Running Time for Max-Heapify(*A*, *n*)

- Fixing up relationships between *i*, *l*, and *r* takes Θ(1) time
- $T(n) = T(largest) + \Theta(1)$
- *If the heap at i has n elements, how many elements can the subtrees at l or r have?*
  - Draw it
- Answer: 2*n*/3 (worst case: bottom row 1/2 full)
- *largest* ≤ 2*n*/3 (worst case occurs when the last row of tree is exactly half full)
- So time taken by **`Max-Heapify()`** is given by

  $T(n) \le T(2n/3) + \Theta(1)$

L6.1

Analyzing Heapify(): Formal

- So we have

  $T(n) \le T(2n/3) + \Theta(1)$
- By case 2 of the Master Theorem,

  $T(n) = O(\lg n)$
- Thus, Max-`Heapify()` takes logarithmic time

L6.2

6.3 Building a heap

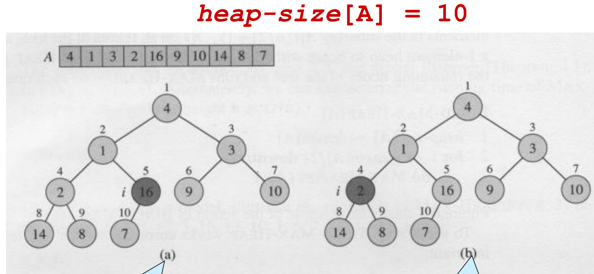
- Use Max-Heapify to convert an array *A* into a max-heap.
- How?
- Call Max-Heapify on each element in a bottom-up manner.

<u>*Build-Max-Heap(A)*</u>
1. *A.heap-size = A.length*
2. **for** *i* = ⌊*A.length* / 2⌋ **downto** 1
3. **do** *Max-Heapify(A, i)*

L6.3

`Build-Max-Heap(A)`

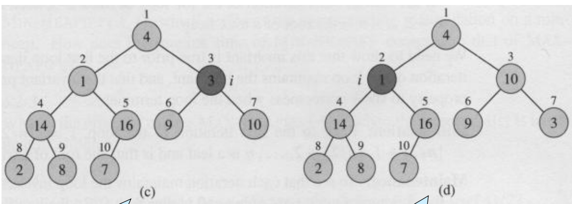
`heap-size` **`[A] = 10`**



i=5
Max-Heapify(A, i)

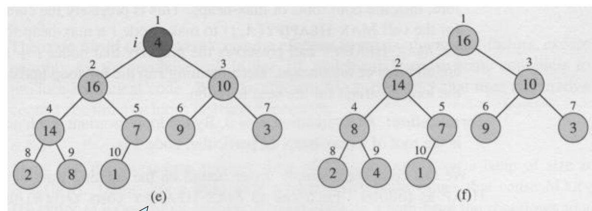
i=4
Max-Heapify(A, i)

L6.4



i=3
Max-Heapify(A, i)

i=2
Max-Heapify(A, i)



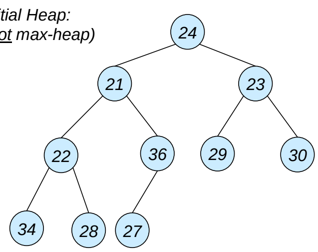
i=1
Max-Heapify(A, i)

Build-Max-Heap – Example

Input Array:

24	21	23	22	36	29	30	34	28	27

Initial Heap:
(not max-heap)



Build-Max-Heap – Example

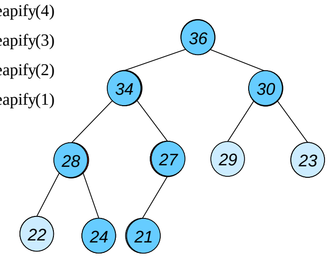
MaxHeapify($\lfloor 10/2 \rfloor$ = 5)

MaxHeapify(4)

MaxHeapify(3)

MaxHeapify(2)

MaxHeapify(1)



Running Time of *Build-Max-Heap*

- Loose upper bound:
  - Cost of a *Max-Heapify* call × No. of calls to *Max-Heapify*
  - $O(\lg n) \times O(n) = O(n\lg n)$
- Tighter bound:
  - Cost of a call to *Max-Heapify* at a node depends on the height, h, of the node – $O(h)$.
  - Height of most nodes smaller than n.
  - Height of nodes h ranges from 0 to $\lfloor \lg n \rfloor$.
  - No. of nodes of height h is $\lceil n/2^{h+1} \rceil$

L6.9

Running Time of *Build-Max-Heap*

Tighter Bound for T(Build-Max-Heap)

T(Build-Max-Heap)

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

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \le \sum_{h=0}^{\infty} \frac{h}{2^h} \quad (\Theta \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}) = \frac{1/2}{(1-1/2)^2} \quad , x = 1/2 \text{ in (A.8)} = 2$$

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

Can build a heap from an unordered array in linear time

L6.10

6.4 Heapsort algorithm

- Sort by maintaining the as yet unsorted elements as a max-heap.
- Start by building a max-heap on all elements in A.
  - Maximum element is in the root, $A[1]$.
- Move the maximum element to its correct final position.
  - Exchange $A[1]$ with $A[n]$.
- Discard $A[n]$ – it is now sorted.
  - Decrement heap-size[A].
- Restore the max-heap property on $A[1..n-1]$.
  - Call *Max-Heapify*(*A*, 1).
- Repeat until heap-size[A] is reduced to 2.

L6.11

Heapsort algorithm

Heapsort(A)

- To sort an array in place.

```
Heapsort(A)
1  Build-Max-Heap(A)
2  for i = A.length down to 2
3      exchange A[1]↔A[i]
4        A.heap-size = A.heap-size -1
5        Max-Heapify(A,1)
```

Heapsort Visualization 1

Heapsort Visualization 2

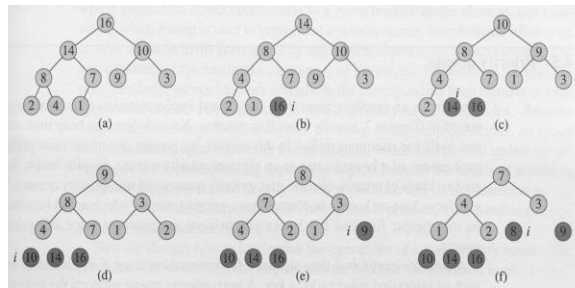
L6.12

Heapsort

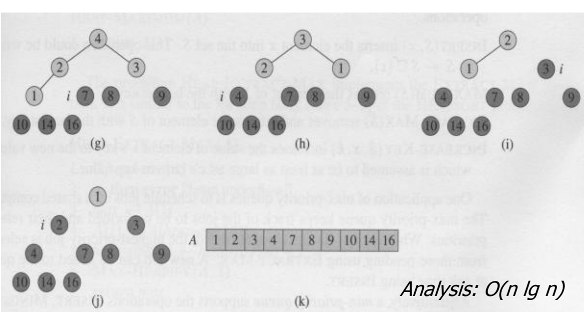
```
Heapsort(A)
{
      Build-Max-Heap(A);
      for (i = length(A) downto 2)
      {
            Swap(A[1], A[i]);
            heap_size(A) -= 1;
            Max-Heapify(A, 1);
      }
}
```

L6.13

Operation of Heapsort



L6.14



Analysis: O(n lg n)

L6.15

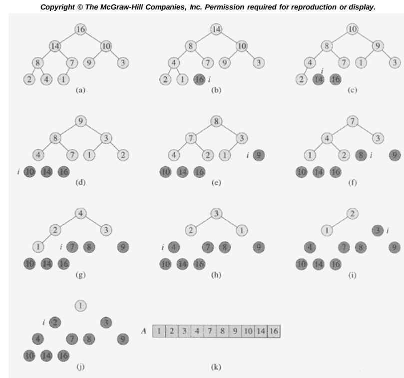


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after it has been built by BUILD-MAX-HEAP. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

L6.16

Algorithm Analysis

```
Heapsort(A)
  Build-Max-Heap(A)
  for i = A.length down to 2
       exchange A[1]↔A[i]
        A.heap-size = A.heap-size -1
        Max-Heapify(A,1)
```

- In-place
- Not Stable
- Build-Max-Heap takes $O(n)$ and each of the n-1 calls to Max-Heapify takes time $O(\lg n)$.
- Therefore, $T(n) = O(n \lg n)$

Heap Procedures for Sorting

- Max-Heapify $O(\lg n)$
- Build-Max-Heap $O(n)$
- HeapSort $O(n \lg n)$

Analyzing Heapsort

- The call to **`Build-Max-Heap()`** takes $O(n)$ time
- Each of the n - 1 calls to **`Max-heapify()`** takes $O(\lg n)$ time
- Thus the total time taken by **`HeapSort()`**
  $= O(n) + (n - 1)\, O(\lg n)$
  $= O(n) + O(n \lg n)$
  $= O(n \lg n)$

6.5 Priority Queues

- Popular & important application of heaps.
- Max and min priority queues.
- Maintains a *dynamic* set S of elements.
- Each set element has a *key* – an associated value.
- Goal is to support insertion and extraction efficiently.
- Applications:
  - Ready list of processes in operating systems by their priorities – the list is highly dynamic
  - In event-driven simulators to maintain the list of events to be simulated in order of their time of occurrence.

Priority Queues

- Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins
- But the heap data structure is incredibly useful for implementing *priority queues*
  - A data structure for maintaining a set *S* of elements, each with an associated value or *key*
  - Supports the operations `Insert()`, `Maximum()`, and `ExtractMax()`
  - ***What might a priority queue be useful for?***

Priority Queues

- A max-priority queue support the following operations:

`Insert(S, x)` O(lg *n*)

inserts the element *x* into the set *S*.

`Maximum(S)` O(1)

returns the element of *S* with the largest key.

`Extract-Max(S)` O(lg *n*)

removes and returns the element of *S* with the largest key.

`Increase-Key(S,x,k)` O(lg *n*)

increases the value of element *x*'s key to the new value *k*, where $k \geq x$'s current key value

Heap-Maximum

```
Heap-Maximum(A)
return A[1]
```


Heap_Extract-Max

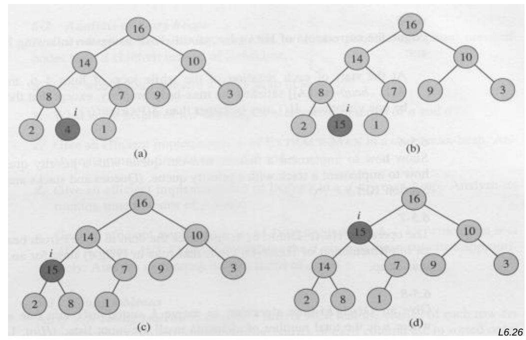
```
Heap_Extract-Max(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1]= A[A.heap-size]
A.heap-size = A.heap-size - 1
Max-Heapify(A, 1)
return max
```

Running time : Dominated by the running time of Max-Heapify
= O(lg n)

Heap-Increase-Key

```
Heap-Increase-Key (A, i, key)
1  if key < A[i]
2      error "new key is smaller than
                          current key"
3  A[i] = key
4  while i > 1 and A[Parent(i)] < A[i]
5       exchange A[i] ↔ A[Parent(i)]
6       i = Parent(i)
```


Heap-Increase-Key(*A*,*i*,15)



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

```
Heap-Increase-Key(A, i, key)
1   If key < A[i]
2        error "new key is smaller than the current key"
3   A[i] = key
4   while i > 1 and  A[Parent[i]] < A[i]
5        exchange A[i] ↔ A[Parent[i]]
6             i = Parent[i]
```

```
Heap-Insert(A, key)
1   A.heap-size = heap-size[A] + 1
2   A[A.heap-size]  = –∞
3   Heap-Increase-Key(A, A.heap-size, key)
```


Examples

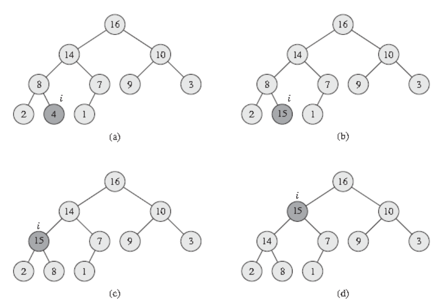


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.