Algorithms and Data Structures

9th Lecture: Heap

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
S. Semba, Deepika Saxena, Yinghu Zhou, Akila Siriweera

University of Aizu

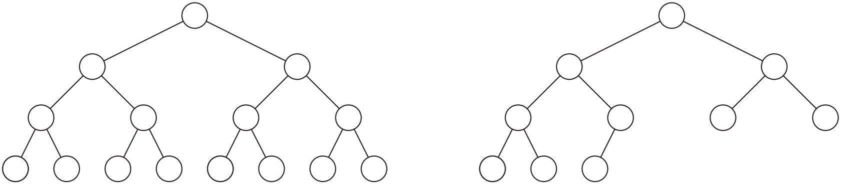
Last Updated: 2025/01/12

Outline

- Heap
- Heap Properties
- Operations on Heaps
- Priority Queues
- Heap Sort Algorithm

Heap (1)

- The (binary) heap data structure is an array object that can be viewed as a **nearly complete binary tree**.



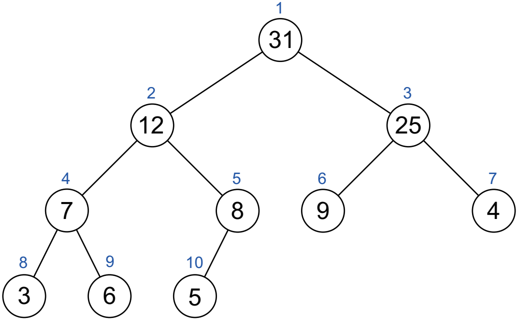
(a) Complete Binary Tree

(b) Nearly Complete Binary Tree

Heap (2)

- Each node of the tree corresponds to an element of the array that stores the value in the node.
- An array A that represents a heap is an object with two attributes:
    - $A.length$, which is the number of elements in the array, and
    - $A.heap\_size$, the number of elements in the heap stored within array A.
- Viewing a heap as a tree, we define the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf.

Heap: An Example

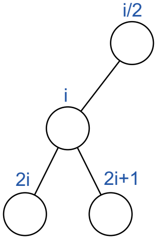


1	2	3	4	5	6	7	8	9	10
31	12	25	7	8	9	4	3	6	5

Operations on Heaps

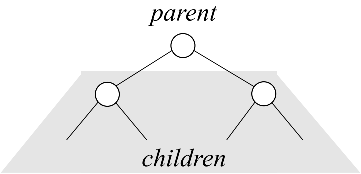
The root of the tree is $A[1]$, and given the index i of a node, the indices of its parent $parent(i)$, left child $left(i)$, and right child $right(i)$ can be computed simply:

```
parent(i)
    return floor(i/2)

left(i)
    return 2i

right(i)
    return 2i + 1
```



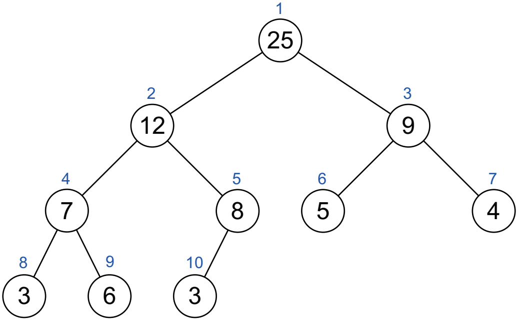
Heap Properties

- There are two kinds of binary heaps:
  - **max-heaps** and
  - **min-heaps**.
- In both kinds, the values in the nodes satisfy a **heap property**.



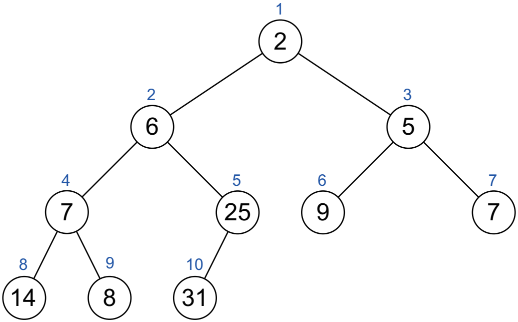
Max-heap

- In a max-heap, the **max-heap property** is that, for every node i other than the root, $A[i] \leq A[parent(i)]$, that is, the value of a node is at most the value of its parent.
- The largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no larger than that contained at the node itself.



Min-heap

- In a min-heap, the **min-heap property** is that, for every node *i* other than the root, $A[parent(i)] \leq A[i]$.
- The smallest element in a min-heap is at the root.

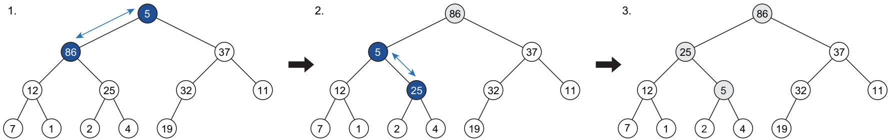


Maintaining the Heap Property

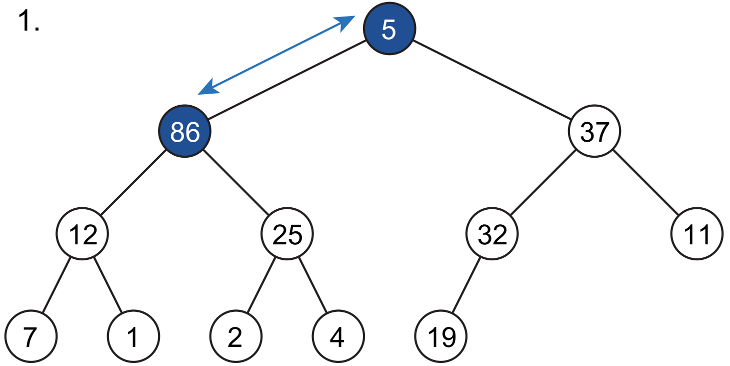
maxHeapify is an important subroutine for manipulating max-heap

```
maxHeapify(A, i)
    l = left(i)
    r = right(i)
    if l <= A.heap_size and A[l] > A[i]
        largest = l
    else
        largest = i
    if r <= A.heap_size and A[r] > A[largest]
        largest = r

    if largest != i
        exchange A[i] and A[largest]
        maxHeapify(A, largest)
```

MaxHeapify: An Example

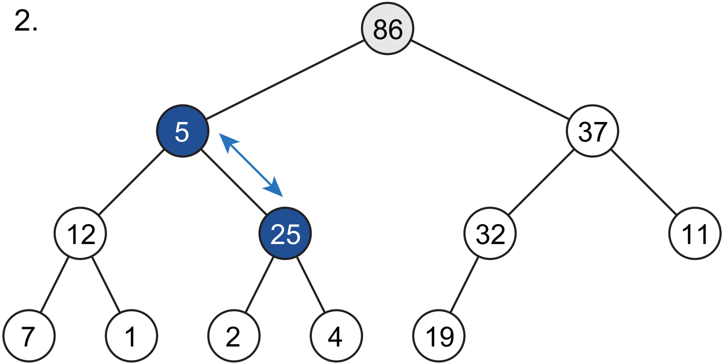


MaxHeapify: Down Heap (1)



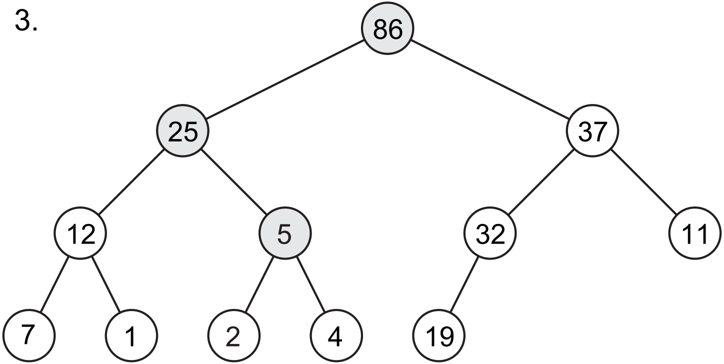
MaxHeapify: Down Heap (2)

2.



MaxHeapify: Down Heap (3)

3.



Building a Heap (1)

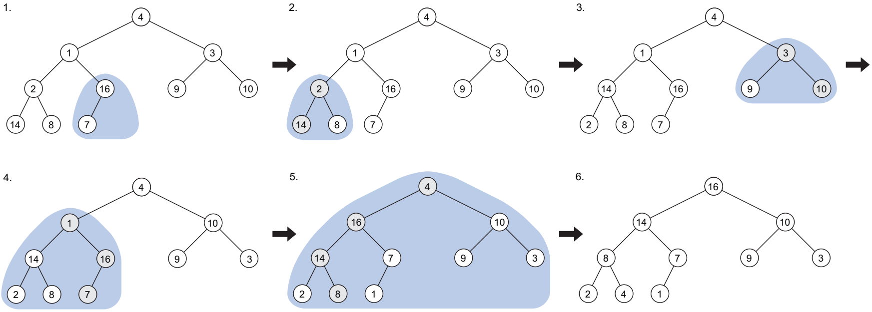
- We use the procedure maxHeapify in a bottom-up manner to convert an array $A[1..n]$, where $n = A.length$, into a max-heap.
- The elements in the subarray $A[floor(n/2) + 1..n]$ are all leaves of the tree, and so each is a 1-element heap to begin with.
- The procedure buildMaxHeap goes through the remaining nodes of the tree and runs maxHeapify on each one.

Building a Heap (2)

```
buildMaxHeap(A)
    A.heap_size = A.length
    for i = floor(A.length/2) down to 1
        maxHeapify(A, i)
```

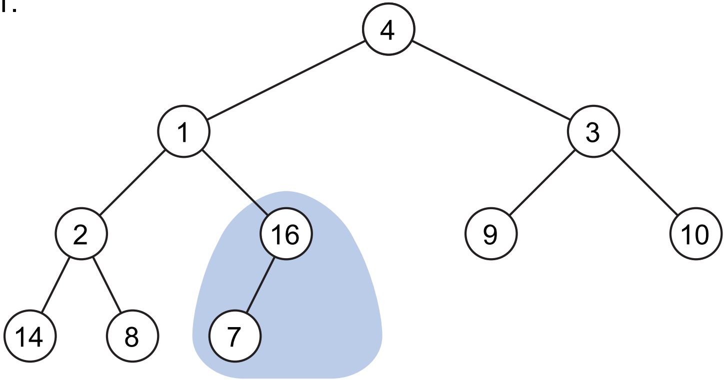
- The time required by maxHeapify is $O(\log n)$.
- We can build a max-heap by the procedure buildMaxHeap in time $O(n)$.
    - Perform maxHeapify to $n/2$ sub-trees with height 1, $n/4$ sub-trees with height 2, ..., 1 sub-tree with height $\log n$, respectively.
    - We obtain $n \times \sum_{k=1}^{\log n} \frac{k}{2^k} = O(n)$

Building a Heap: An Example



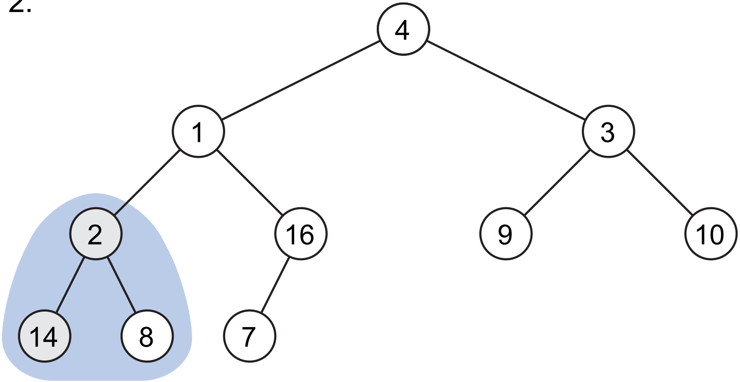
Building a Heap: maxHeapify on a Subtree (1)

1.



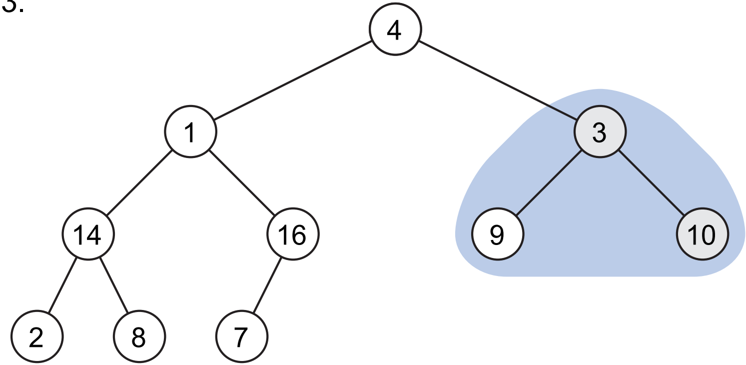
Building a Heap: maxHeapify on a Subtree (2)

2.



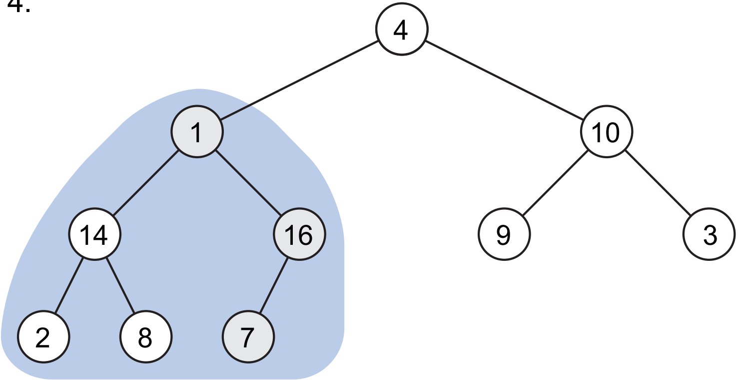
Building a Heap: maxHeapify on a Subtree (3)

3.



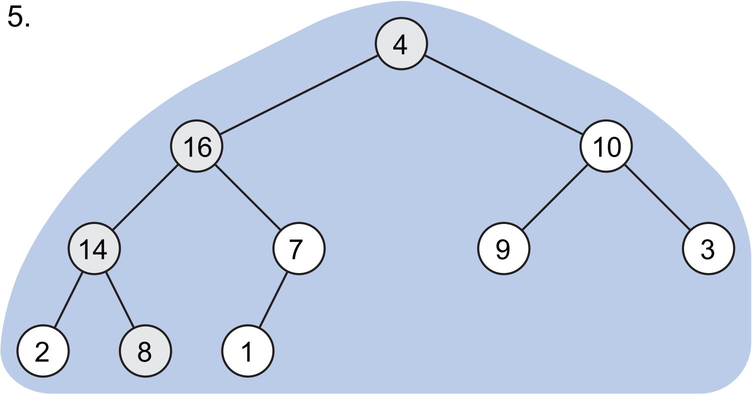
Building a Heap: maxHeapify on a Subtree (4)

4.



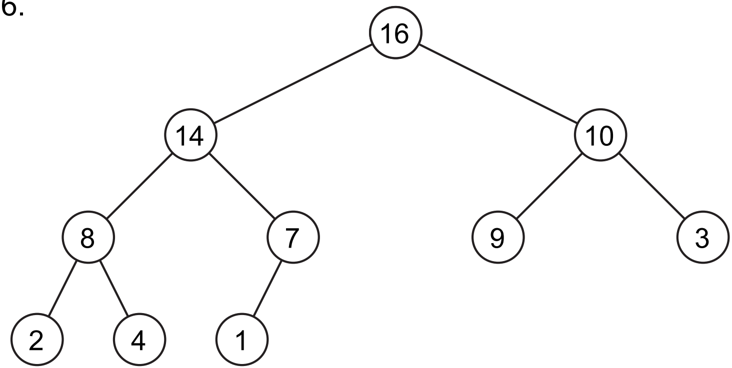
Building a Heap: maxHeapify on a Subtree (5)

5.



Building a Heap: maxHeapify on a Subtree (6)

6.



Priority Queues

- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key.
- A **max-priority queue** supports the following operations.
  - $insert(S, x)$ inserts the element x into the set S.
  - $maximum(S)$ returns the element of S with the largest key.
  - $extractMax(S)$ removes and returns the element of S with the largest key.
  - $increaseKey(S, p, k)$ increases the value of element p's key to the new value k, which is assumed to be the least as large as p's current key value.

HeapMaximum

The procedure heapMaximum implements the *maximum* operation in $O(1)$ time.

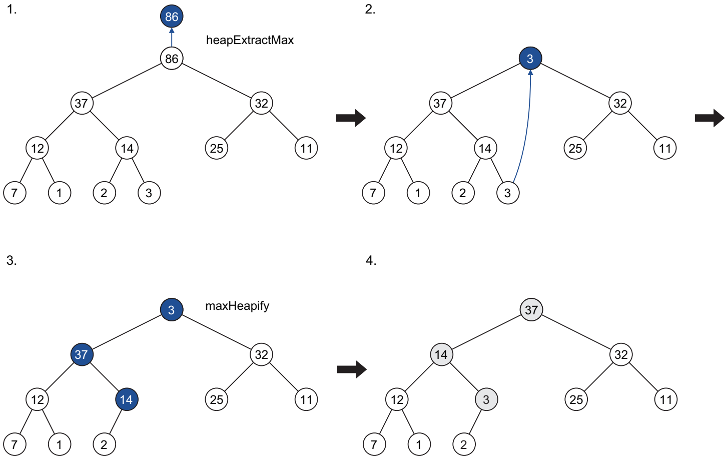
```
heapMaximum(A)
   return A[1]
```

HeapExtractMax

The procedure heapExtractMax implements the *extractMax* operation in $O(\log n)$.

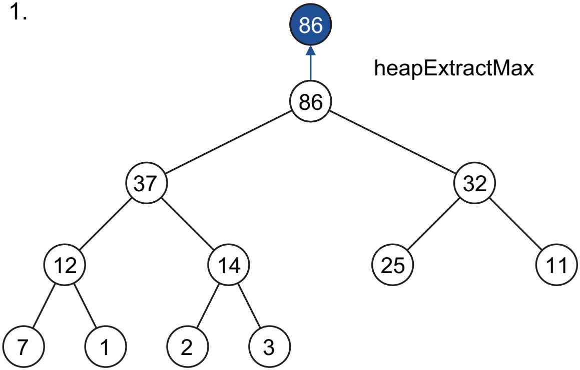
```
heapExtractMax(A)
    if A.heap_size < 1
        output error "heap underflow"
    max = A[1]
    A[1] = A[A.heap_size]
    A.heap_size--
    maxHeapify(A, 1)
    return max
```

HeapExtractMax: An Example



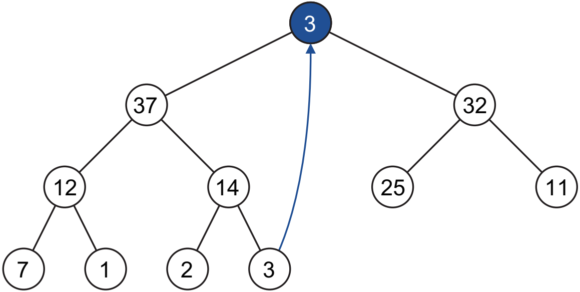
HeapExtractMax (1)

1.



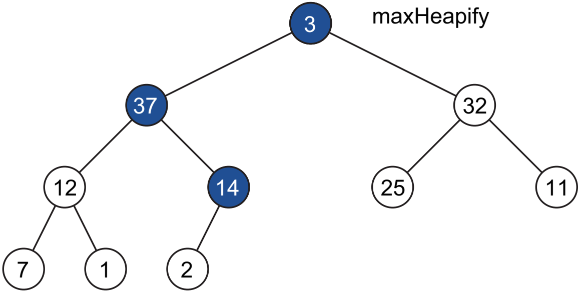
HeapExtractMax (2)

2.



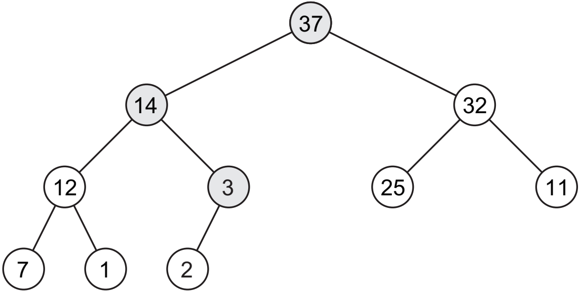
HeapExtractMax (3)

3.



HeapExtractMax (4)

4.



MaxHeapInsert

The procedure maxHeapInsert implements the *insert* operation in $O(\log n)$ time.

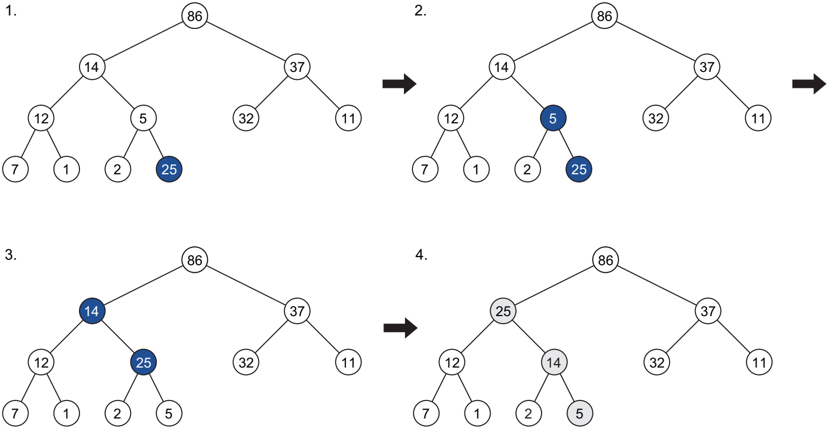
```
maxHeapInsert(A, key)
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = -INF
    heapIncreaseKey(A, A.heap_size, key)
```

HeapIncreaseKey

The procedure heapIncreaseKey implements the *increaseKey* operation in $O(\log n)$ time.

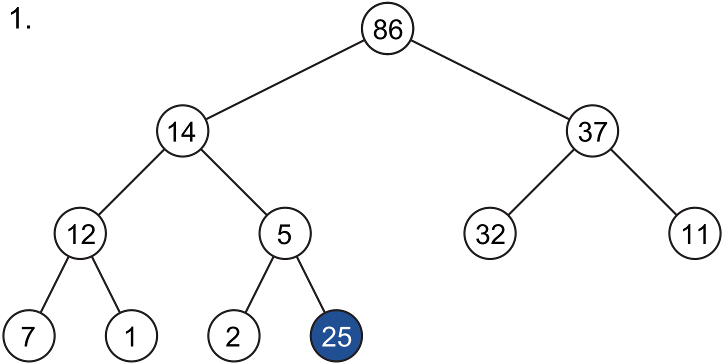
```
heapIncreaseKey(A, i, key)
    if key < A[i]
        output error "new key is smaller than current key"
    A[i] = key
    while i > 1 and A[parent(i)] < A[i]
        exchange A[i] and A[parent(i)]
        i = parent(i)
```

HeapIncreaseKey: An Example



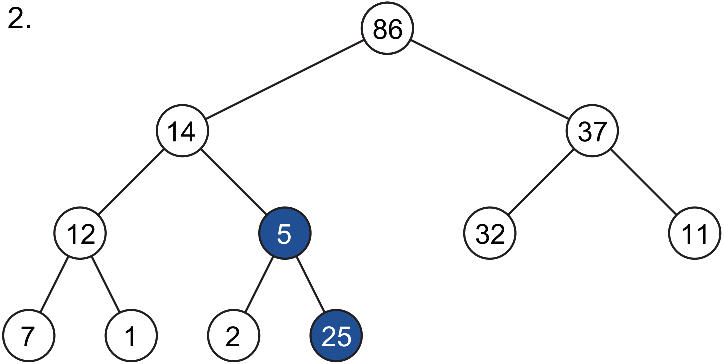
HeapIncreaseKey (1)

1.



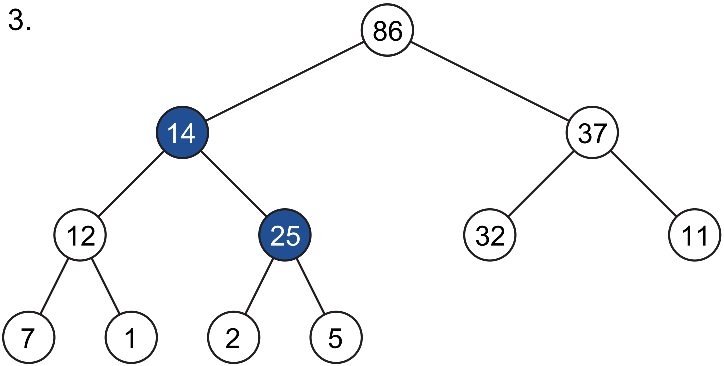
HeapIncreaseKey (2)

2.



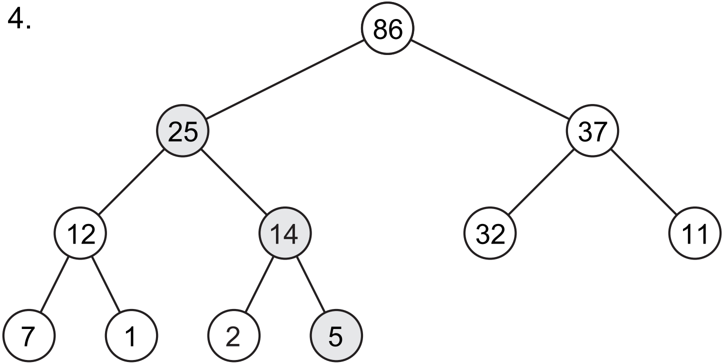
HeapIncreaseKey (3)

3.



HeapIncreaseKey (4)

4.



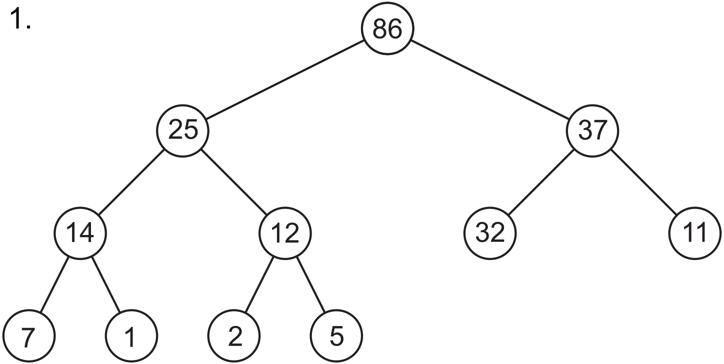
Heap Sort (1)

Here is Heap Sort Algorithm working on the input array $A[1..n]$, where $n = A.length$.

```
heapsort(A)
    buildMaxHeap(A)
    for i = A.length down to 2
        exchange A[1] and A[i]
        A.heap_size--
        maxHeapify(A, 1)
```

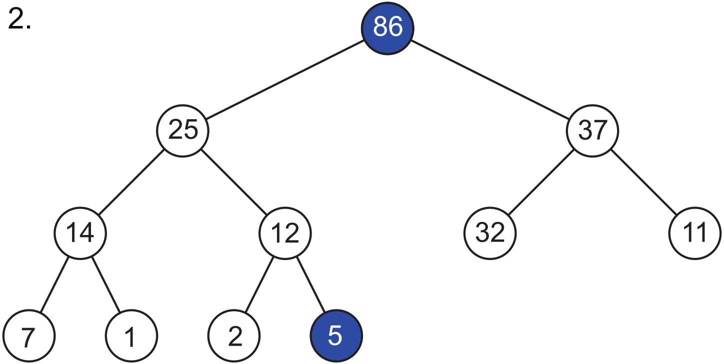
Heap Sort (1)

1.



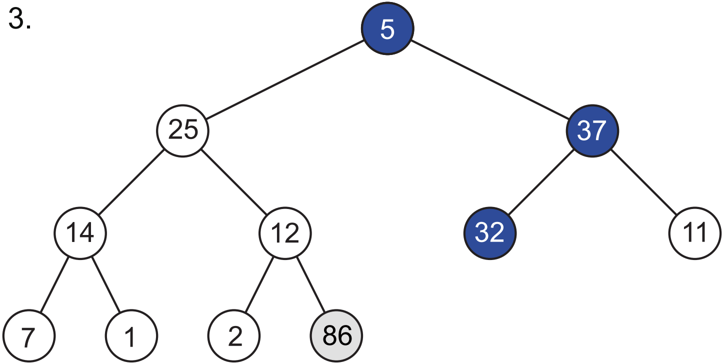
Heap Sort (2)

2.



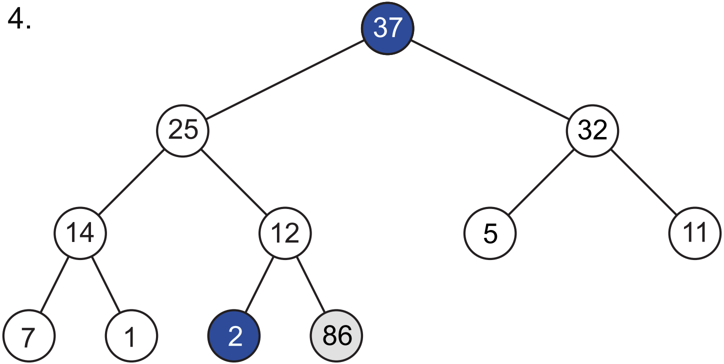
Heap Sort (3)

3.



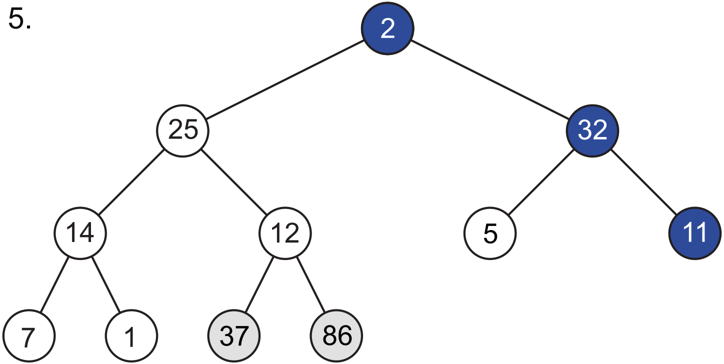
Heap Sort (4)

4.



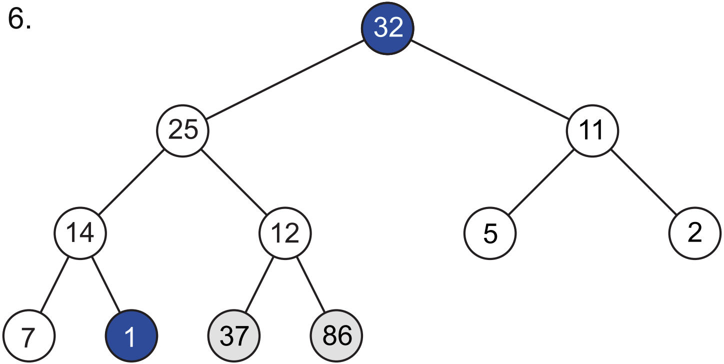
Heap Sort (5)

5.



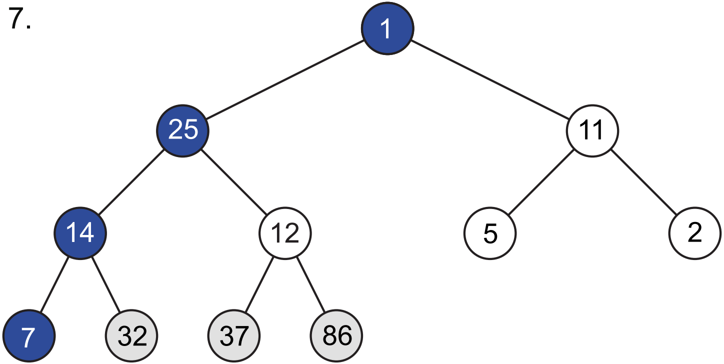
Heap Sort (6)

6.



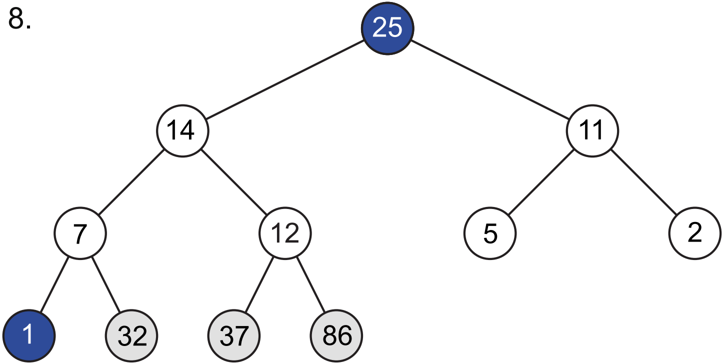
Heap Sort (7)

7.



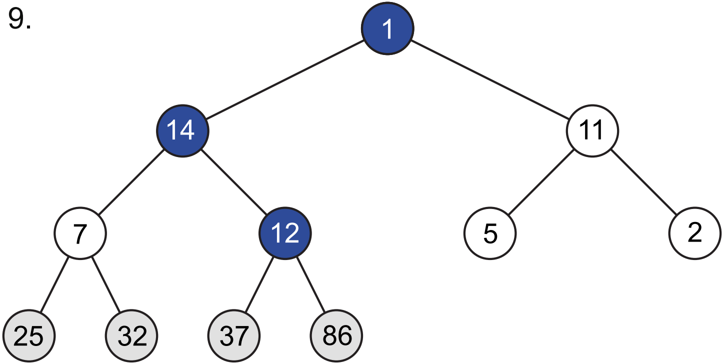
Heap Sort (8)

8.



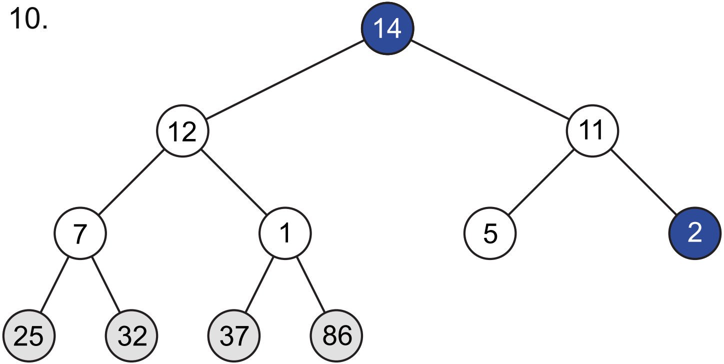
Heap Sort (9)

9.



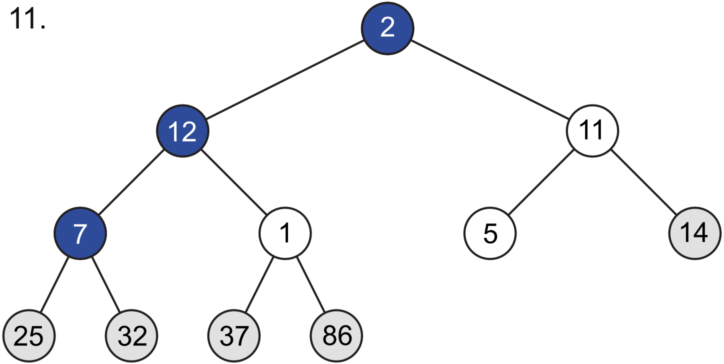
Heap Sort (10)

10.



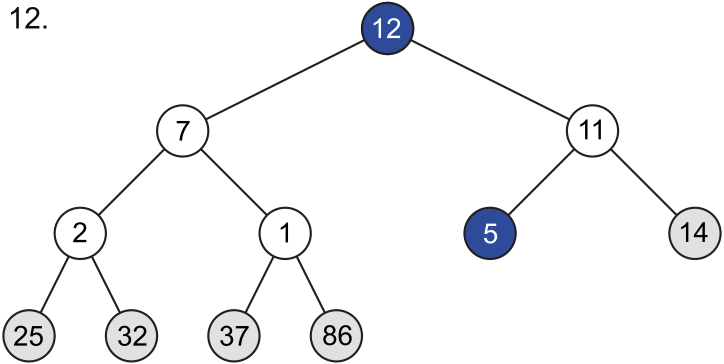
Heap Sort (11)

11.



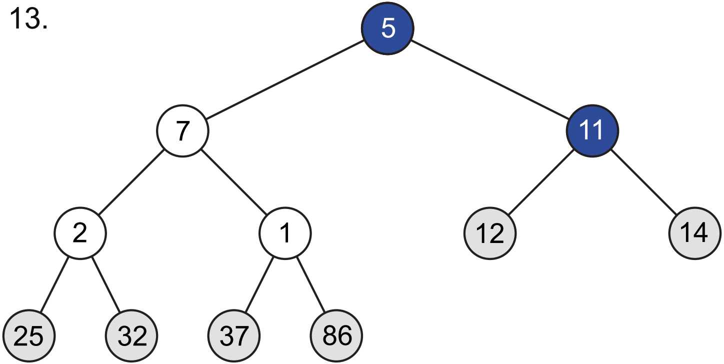
Heap Sort (12)

12.



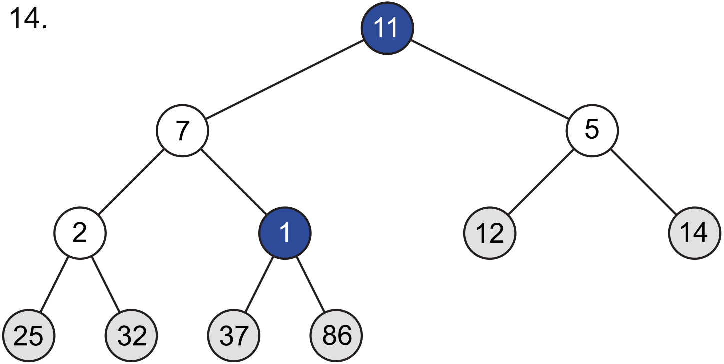
Heap Sort (13)

13.



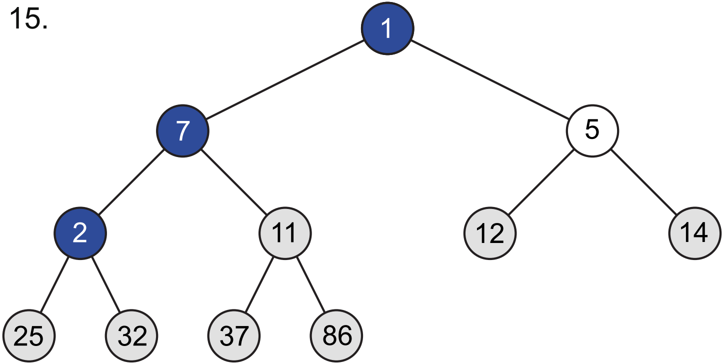
Heap Sort (14)

14.



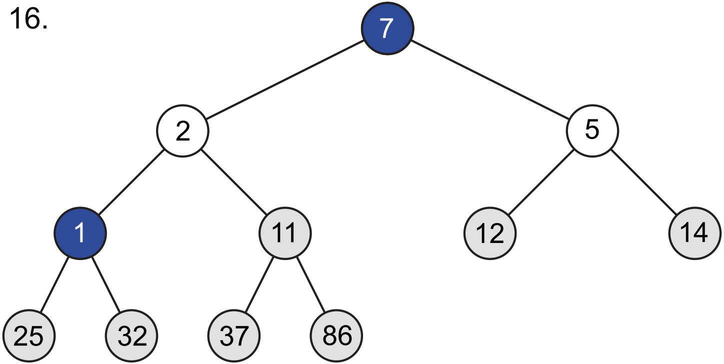
Heap Sort (15)

15.



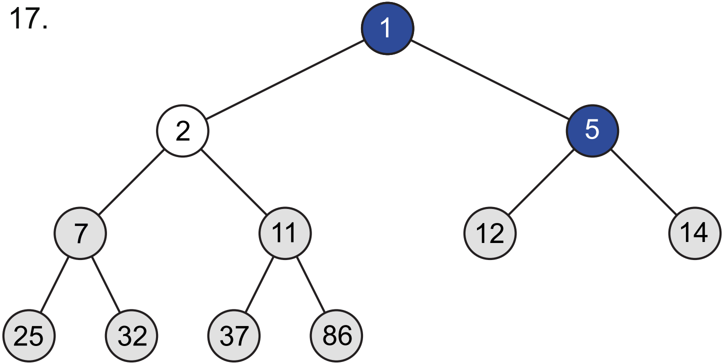
Heap Sort (16)

16.



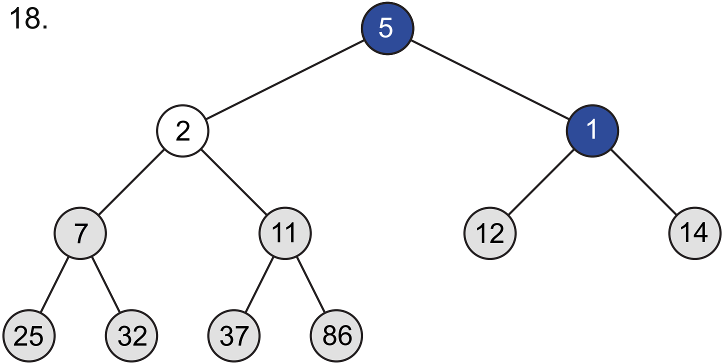
Heap Sort (17)

17.



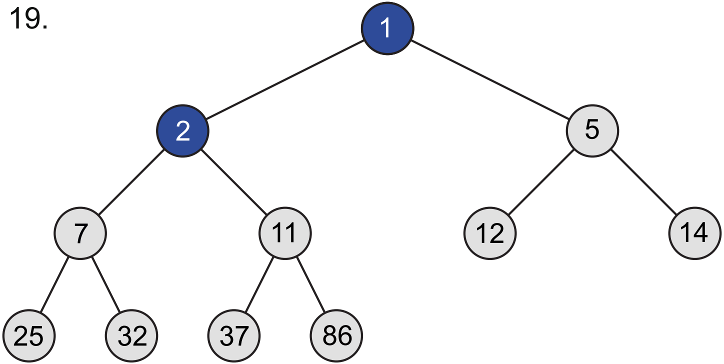
Heap Sort (18)

18.



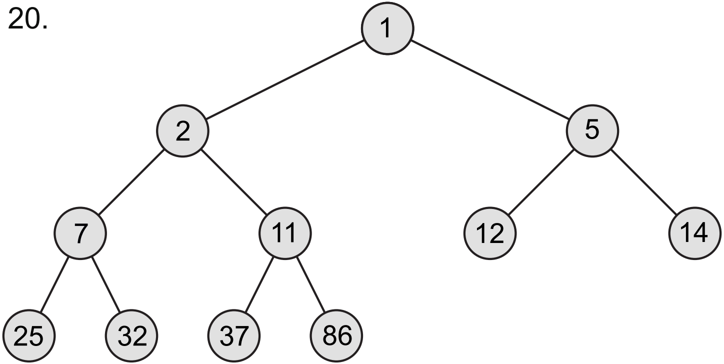
Heap Sort (19)

19.



Heap Sort (20)

20.



Performance and Stability of Heap Sort

- The heapsort procedure takes time $O(n \log n)$, since the call to buildMaxHeap takes time $O(n)$ and each of the $n-1$ calls to maxHeapify takes time $O(\log n)$.
- Heap sort is not a stable sort because it swaps distant elements of the array. Moreover, by its nature, those elements are likely to be far from each other, which may affect execution speed depending on the architecture.

Reference

1. Introduction to Algorithms (third edition), Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2012.