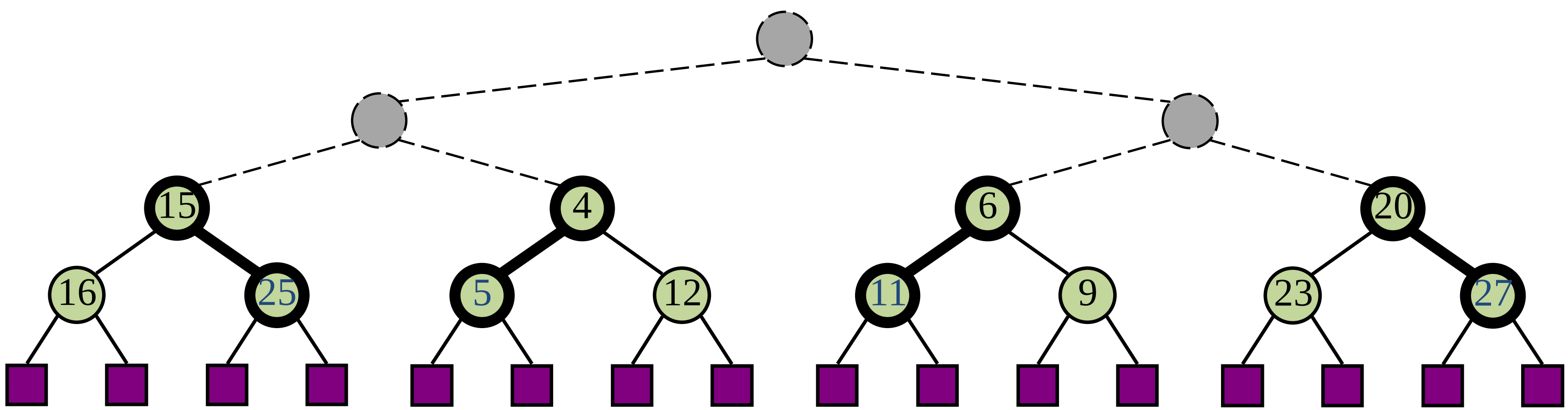
Example



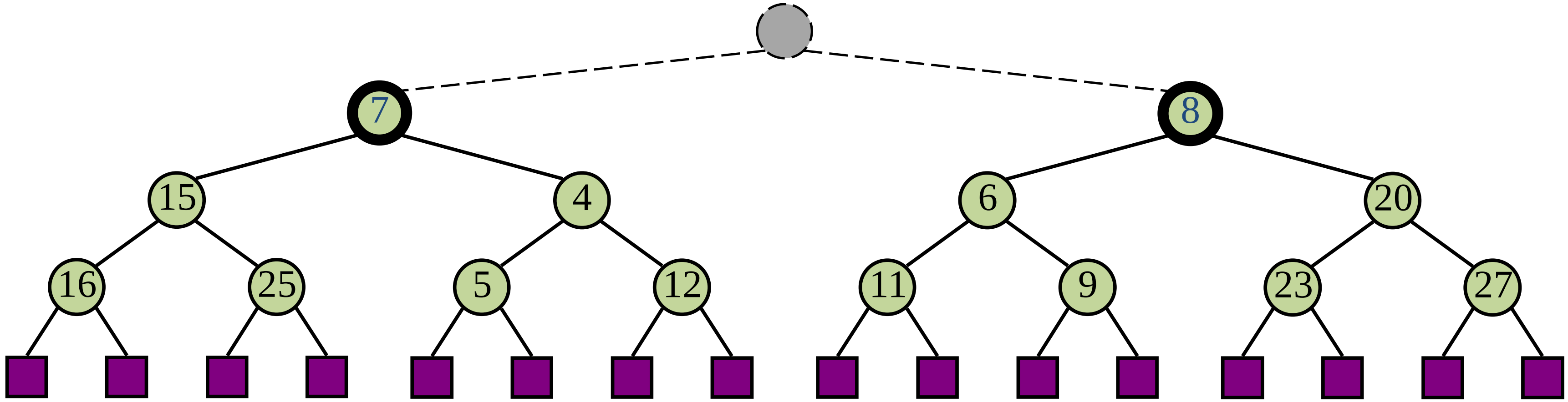


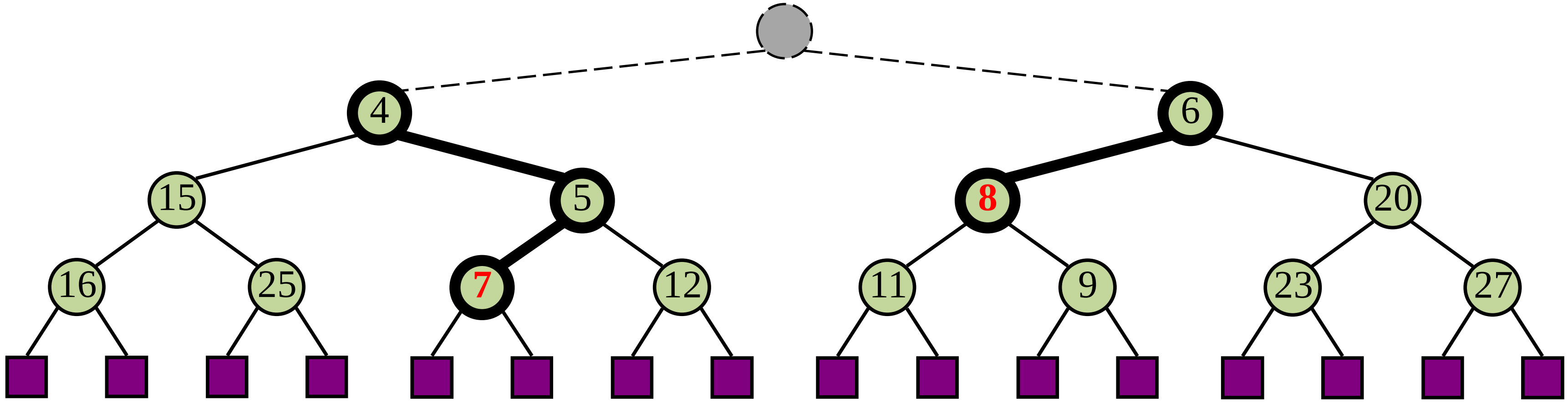
Example (contd.)



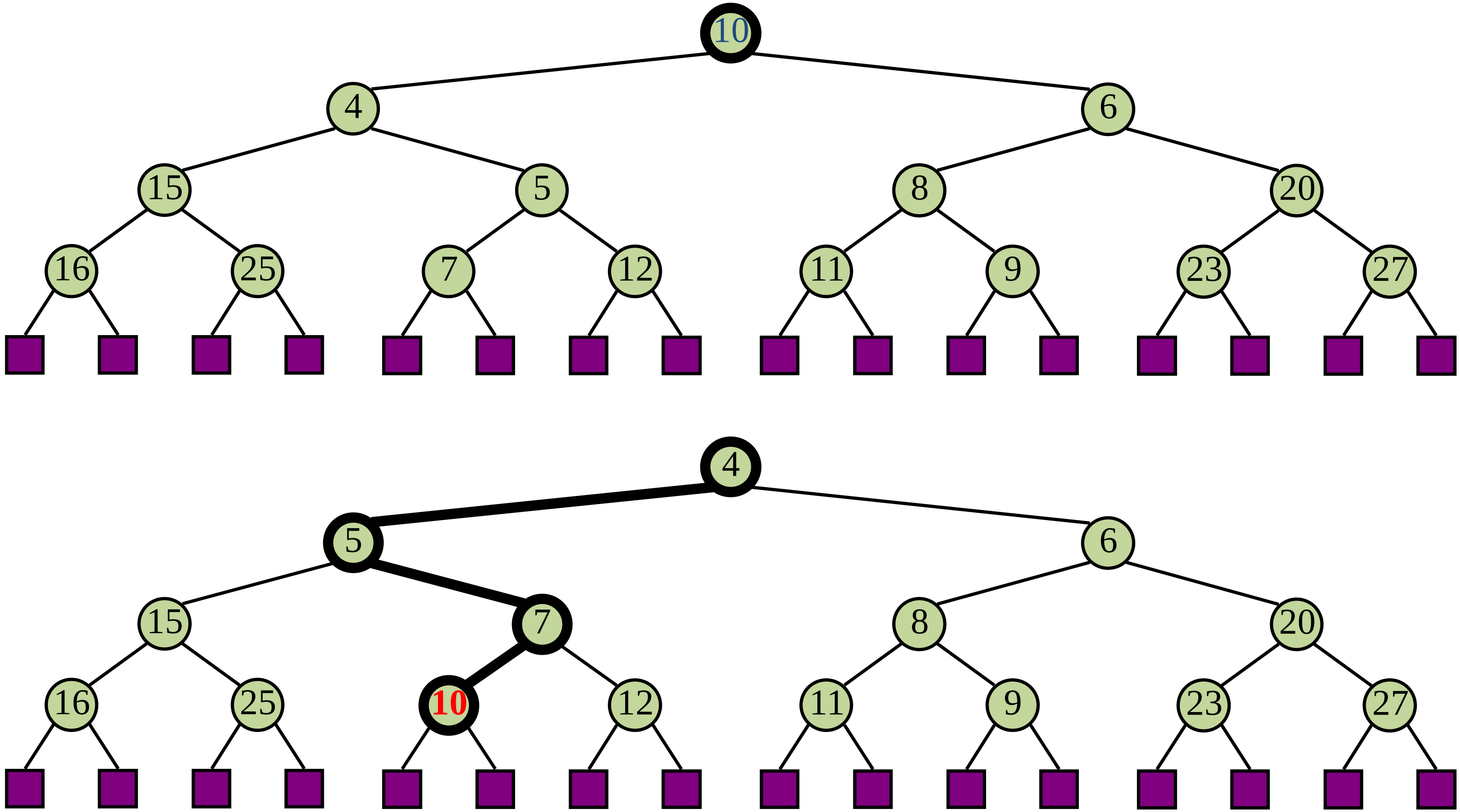


Example (contd.)





Example (end)



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