Heap Data Structure

Heapsort, Priority Queue

Definition

- A heap can be viewed as a nearly complete binary tree.
- Each node in the heap satisfies the *heap property*
- A heap could be of two types: *min-heap and max-heap*
- For a max-heap, the heap property states that the value at each node is at most as large as its parent
- For min-heap, the heap property is satisfied if the value at each node is at least as large as its parent
- Heaps are used for sorting and for implementing priority queue

Types of Binary Trees

- Full Binary tree: each node is either of degree 0 or 2. There is no deg-1 node
- Complete Binary Tree: all leaves are at the same depth and all internal nodes are of degree 2
- Nearly Complete Binary tree: A full binary tree where each node is either of degree 0 or 2 accept possibly at the last-but-one level where the nodes might have deg-1 such that the positional information is preserved
- i.e. if the nodes are numbered from left to right then the numbering does not change in a full binary tree and a complete binary tree

Array elements as Heap

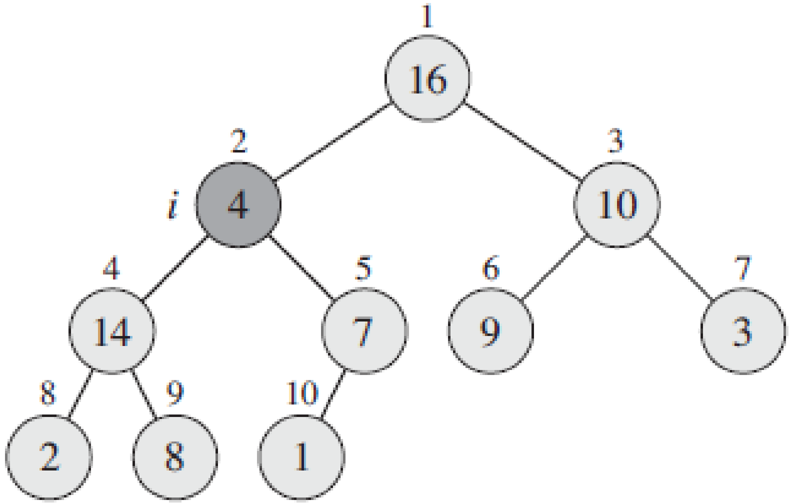
- Array elements can be represented in the form of a heap
- The array elements can be viewed as forming a nearly complete binary tree.
- For a given array element indexed at i we can compute the index of its parents and children as:
  - Parent(i) = $\lfloor i/2 \rfloor$
  - Left_child(i) = $2i$
  - Right_child(i) = $2i + 1$
- For an array A of size A.length, the size of the heap is defined as A.heapsize and we have $1 \leq A.heapsize \leq A.length$

Heap Procedures

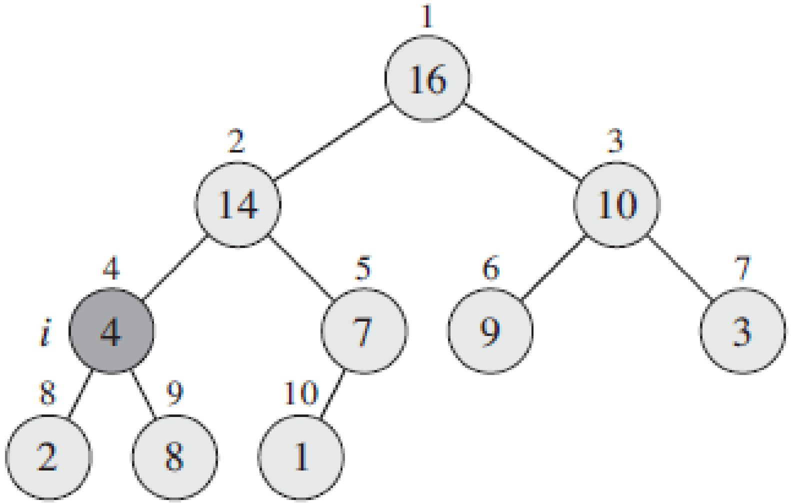
- *Max-Heapify*: runs in O(lgn) time, used to maintain the heap property
- *Build-Max-Heap*: runs in O(n) time, produces a max-heap from an unordered array
- *Heapsort*: runs in O(nlgn) time, sorts an array *in place*
- *Max-Heap-Insert, Heap-Extract-Max, Heap-Increase-Key, Heap-Maximum* procedures, run in O(lgn) time, allow priority queue to be implemented using heap data structure

Maintaining the Heap property

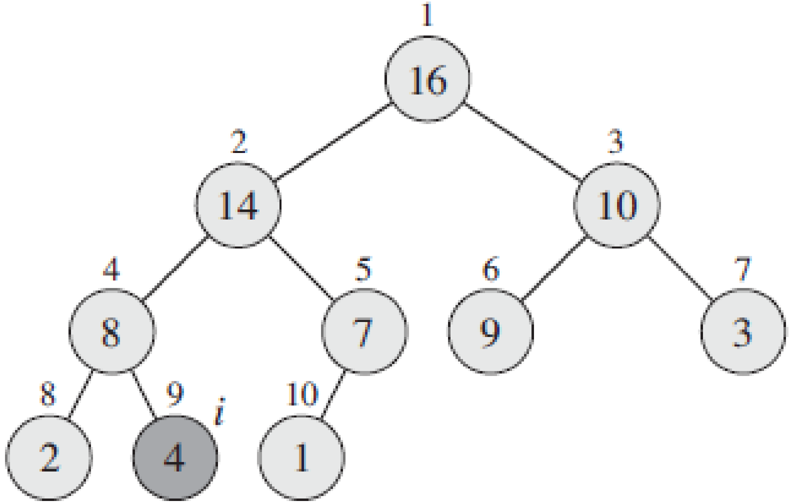
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
        largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
        largest = r
if largest ≠ i
        exchange A[i] with A[largest]
        MAX-HEAPIFY(A, largest)
```


1
16
2
i
4
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1

(a)


1
16
2
14
3
10
4
i
4
5
7
6
9
7
3
8
2
9
8
10
1

(b)


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
i
4
10
1

(c)

Complexity Analysis

- Steps 1 to 9 take constant time
- Step 10 is a recursive call to Max-Heapify
- In order to write the recurrence, we need to find the maximum size of a sub-tree
- Suppose the heap has n nodes in total. Since, heap is a nearly complete binary tree, the left and right-subtrees can have at most a difference of 1 in their heights

- Thus, we get the following expression:

$$1 + 2^{h+1} - 1 + 2^{h+2} - 1 = n$$

$$\Rightarrow 2^{h+1} \quad (2 + 1) - 1 = n$$

$$\Rightarrow 2^{h+1} = \frac{n+1}{3} \approx \frac{n}{3}$$

$$\Rightarrow 2^{h+2} \approx \frac{2n}{3}$$

- Thus, the maximum height of the subtree could be 2n/3

- We can write the recurrence as below:

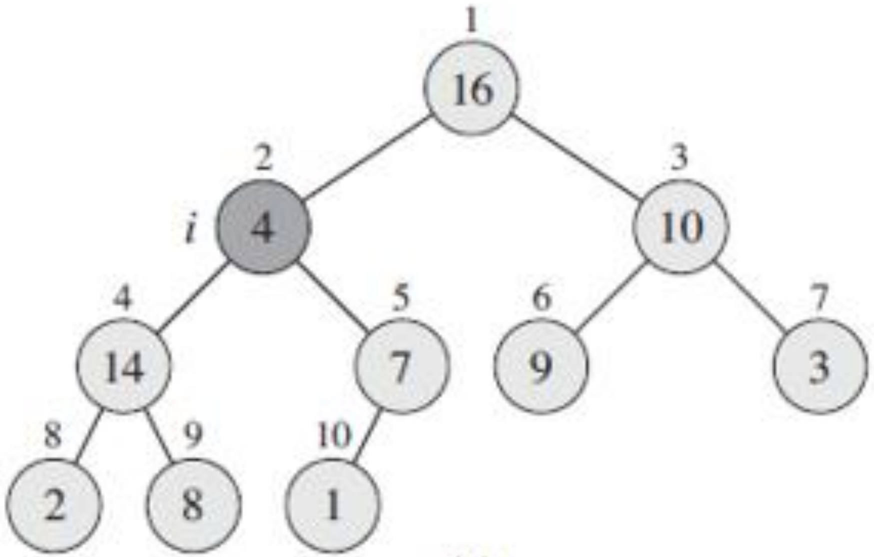
$$T(n) = T\left(\frac{2n}{3}\right) + \theta(1)$$

Using the case 2 of Master method, the solution to the above recurrence is $T(n) = O(lgn)$

Thus, the Heapify method runs in O(lgn) time

Exercise

- Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots n$



(a)

Exercise

- What is the effect of calling MAX-HEAPIFY (A,i) for i > A.*heap-size/*2?

Exercise

1. Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$

- Worst case occurs when the Max-Heapify is called along the longest path from the root to the leaf
- In this case the running time will be bounded by the height of the heap i.e. $\Omega(\lg n)$

Exercise

- Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

Proof: Maximum nodes with height h

- All the nodes from $\lfloor n/2 \rfloor + 1, \ldots n$ are leaf nodes
- Therefore, no. of nodes with height = 0 => $\lceil n/2 \rceil$
- No. of nodes with height = 1 will be half of this value => $\lceil n/2 \rceil \times \frac{1}{2}$
- No. of nodes with height = 2 will be half of this value => $\lceil n/2 \rceil \times \frac{1}{2^2}$

- Maximum no. of nodes with height h $= \left\lceil \frac{n}{2^{h+1}} \right\rceil$

Building a Heap

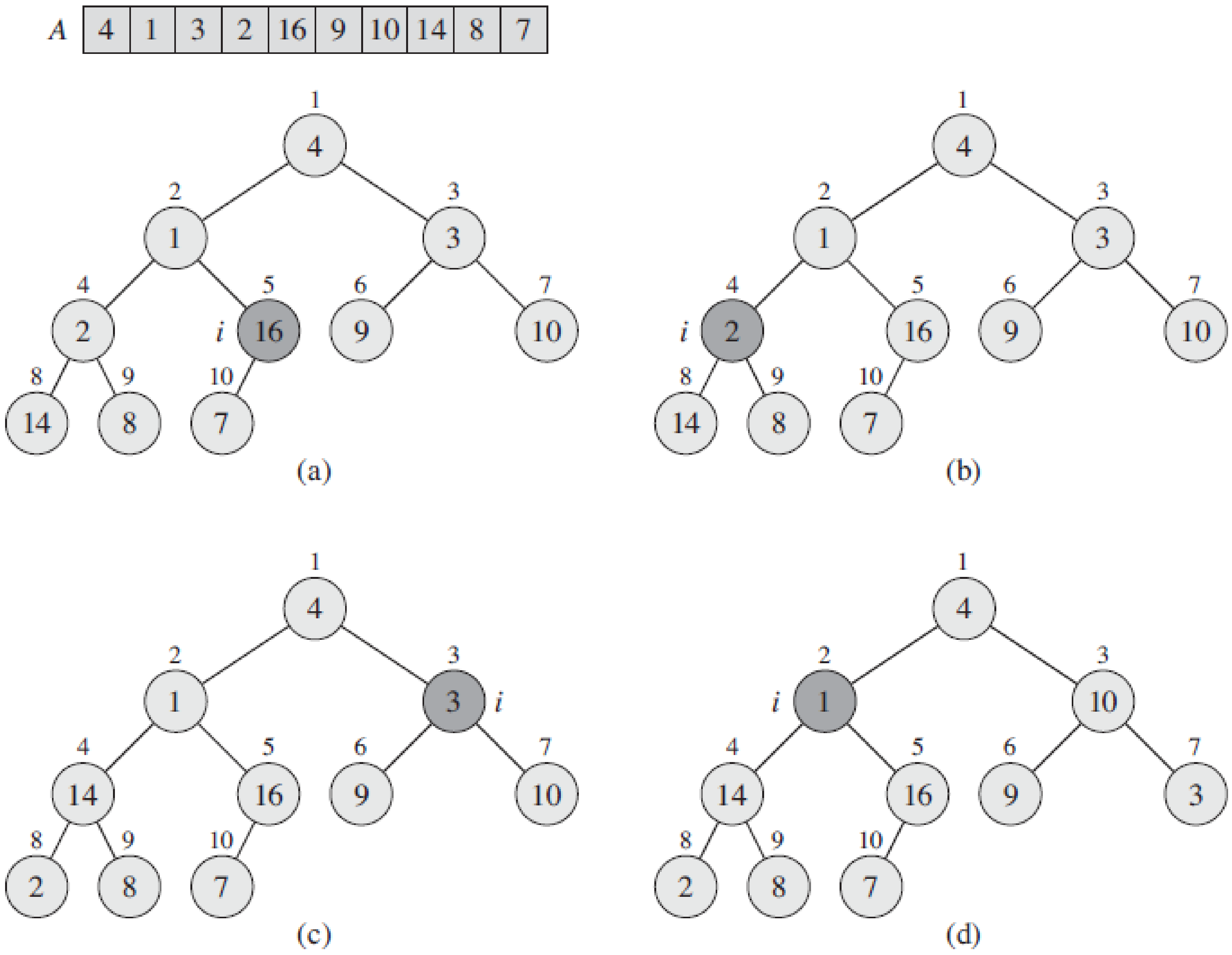
- Elements of an array can be built-up into a max-heap
- We use the procedure Build-max-heap for this
- All elements of the array from $\lfloor n/2 \rfloor + 1 \; upto \; n$ will comprise the leaf nodes and therefore each is a heap in itself
- In our algorithm, we will add higher level node one by one and use the heapify procedure to maintain the heap property as we add elements

Build-max-Heap Procedure

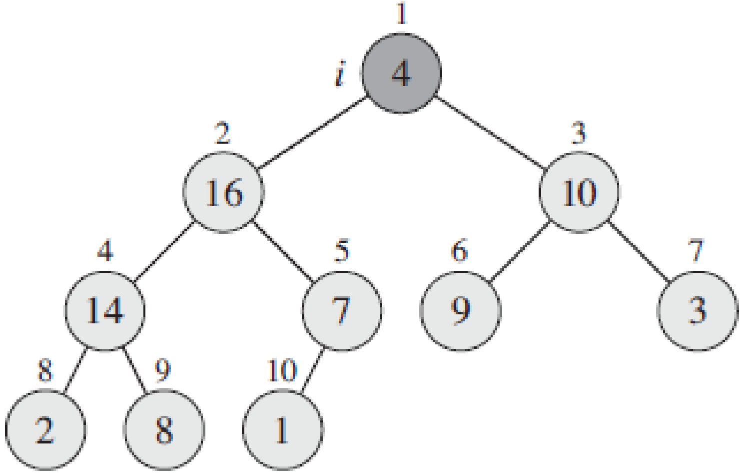
```
BUILD-MAX-HEAP(A)
A.heap-size = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)
```

Proof of Correctness

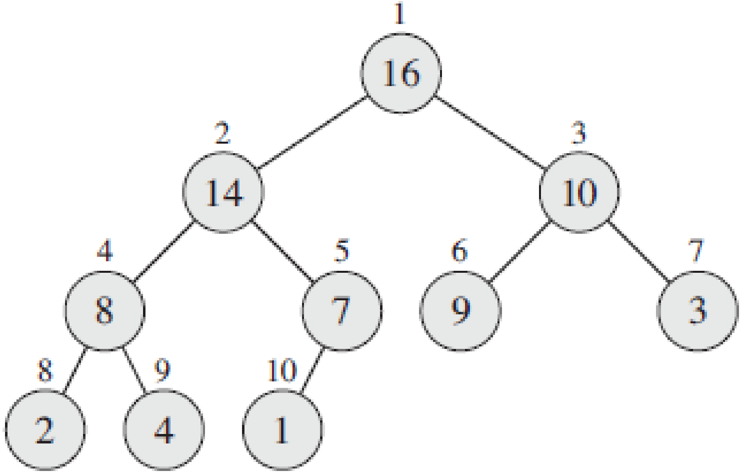
- Find the loop invariant
- Show that it is true prior to execution of the loop
- Show that it holds after execution of the loop
- Show that the loop invariant provides a useful property to show the correctness of the algorithm when loop terminates
- Loop invariant => all the elements from the initial value down to the highest value of i form a max-heap



A
4 1 3 2 16 9 10 14 8 7
i
(a)
(b)
(c)
(d)



(e)



(f)

Complexity Analysis

- Each call to heapify takes at most O(lgn) time
- The procedure heapify is called for at most the total number of nodes in the heap i.e. n
- Therefore the upper bound on the running time of the algorithm is O(nlgn)
- This is not however asymptotically tight
- A tighter bound can be obtained by noting that an n-element heap has:
  - Maximum height = $\lfloor \lg n \rfloor$
  - Maximum number of nodes with height h = $\left\lceil \frac{n}{2^{h+1}} \right\rceil$

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right).$$

We evalaute the last summation by substituting $x = 1/2$ in the formula yielding

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$$

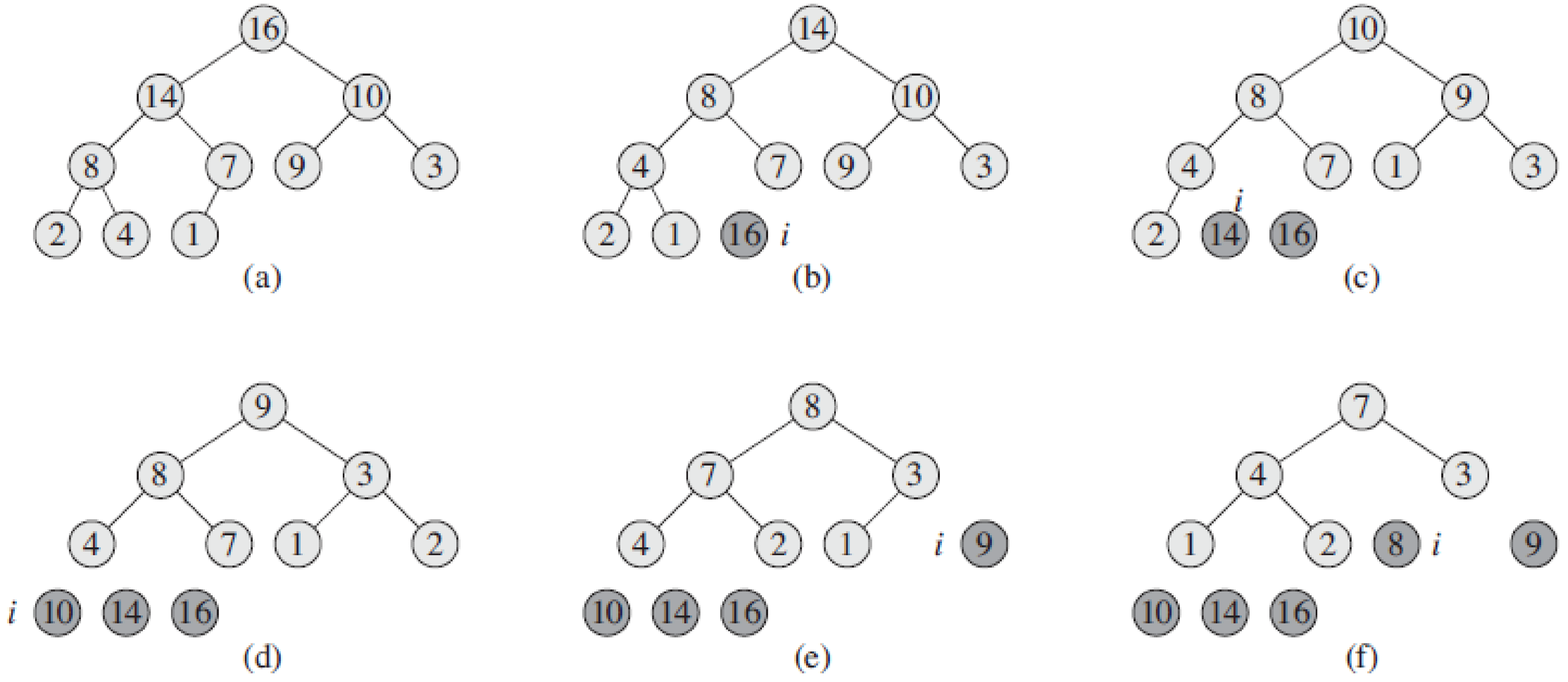
Thus, we can bound the running time of BUILD-MAX-HEAP as

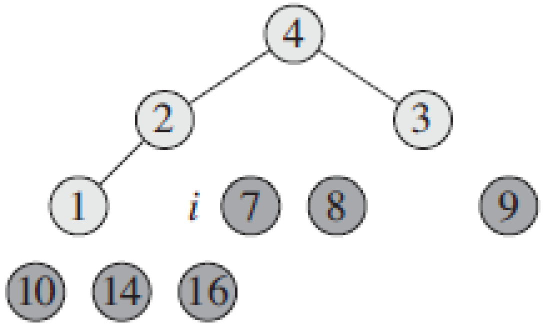
$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$$

Heapsort

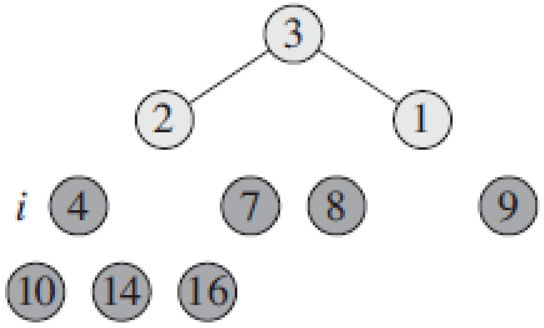
```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size − 1
    MAX-HEAPIFY(A, 1)
```

- Largest element is at the top and therefore can be placed at the end of the array
- This is done by exchanging the value of 1st element with the last element of the heap
- Since one element of the array is at its correct sorted position, we remove it from the heap by reducing the heap-size by 1
- After exchange, the root of the heap might not follow the heap property so we call Heapify with the root (1)

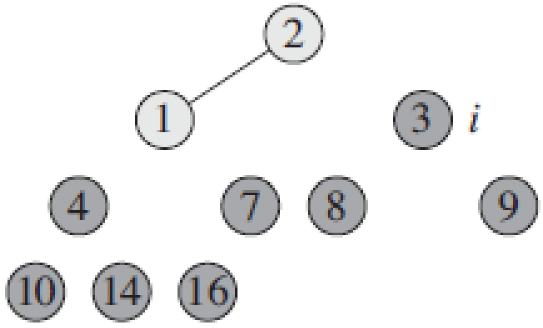

(a)
(b)
(c)
(d)
(e)
(f)



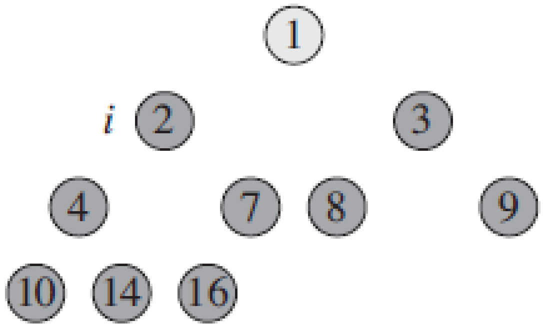
(g)



(h)



(i)



(j)

A	1	2	3	4	7	8	9	10	14	16

(k)

Complexity Analysis

- Build-heap takes O(n) time
- Each call to Heapify takes O(lgn) time
- The total running time is thus O(nlgn)

Priority Queue

- A data structure where the elements are accessed in the order of their priority
- For a set of elements S, each element is associated with a *key* value
- Priority queue is of two types: max-priority queue and min-priority queue
- Operations (Max-priority queue):
  - Insert(S,x)
  - Maximum(S)
  - Extract-Max(S)
  - Increase-Key(S,x,k)

Implementation of Priority Queue

- The priority queue designed for a particular application stores only the key values
- Each element of the priority queue is associated with an object of the application for which priority queue is designed
- Key values are nothing but the priorities of the associated objects
- All the manipulation is done using the key values only
- A handle is maintained that associates the key value with the corresponding object
- The program object stores the index value of the key and the key stores the handle of the program object (could be a pointer to the object)
- Each time a key is moved from its position, the handles are updated so as to maintain the correct index value

Heap-Maximum

```
HEAP-MAXIMUM(A)
    return A[1]
```

Heap-Extract-Max(A)

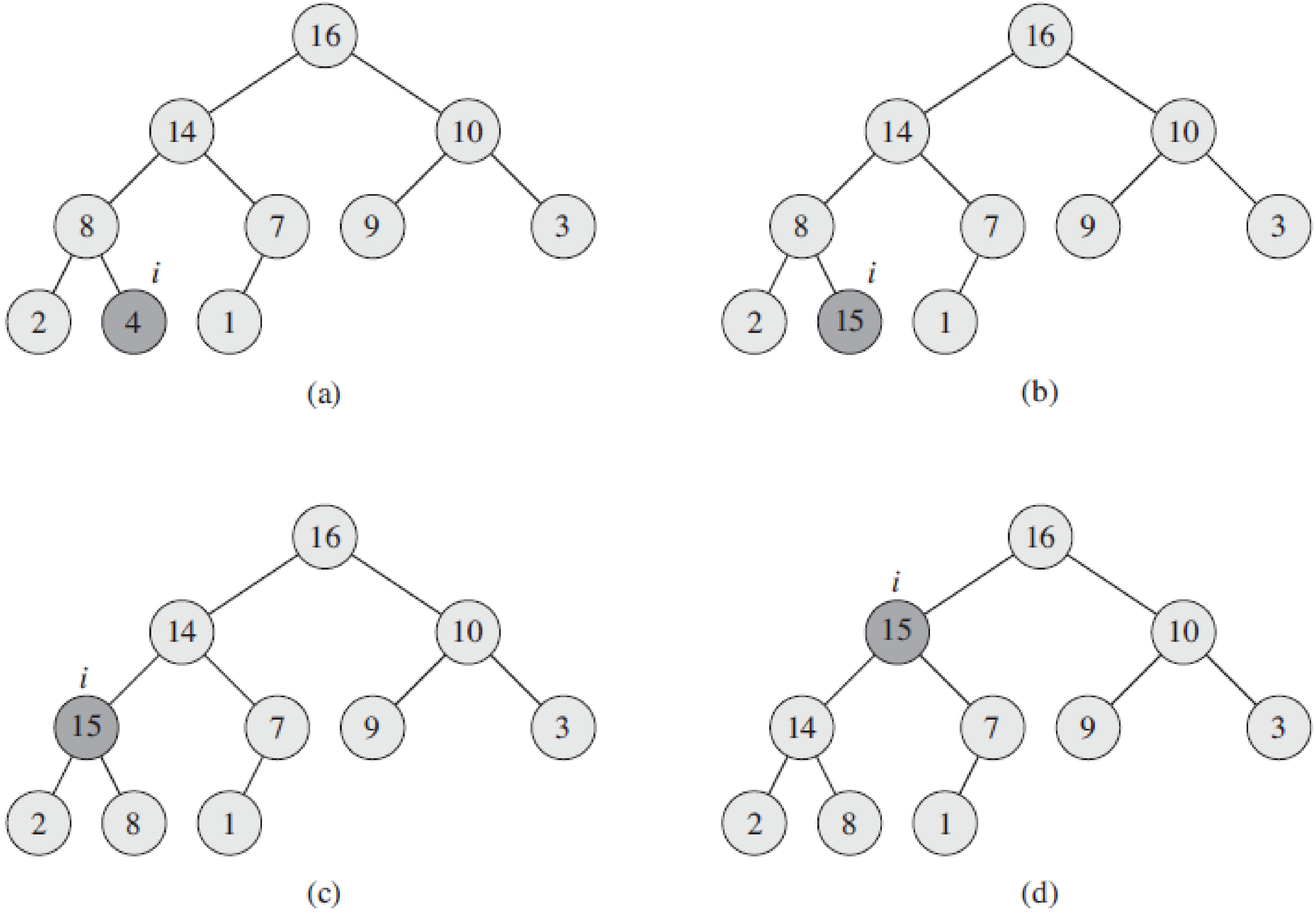
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

- Performs only constant amount of operations other than the call to Max-Heapify
- Running time is same as that of Max-Heapify i.e. O(lgn)

Heap-Increase-Key(A, i, key)

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

- The while loop runs at most up to the complete height of the heap
- Running time of the algorithm is O(lgn)


(a)
(b)
(c)
(d)

Max-Heap-Insert(A, key)

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$

- Steps 1 and 2 add constant time
- Step 3 runs in O(lgn) time
- Therefore, runtime is bounded by O(lgn)