Binary Heap

Motivation

- Heap Sort (CLRS is organized that way!)
- Priority Queue
- Most space efficient data structure

Priority Queue

- “Queue” data structure has a FIFO property
- Some times it is useful to consider **priority**
- Output element with highest priority first

Priority Queue - Major Operations

- Insert
- FindMin (resp. FindMax)
- DeleteMin (resp. DeleteMax)
- DecreaseKey (resp. IncreaseKey)

Priority Queue - Applications[1]

- Dijkstra's shortest path algorithm
- Prim's MST algorithm
- Heapsort
- Online median
- Huffman Encoding
- A* Search (or any Best first search)
- Discrete event simulation
- CPU Scheduling
- ...
- See Wikipedia entry for priority for details

[1] Kleinberg-Tardos Book and Wikipedia

Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert:

Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin:

Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin:

Priority Queue - Candidate Implementations

- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin: $O(1)$
  - DecreaseKey:

Priority Queue - Candidate Implementations

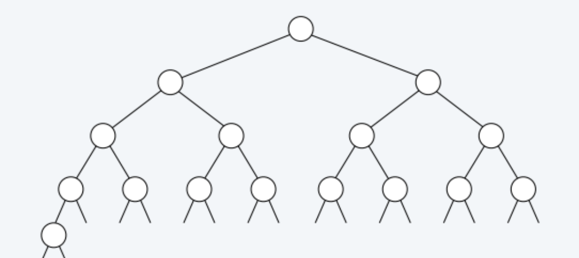
- Assume: for DeleteMin and DecreaseKey, pointer to element is given
- LinkedList
  - Insert: $O(1)$
  - FindMin: $O(n)$
  - DeleteMin: $O(1)$
  - DecreaseKey: $O(1)$
- Binary Heap
  - Insert: $O(\lg n)$
  - FindMin: $O(1)$
  - DeleteMin: $O(\lg n)$
  - DecreaseKey: $O(\lg n)$
- Binomial Heaps, Fibonacci Heaps etc.

Binary Heaps

- Perfect data structure for implementing Priority Queue
- MaxHeap and MinHeap
- We will focus on MaxHeaps in this lecture

Complete Tree[2]

- Perfectly balanced, except for bottom level
- Elements were inserted top-to-bottom and left-to-right



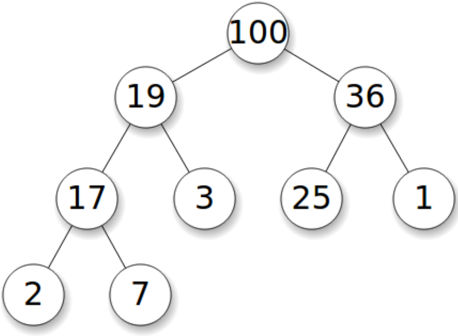
complete tree with n = 16 nodes (height = 4)

[2] http://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/BinomialHeaps.pdf

Heap Property

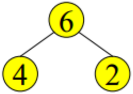
- Heap is a binary tree (**NOT BST**)
- Heap:
  - **Completeness** Property: Heap has restricted structure. It must be a complete binary tree .
  - **Ordering** Property: Relates parent value with that of its children
- MaxHeap property: Value of parent must be greater than **both** its children
- MinHeap property: Value of parent must be less than **both** its children
- Heap with n elements has height $O(\lg n)$

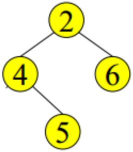
Max Heap Example[3]

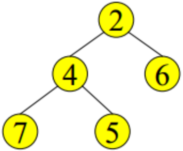


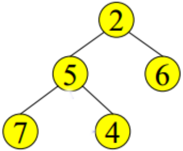
[3] Wikipedia page for Heap

Heap Property[4]









[4] http://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture10.pdf

Major Operations

- Insert
- FindMax
- DeleteMax (aka ExtractMax)
- IncreaseKey

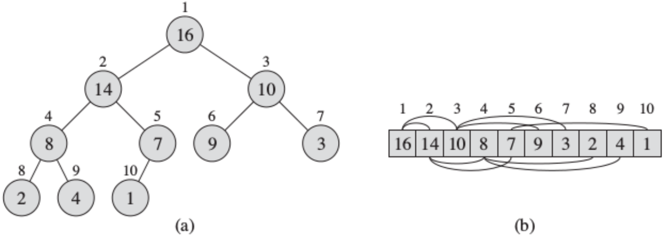
Key Helper Routines

- Max-Heapify (or Min-Heapify)
- Bubble-Up
- Bubble-Down
- Heapify

Representation: Arrays

- Very efficient implementation using arrays
- Possible due to completeness property
- Parent(i): return $\lfloor i/2 \rfloor$
- LeftChild(i): return $2i$
- RightChild(i): return $2i + 1$

Representation: Arrays[5]

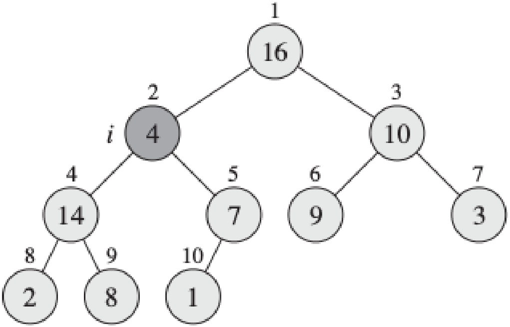


[5] CLRS Fig 6.1

Max-Heapify

- Objective: Maintain heap property
- Invocation: Max-Heapify(A, i)
- Assume: Left(i) and Right(i) are valid max-heaps
- $A[i]$ might violate max-heap property
- Bubble-Down the violation
- **Analysis:** $O(\lg n)$

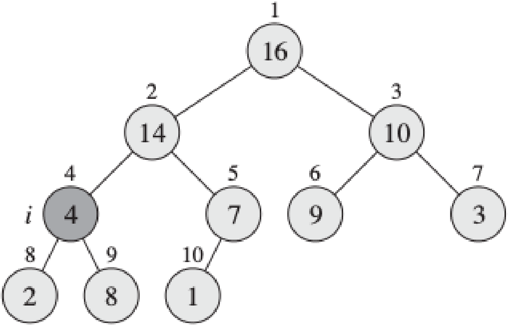
Max-Heapify: Example[6]



(a)

[6]CLRS Fig 6.2

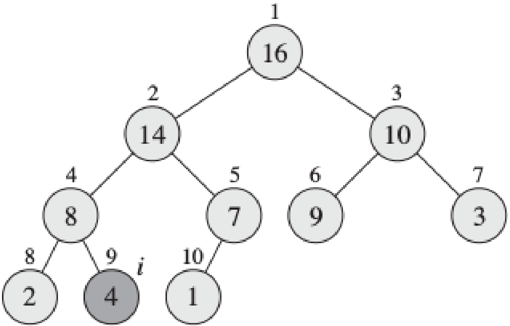
Max-Heapify: Example[7]



(b)

[7]CLRS Fig 6.2

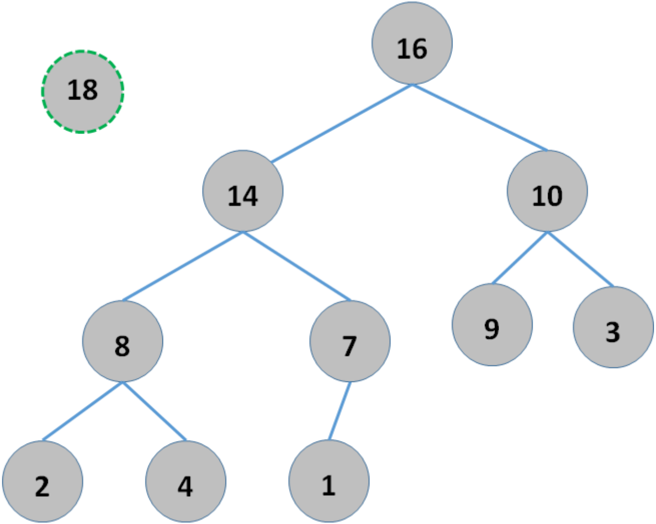
Max-Heapify: Example[8]



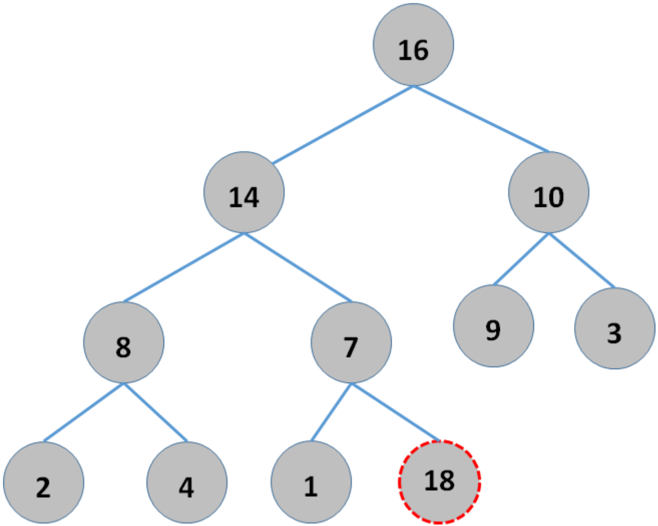
(c)

[8] CLRS Fig 6.2

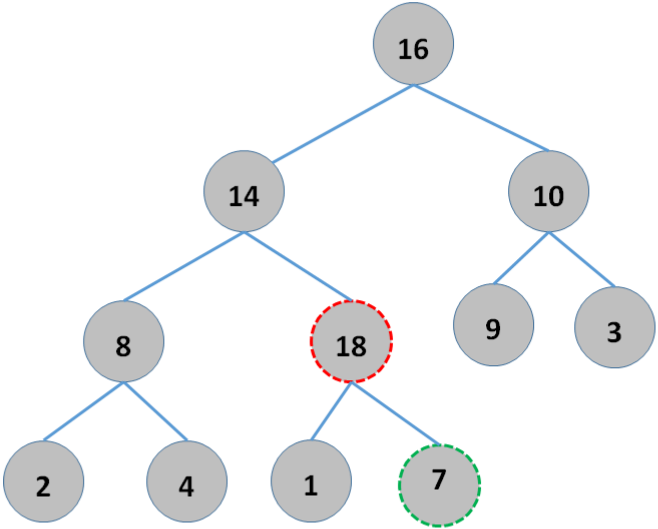
Heap : Insert



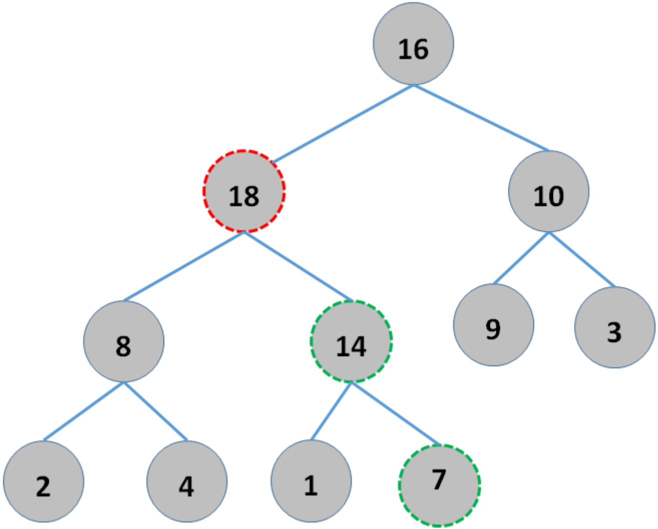
Heap : Insert



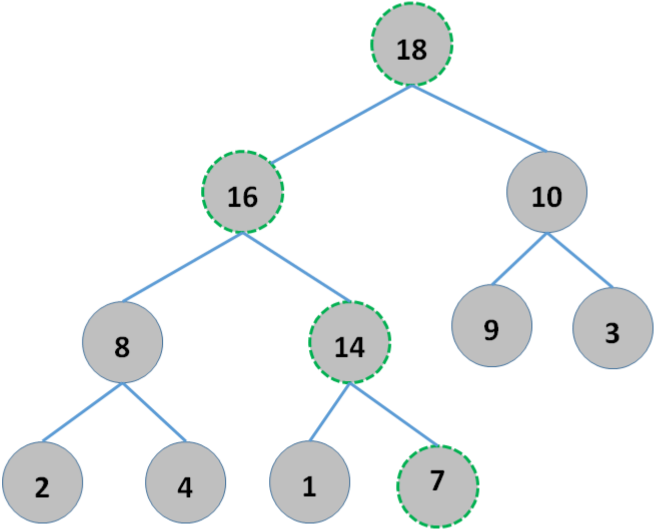
Heap : Insert



Heap : Insert



Heap : Insert



Heap : Insert

- Insert element at first available slot (no completeness property violation!)
- Fix heap property violations by bubbling up the vilolation till it is fixed
- Complexity:

Heap : Insert

- Insert element at first available slot (no completeness property violation!)
- Fix heap property violations by bubbling up the vilolation till it is fixed
- Complexity: $O(\lg n)$

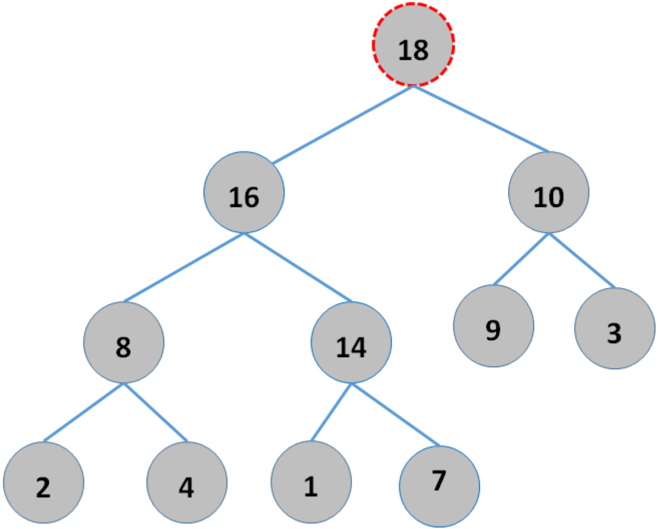
Heap : FindMax

- Look at the root element
- Time complexity: $O(1)$

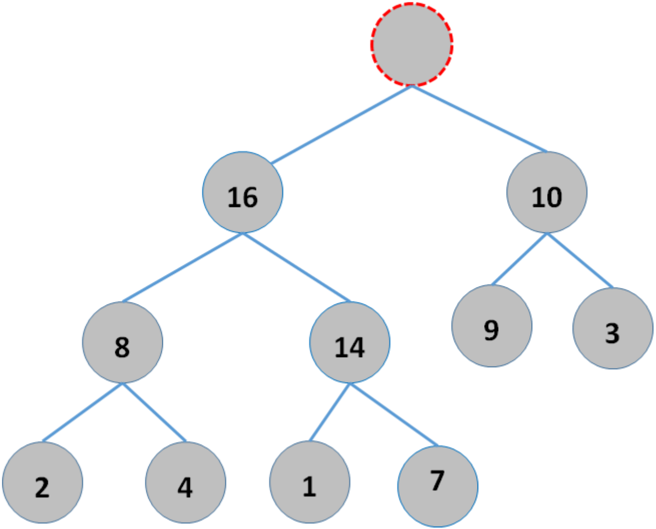
Heap : DeleteMax

- Delete the maximum element (root)
- Fix the heap

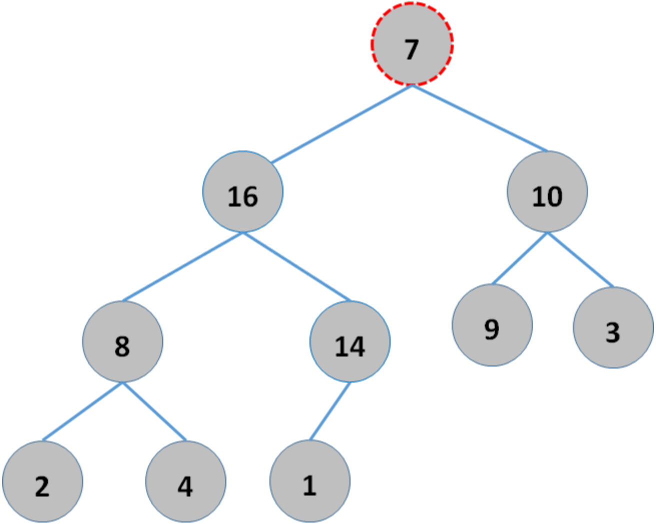
Heap : DeleteMax



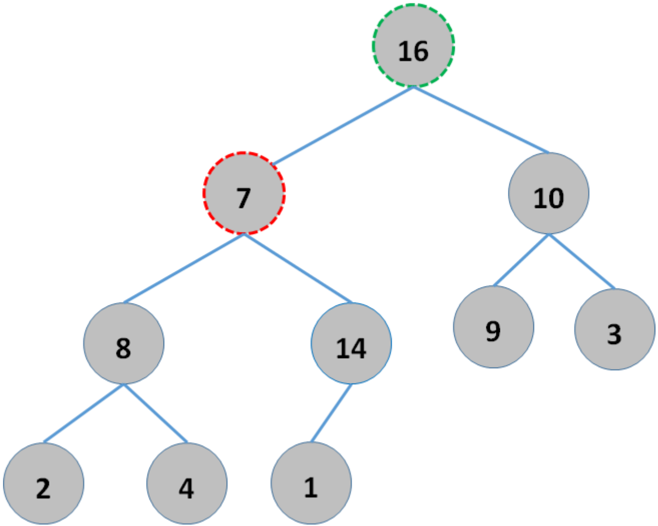
Heap : DeleteMax



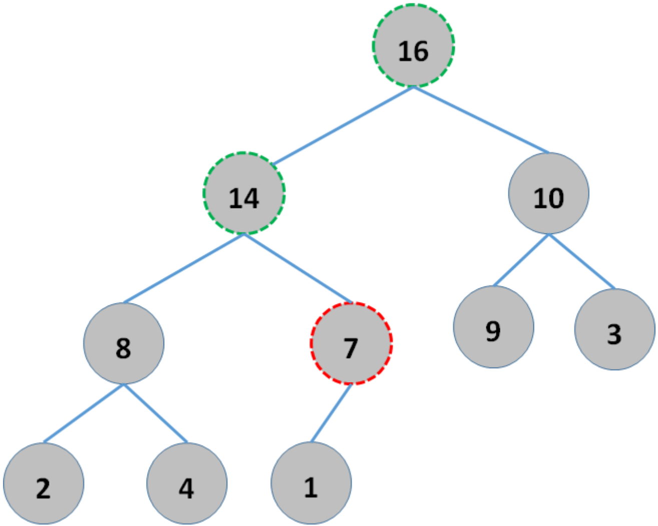
Heap : DeleteMax



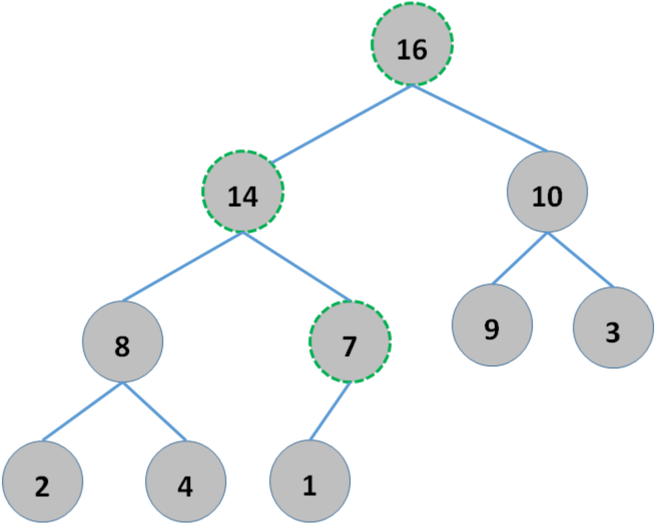
Heap : DeleteMax



Heap : DeleteMax



Heap : DeleteMax



Heap : DeleteMax

- Remove root
- Replace root with last element (does not affect Completeness property)
- Fix heap violations by bubbling it down till it is fixed
- Complexity:

Heap : DeleteMax

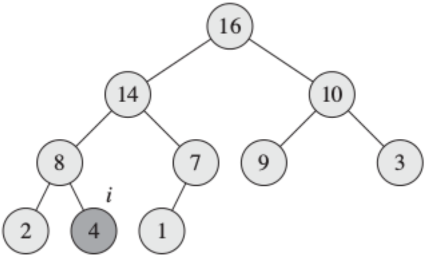
- Remove root
- Replace root with last element (does not affect Completeness property)
- Fix heap violations by bubbling it down till it is fixed
- Complexity: $O(\lg n)$

Heap : IncreaseKey

- Given a node, increase its priority to a new, higher value
- Fix heap property violations

Heap : IncreaseKey[9]

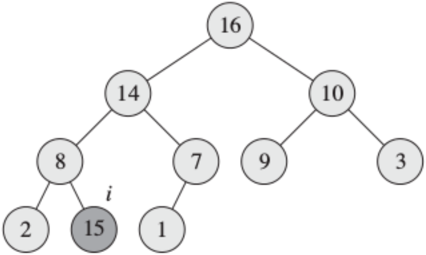
IncreaseKey: Increase value of 4 to 15



(a)

[9]CLRS Fig 6.5

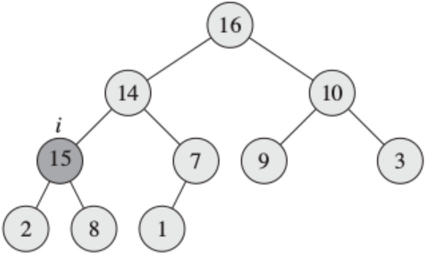
Heap : IncreaseKey[10]



(b)

[10]CLRS Fig 6.5

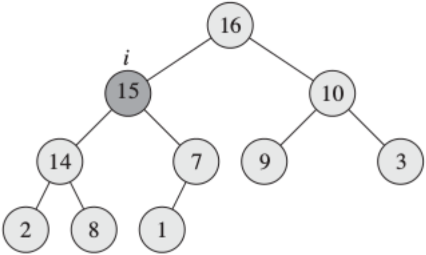
Heap : IncreaseKey[11]



(c)

[11] CLRS Fig 6.5

Heap : IncreaseKey[12]



(d)

[12] CLRS Fig 6.5

Heap : IncreaseKey

- Update element
- Fix heap violations by bubbling it up till it is fixed
- Complexity:

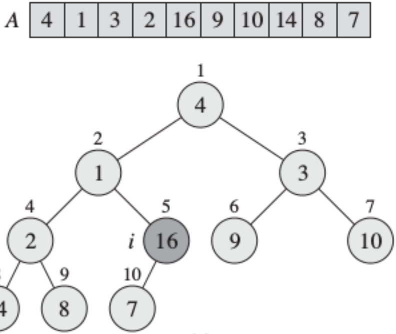
Heap : IncreaseKey

- Update element
- Fix heap violations by bubbling it up till it is fixed
- Complexity: $O(\lg n)$

Build-Max-Heap

- Given an array A, convert it to a max-heap
- $A.length$: Length of the array
- $A.heapSize$: Elements from $1 \dots$ A.heapSize form a heap
- Build-Max-Heap(A):
  - A.heapSize = A.length
  - for $i = \lfloor A.length/2 \rfloor$ down to 1
    Max-Heapify(A, i)
- **Analysis:** $O(n)$ (See book for details)

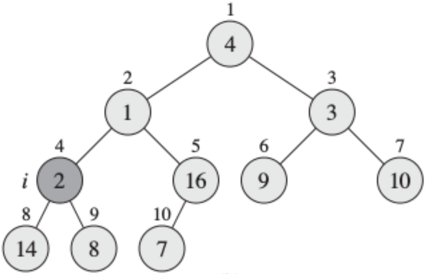
Build-Max-Heap : Example [13]



(a)

[13] CLRS Fig 6.3

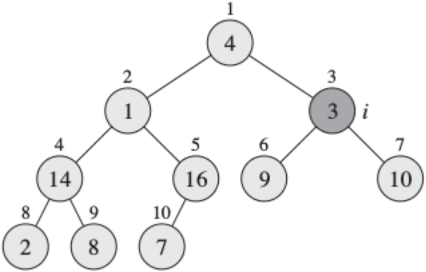
Build-Max-Heap : Example [14]



(b)

[14] CLRS Fig 6.3

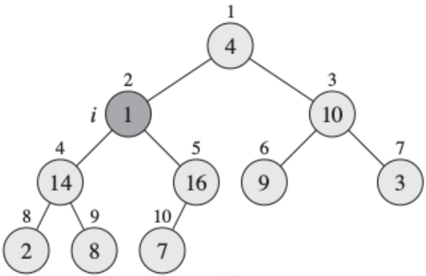
Build-Max-Heap : Example [15]



(c)

[15] CLRS Fig 6.3

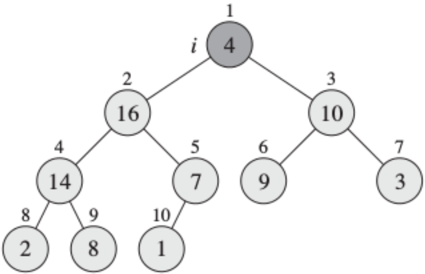
Build-Max-Heap : Example [16]



(d)

[16] CLRS Fig 6.3

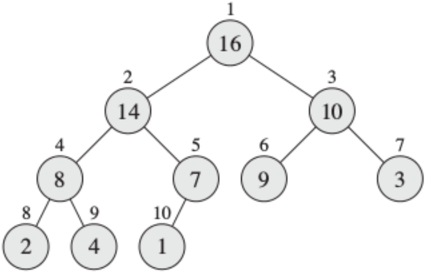
Build-Max-Heap : Example [17]



(e)

[17] CLRS Fig 6.3

Build-Max-Heap : Example [18]



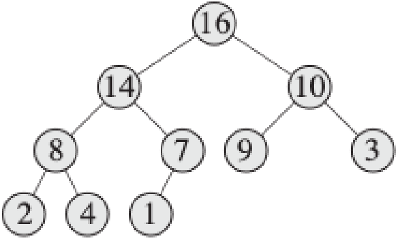
(f)

[18] CLRS Fig 6.3

HeapSort

```
HeapSort(A):
    Build-Max-Heap(A)
    for i = A.length down to 2
        Exchange A[1] with A[i]
        A.heapSize = A.heapSize - 1
        Max-Heapify(A, 1)
```

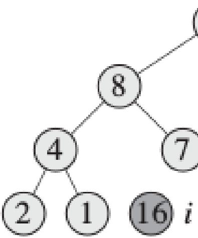
Heap Sort: Example[19]



(a)

[19] CLRS Fig 6.4

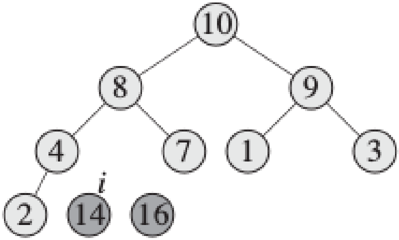
Heap Sort: Example[20]



(b)

[20] CLRS Fig 6.4

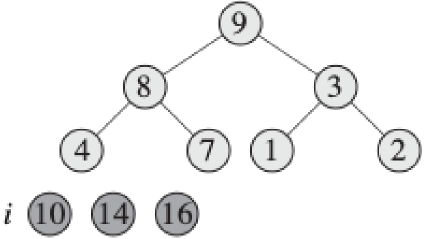
Heap Sort: Example[21]



(c)

[21] CLRS Fig 6.4

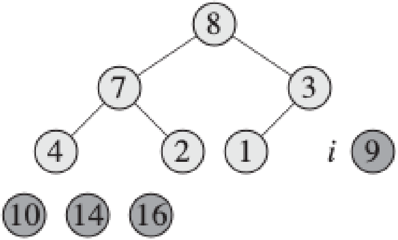
Heap Sort: Example[22]



(d)

[22] CLRS Fig 6.4

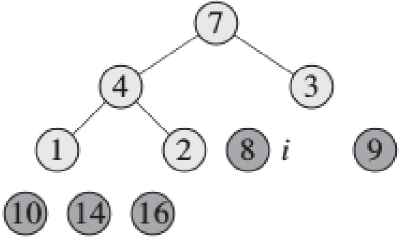
Heap Sort: Example[23]



(e)

[23] CLRS Fig 6.4

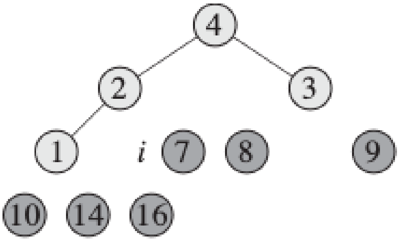
Heap Sort: Example[24]



(f)

[24] CLRS Fig 6.4

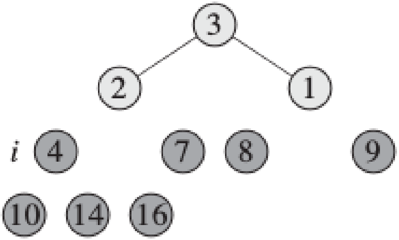
Heap Sort: Example[25]



(g)

[25] CLRS Fig 6.4

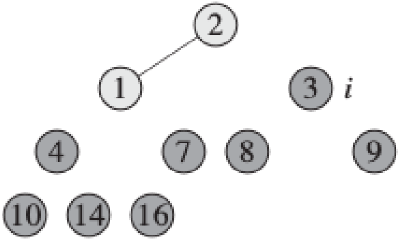
Heap Sort: Example[26]



(h)

[26] CLRS Fig 6.4

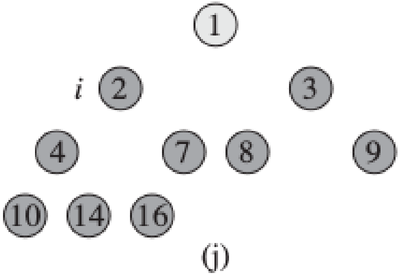
Heap Sort: Example[27]



(i)

[27] CLRS Fig 6.4

Heap Sort: Example[28]



[28] CLRS Fig 6.4

Heap Sort: Example[29]

A	1	2	3	4	7	8	9	10	14	16

[29] CLRS Fig 6.4

HeapSort: Analysis

- Operations:
  - Build-Max-Heap:

HeapSort: Analysis

- Operations:
  - Build-Max-Heap: $O(n)$
  - n Max-Heapify:

HeapSort: Analysis

- Operations:
  - Build-Max-Heap: $O(n)$
  - n Max-Heapify: $n \times \lg n = O(n \lg n)$
  - Complexity: $O(n) + O(n \lg n) = O(n \lg n)$

HeapSort

- Very efficient in practice - often competitive with QuickSort
- In-Place but not stable (why?)
- Requires constant extra space
- Best, average and worst case complexity is $O(n \lg n)$ (unlike Quicksort)