Chapter 6
Heap Sort

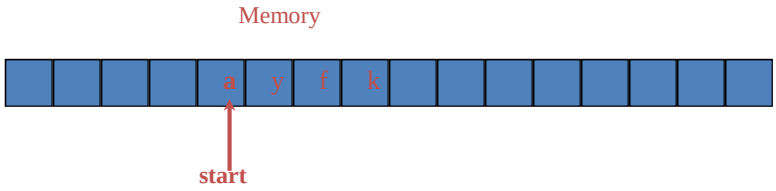
1

Outlines: Heap Sort

- Input: One-Dimension Array
- Advantages of Insertion Sort and Merge Sort
- Heap Sort:
  - **The Heap Property**
  - **Heapify Function**
  - **Build Heap Function**
  - **Heap Sort Function**
- Max-Priority Queues (Basic Operations):
  - **Maximum**
  - **Extract-Max**
  - **Increase Key**
  - **Insert Key**

2

1D Array



- 1-dimensional array x = [a, y, f, k]
- x[1] = a; x[2] = y; x[3] = f; x[4] = k

3

Sorting Revisited

- So far we've talked about two algorithms to sort an array of numbers
  - **What is the advantage of merge sort?**
    - **Answer: good worst-case running time O(n lg n)**
    - **Conceptually easy, Divide-and-Conquer**
  - **What is the advantage of insertion sort?**
    - **Answer: sorts in place: only a constant number of array elements are stored outside the input array at any time**
    - **Easy to code, When array "nearly sorted", runs fast in practice**

	avg case	worst case
Insertion sort	n^2	n^2
Merge sort	n log n	n log n

- Next on the agenda: *Heapsort*
  - **Combines advantages of both previous algorithms**

4

Heaps

- A heap can be seen as a complete binary tree
- In practice, heaps are usually implemented as arrays

- An array *A* that represent a heap is an object with two attributes: *A*[1 .. *length*[*A*]]
  - *length*[*A*]: # of elements in the array
  - *heap-size*[*A*]: # of elements in the heap stored within array *A*, where *heap-size*[*A*] ≤ *length*[*A*]
  - No element past *A*[*heap-size*[*A*]] is an element of the heap

A =

16	14	10	8	7	9	3	2	4	1

5

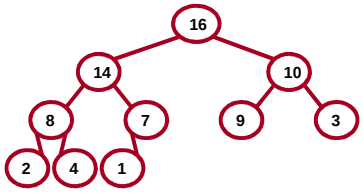
Heaps

- For example, **heap-size** of the following heap = 10
- Also, *length*[*A*] = 10

A =

16	14	10	8	7	9	3	2	4	1

=

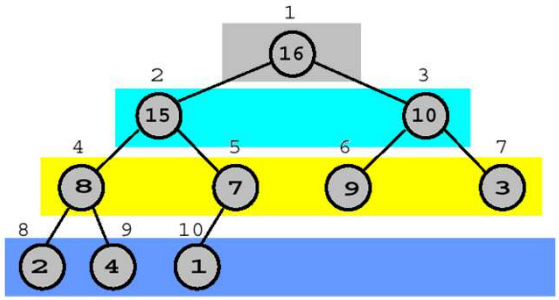


6

Referencing Heap Elements

- The root node is $A[1]$
- Node i is $A[i]$

- **Parent(*i*)**
  - **return** $\lfloor i/2 \rfloor$
- **Left(*i*)**
  - **return 2*i**
- **Right(*i*)**
  - **return 2*i + 1**



1	2	3	4	5	6	7	8	9	10
16	15	10	8	7	9	3	2	4	1

Level: 3 2 1 0

7

The Heap Property

- Heaps also satisfy the *heap property*:
  - $\mathbf{A}[Parent(i)] \geq \mathbf{A}[i]$ for all nodes $i > 1$
  - In other words, the value of a node is at most the value of its parent
  - ➔ The largest value in a heap is at its **root** (**A**[1])
  - ➔ and subtrees rooted at a specific node contain values no larger than that node's value

8

Heap Operations: Heapify()

- **`Heapify()`**: maintain the heap property
  - Given: a node *i* in the heap with children *L* and *R*
  - two subtrees rooted at *L* and *R*, **assumed to be heaps**
  - Problem: The subtree rooted at *i* may violate the heap property (*How?*)
    - **A[i] may be smaller than its children value**
  - Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property
    - **If A[i] < A[L] or A[i] < A[R], swap A[i] with the largest of A[L] and A[R]**
    - **Recurse on that subtree**

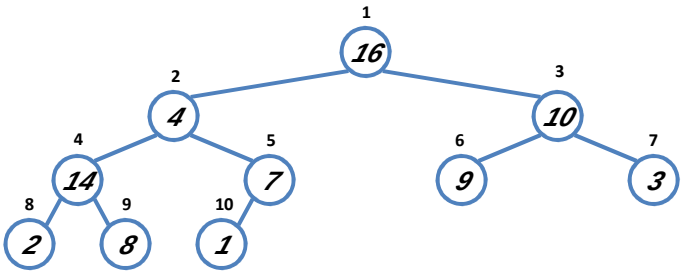
9

Heap Operations: Heapify()

```
Heapify(A, i)
{
  1. L ← left(i)
  2. R ← right(i)
  3. if L ≤ heap-size[A] and A[L] > A[i]
  4.    then largest ← L
  5.    else largest ← i
  6. if R ≤ heap-size[A] and A[R] > A[largest]
  7.    then largest ← R
  8. if largest ≠ i
  9.    then exchange A[i] ↔ A[largest]
  10.         Heapify(A, largest)
}
```

10

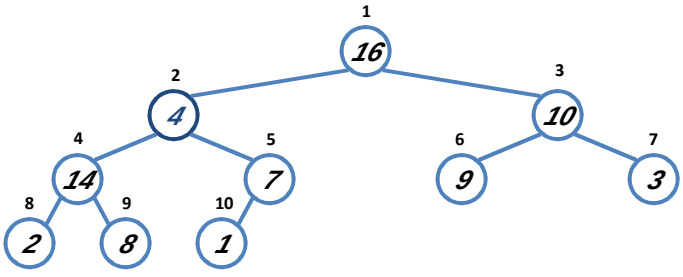
Heapify(A, 2) Example



A = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

11

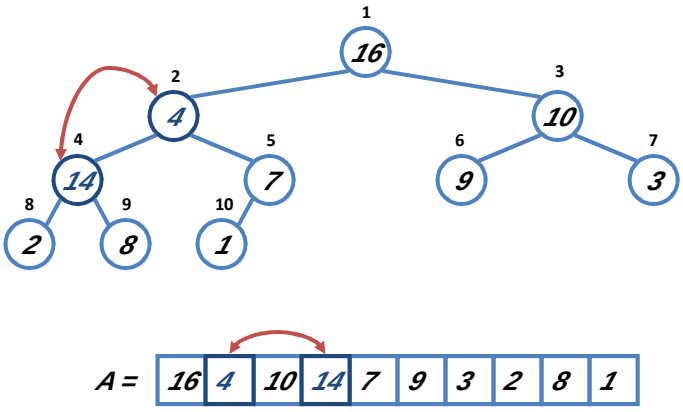
Heapify(A, 2) Example



A = | 16 | 4 | 10 | 14 | 7 | 9 | 3 | 2 | 8 | 1 |

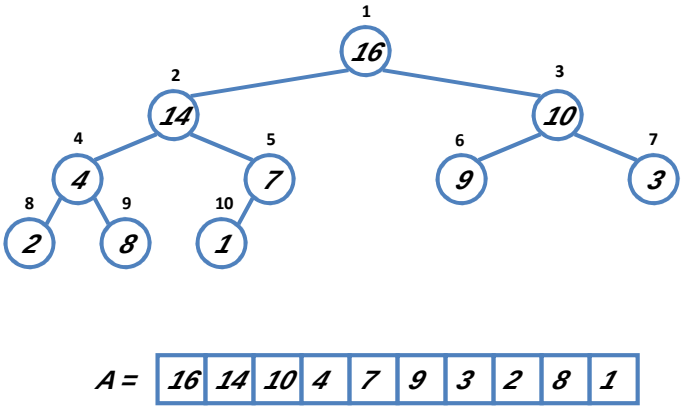
12

Heapify(A, 2) Example



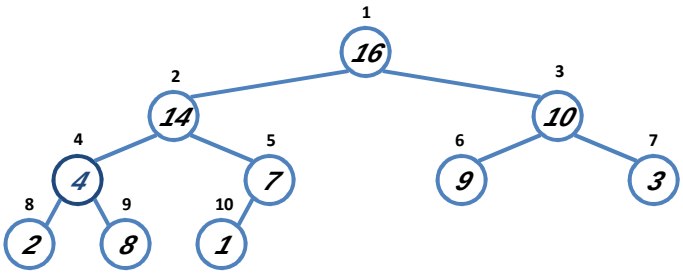
13

Heapify(A, 2) Example



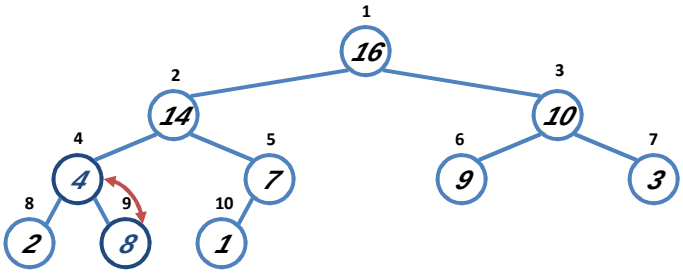
14

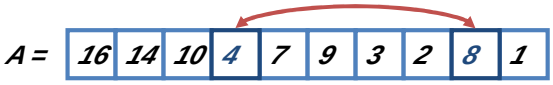
Heapify(A, 2) Example



15

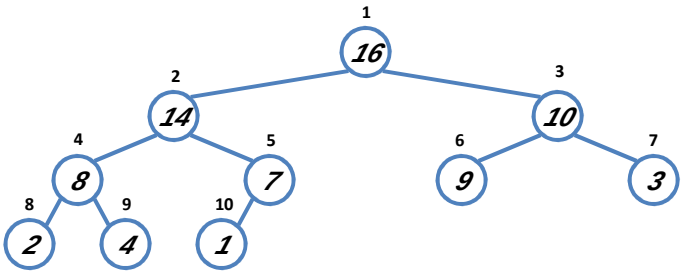
Heapify() Example





16

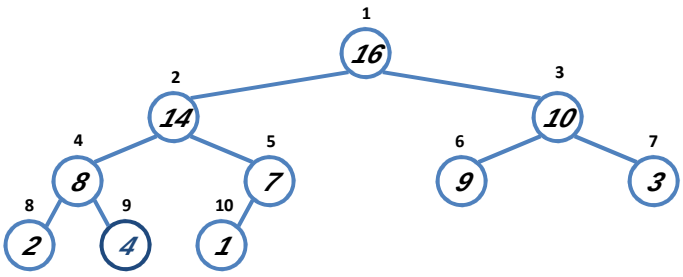
Heapify(A, 2) Example



A = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

17

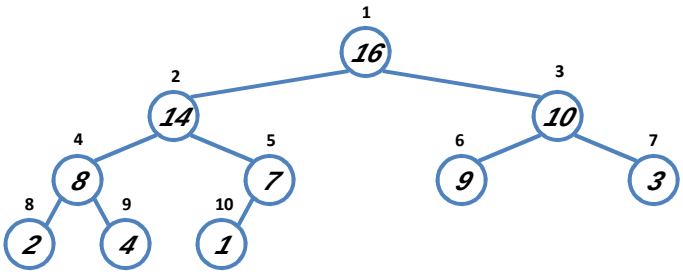
Heapify(A, 2) Example



A = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

18

Heapify(A, 2) Example



A = | 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |

19

Heap Height

- Definitions:
  - The *height* of a node in the tree = the number of edges on the longest downward path to a leaf

- *What is the height of an n-element heap? Why?*
  - The height of a tree for a heap is $\Theta(lg\ n)$
  - Because the heap is a binary tree, the height of any node is at most $\Theta(\lg n)$
    - **Thus, the basic operations on heap runs in $O(lg\ n)$**

20

of nodes in each level

- Fact: an n-element heap has at most 2^{h-k} nodes of level k, where h is the height of the tree

- for $k = h$ (root level) → $2^{h-h} = 2^0 = 1$
- for $k = h-1$ → $2^{h-(h-1)} = 2^1 = 2$
- for $k = h-2$ → $2^{h-(h-2)} = 2^2 = 4$
- for $k = h-3$ → $2^{h-(h-3)} = 2^3 = 8$
- …
- for $k = 1$ → $2^{h-1} = 2^{h-1}$
- for $k = 0$ (leaves level) → $2^{h-0} = 2^h$

21

Heap Height

- A heap storing n keys has height $h = \lfloor \lg n \rfloor = \Theta(\lg n)$
- Due to heap being **complete**, we know:
  - The maximum # of nodes in a heap of height h
    - $2^h + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^0 =$
    - $\sum_{i=0 \text{ to } h} 2^i = (2^{h+1}-1)/(2-1) = 2^{h+1} - 1$
  - The minimum # of nodes in a heap of height h
    - $\mathbf{1} + 2^{h-1} + \ldots + 2^2 + 2^1 + 2^0 =$
    - $\sum_{i=0 \text{ to } h-1} 2^i + 1 = [(2^{h-1+1}-1)/(2-1)] + 1 = 2^h$
  - Therefore
    - $2^h \le n \le 2^{h+1} - 1$
    - $h \le \lg n$ & $\lg(n+1) - 1 \le h$
    - $\lg(n+1) - 1 \le h \le \lg n$
  - which in turn implies:
    - $h = \lfloor \lg n \rfloor = \Theta(\lg n)$

22

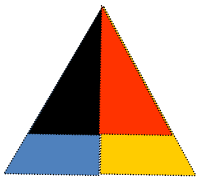
Analyzing Heapify()

- ***Aside from the recursive call, what is the running time of `Heapify()`?***
- ***How many times can `Heapify()` recursively call itself?***
- ***What is the worst-case running time of `Heapify()` on a heap of size n?***

23

Analyzing Heapify()

- The running time at any given node *i* is
  - $\Theta(1)$ time to fix up the relationships among $A[i]$, $A[\text{Left}(i)]$ and $A[\text{Right}(i)]$
  - plus the time to call *Heapify* recursively on a sub-tree rooted at one of the children of node *i*
- And, the children's subtrees each have size at most $2n/3$
  - The worst case occurs when the **last row of the tree is exactly** half full
  - **Blue** =**Yellow** = **Black** = **Red** = ¼ *n*
  - **Blue** + **Black** = ½ *n*
  - **Yellow** + **Red**= ½ *n*
  - Level 0: leave level = **Blue** +**Yellow** = ½ *n* = 2^h



24

Analyzing Heapify()

- So we have

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

- By case 2 of the Master Theorem,

  $T(n) = O(\lg n)$

- Alternately, Heapify takes $T(n) = \Theta(h)$
  - **h = height of heap = lg *n***
    - **$T(n) = \Theta(\lg n)$**

25

Heap Operations: BuildHeap()

- We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays
  - **Fact: for array of length *n*, all elements in range $A[\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2 \,..\, n]$ are heaps (*Why?*)**
    - These elements are **leaves**, they do not have children
  - **We know that**
    - $2^{h+1}-1 = n$ → $2.2^h = n + 1$ →
    - $2^h = (n + 1)/2$ $= \lfloor n/2 \rfloor + 1 = \lceil \boldsymbol{n/2} \rceil$
  - We also know that the leave-level has at most
- 2^h nodes $= \lfloor n/2 \rfloor + 1 = \lceil \boldsymbol{n/2} \rceil$ nodes
  - and other levels have a total of $\lfloor n/2 \rfloor$ nodes
  - $\lfloor n/2 \rfloor + 1 + \lfloor n/2 \rfloor = \lceil n/2 \rceil + \lfloor n/2 \rfloor = n$

26

Heap Operations: BuildHeap()

- **So:**
  - **Walk backwards through the array from *n*/2 to 1, calling `Heapify()` on each node.**
  - **Order of processing guarantees that the children of node *i* are heaps when *i* is processed**

27

BuildHeap()

```
// given an unsorted array A, make A a heap

BuildHeap(A)
{
   1. heap-size[A] ← length[A]
   2. for i ← ⌊length[A]/2⌋ downto 1
   3.      do Heapify(A, i)
}
```

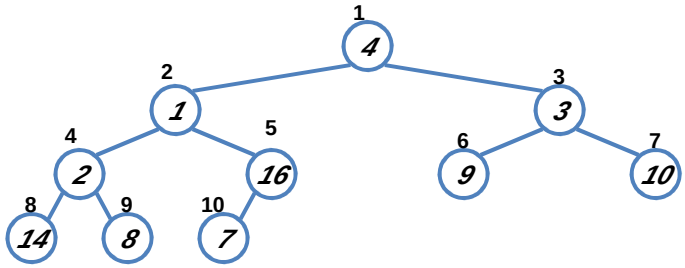
The ***Build Heap*** procedure, which runs in linear time, produces a *max-heap* from an unsorted input array.

However, the ***Heapify*** procedure, which runs in O(lg n) time, is the key to maintaining the heap property.

28

BuildHeap() Example

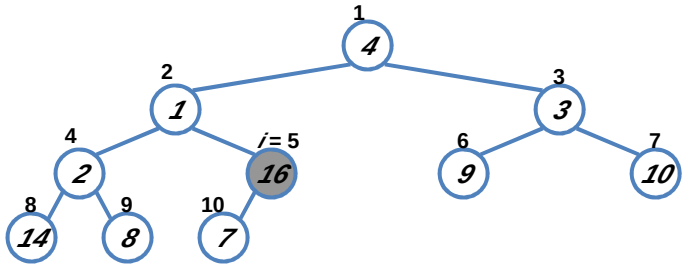
- Work through example
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}
- n=10, n/2=5



29

BuildHeap() Example

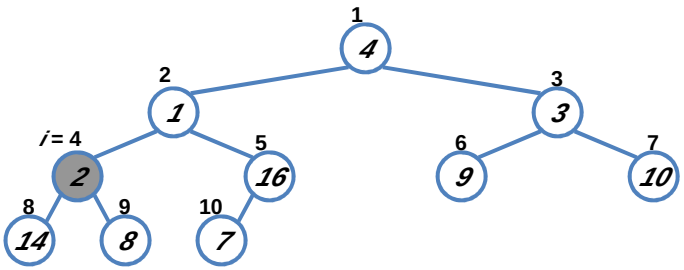
- A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



30

BuildHeap() Example

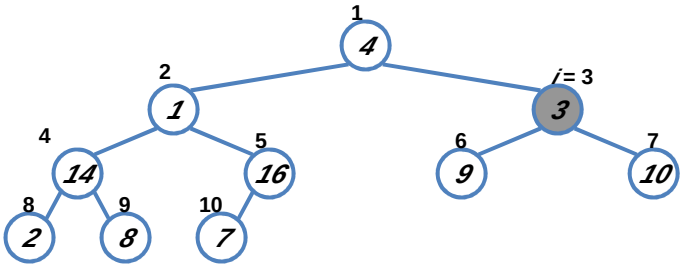
❖ A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



31

BuildHeap() Example

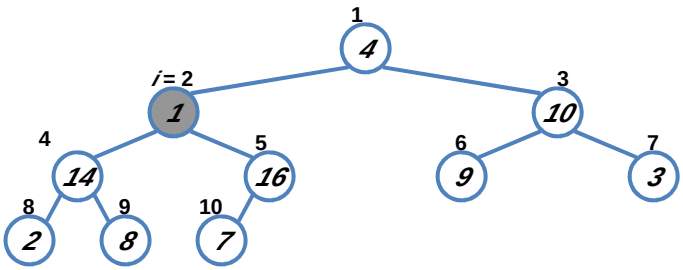
❖ A = {4, 1, 3, **14**, 16, 9, 10, **2**, 8, 7}



32

BuildHeap() Example

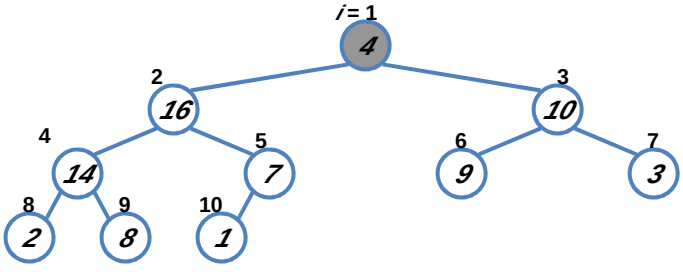
❖ A = {4, 1, **10**, 14, 16, 9, **3**, 2, 8, 7}



33

BuildHeap() Example

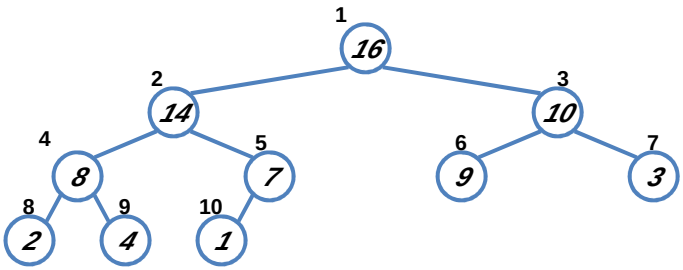
❖ A = {4, **16**, 10, 14, **7**, 9, 3, 2, 8, **1**}



34

BuildHeap() Example

- A = {**16**, **14**, 10, **8**, 7, 9, 3, 2, **4**, 1}



35

Analyzing BuildHeap()

- Each call to **`Heapify()`** takes O(lg *n*) time
- There are O(*n*) such calls (specifically, $\lfloor n/2 \rfloor$)
- Thus the running time is O(*n* lg *n*)
  - ***Is this a correct asymptotic upper bound?***
    - ***YES***
  - ***Is this an asymptotically tight bound?***
    - ***NO***
- A tighter bound is O(*n*)
  - *How can this be? Is there a flow in the above reasoning?*
  - *We can derive a tighter bound by observing that the time for Heapify to run at a node* **varies** *with the height of the node in the tree, and the heights of most nodes are small.*
- Fact: an *n*-element heap has at most 2^{h-k} nodes of level *k*, where *h* is the height of the tree.

36

Analyzing BuildHeap(): Tight

- The time required by *Heapify* on a node of height k is $O(k)$. So we can express the total cost of *BuildHeap* as

$$\sum_{k=0 \text{ to } h} 2^{h-k} O(k) = O(2^h \sum_{k=0 \text{ to } h} k/2^k)$$
$$= O(n \sum_{k=0 \text{ to } h} k(½)^k)$$

From: $\sum_{k=0 \text{ to } \infty} k x^k = x/(1-x)^2$ *where* $x = 1/2$

So, $\sum_{k=0 \text{ to } \infty} k/2^k = (1/2)/(1 - 1/2)^2 = 2$

Therefore, $O(n \sum_{k=0 \text{ to } h} k/2^k) = O(n)$

- So, we can bound the running time for building a heap from an unordered array in linear time.

37

Analyzing BuildHeap(): Tight

- How? By using the following "trick"

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \text{ if } |x| < 1 \quad \text{//differentiate}$$

$$\sum_{i=1}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2} \quad \text{//multiply by } x$$

$$\sum_{i=1}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2} \quad \text{//plug in } x = \frac{1}{2}$$

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = \frac{1/2}{1/4} = 2$$

- Therefore *BuildHeap* time is $O(n)$

38

Heapsort

- Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:
  - **Maximum element is at A[1]**
  - **Discard by swapping with element at A[n]**
    - **Decrement heap_size[A]**
    - **A[n] now contains correct value**
  - **Restore heap property at A[1] by calling `Heapify()`**
  - **Repeat, always swapping A[1] for A[heap_size(A)]**

39

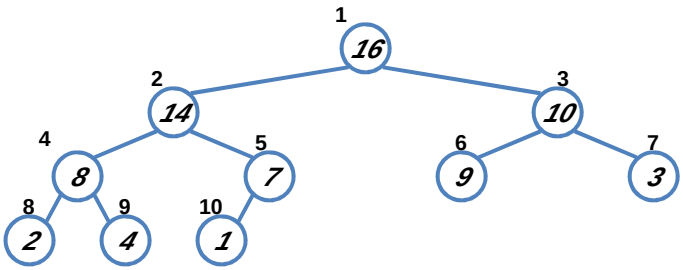
Heapsort

```
Heapsort(A)
{
    1. Build-Heap(A)
    2. for i ← length[A] downto 2
    3.     do exchange A[1] ↔ A[i]
    4.     heap-size[A] ← heap-size[A] - 1
    5.     Heapify(A, 1)
}
```

40

HeapSort() Example

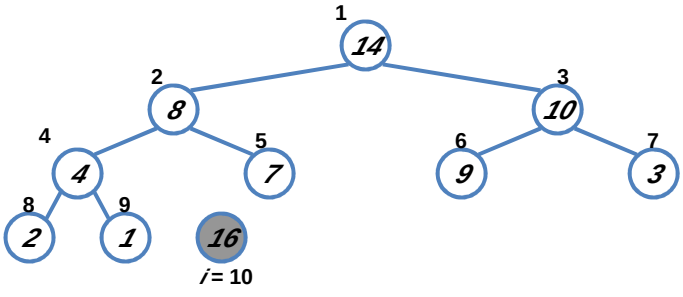
- A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}



41

HeapSort() Example

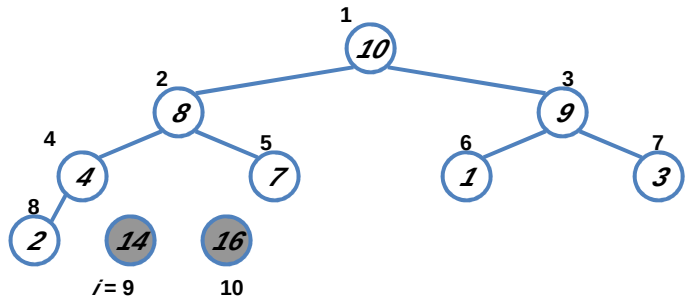
- A = {14, 8, 10, 4, 7, 9, 3, 2, 1, **16**}



42

HeapSort() Example

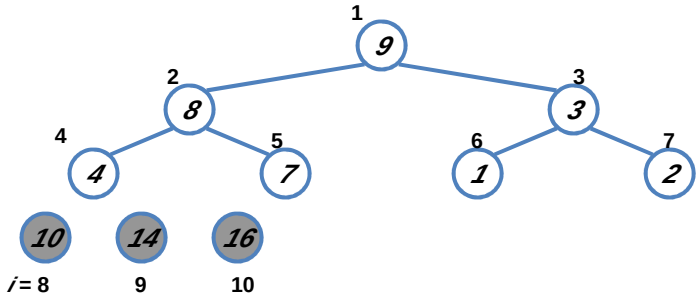
❖ A = {10, 8, 9, 4, 7, 1, 3, 2, **14**, **16**}



43

HeapSort() Example

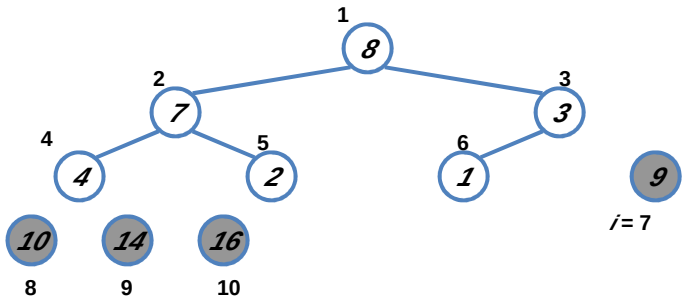
❖ A = {9, 8, 3, 4, 7, 1, 2, **10**, **14**, **16**}



44

HeapSort() Example

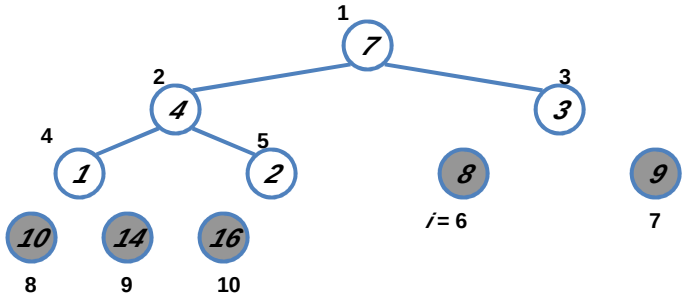
❖ A = {8, 7, 3, 4, 2, 1, **9**, **10**, **14**, **16**}



45

HeapSort() Example

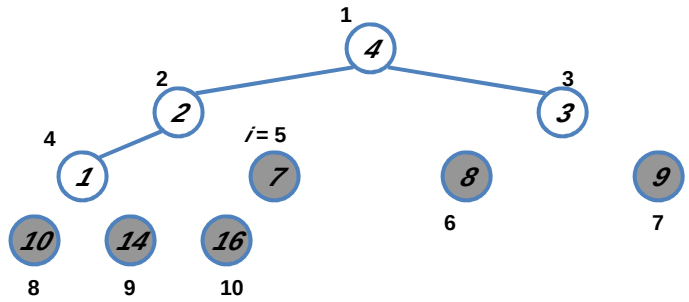
❖ A = {7, 4, 3, 1, 2, **8**, **9**, **10**, **14**, **16**}



46

HeapSort() Example

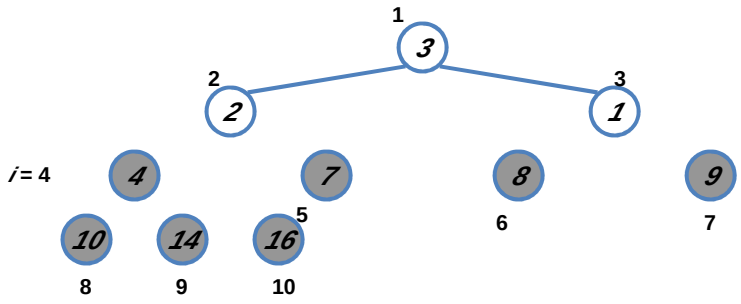
❖ A = {4, 2, 3, 1, **7**, **8**, **9**, **10**, **14**, **16**}



47

HeapSort() Example

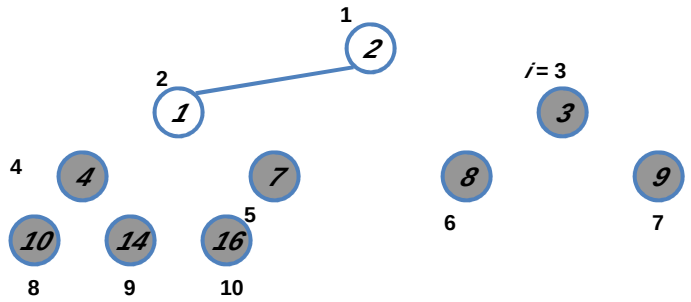
❖ A = {3, 2, 1, **4**, **7**, **8**, **9**, **10**, **14**, **16**}



48

HeapSort() Example

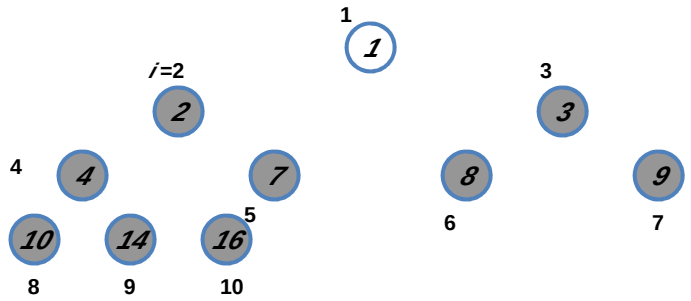
❖ A = {2, 1, **3**, **4**, **7**, **8**, **9**, **10**, **14**, **16**}



49

HeapSort() Example

❖ A = {1, **2**, **3**, **4**, **7**, **8**, **9**, **10**, **14**, **16**}



50

Analyzing Heapsort

- The call to **`BuildHeap()`** takes O(*n*) time
- Each of the *n* - 1 calls to **`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*)

51

Analyzing Heapsort

- The O(n log n) run time of heap-sort is much better than the O(n^2) run time of selection and insertion sort

- Although, it has the same run time as Merge sort, but it is better than Merge Sort regarding memory space
  - **Heap sort is in-place sorting algorithm**
  - But **not stable**
    - **Does not preserve the relative order of elements with equal keys**

52

Max-Priority Queues

- A data structure for maintaining a set *S* of elements, each with an associated value called a *key*.
- Applications:
  - Scheduling jobs on a shared computer.
  - Prioritizing events to be processed based on their predicted time of occurrence.
  - Printer queue.
- Heap can be used to implement a *max-priority queue*.

53

Max-Priority Queue: Basic Operations

- Maximum(*A*): ⟶ *return A*[1]
  - returns the element of *A* with the largest key (value)
- Extract-Max(*A*):
  - removes and returns the element of *A* with the largest key
- Increase-Key(*A*, *x*, *k*):
  - increases the value of element *x*'s key to the new value *k*, $x.value \leq k$
- Insert(*A*, *x*):
  - inserts the element *x* into the set *A*, i.e. $A \rightarrow A \cup \{x\}$

54

Extract-Max(*A*)

1. if *heap-size*[*A*] < 1 // zero elements
2. then error "heap underflow"
3. *max* ← *A*[1] // max element in first position
4. *A*[1] ← *A*[*heap-size*[*A*]]
   // value of last position assigned to first position
5. *heap-size*[*A*] ← *heap-size*[*A*] – 1
6. Heapify(*A*, 1)
7. return *max*

Note *lines 4-6 are similar to the for loop body of Heapsort procedure*

55

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

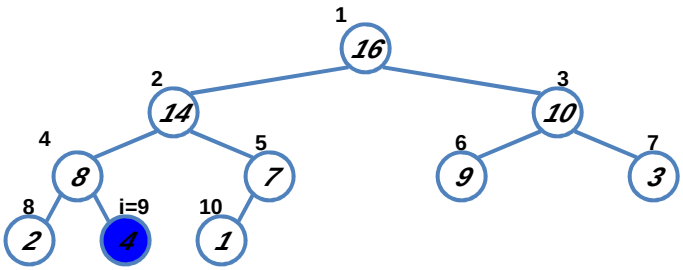
// increase a value (key) in the array

1. **if** *key* < *A*[*i*]
2. **then error** "new key is smaller than current key"
3. *A*[*i*] ← *key*
4. **while** *i* > 1 and *A*[Parent(*i*)] < *A*[*i*]
5. do exchange *A*[*i*] ←→ *A*[Parent(*i*)]
6. *i* ← Parent(*i*) // move index up to parent

56

Increase-Key(A, 4, 15) Example

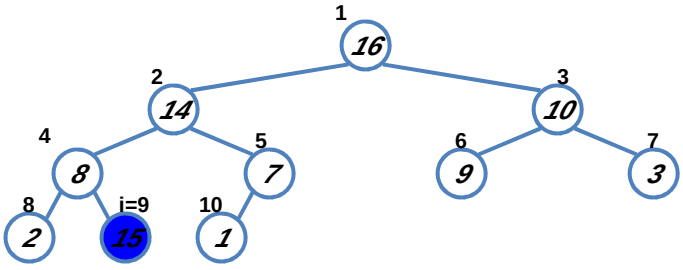
- A = {16, 14, 10, 8, 7, 9, 3, 2, 4, 1}



57

Increase-Key(A, 4, 15) Example

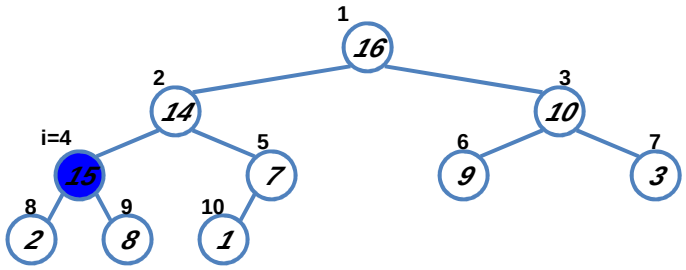
- A = {16, 14, 10, 8, 7, 9, 3, 2, 15, 1}
- The index i increased to 15.



58

Increase-Key(A, 4, 15) Example

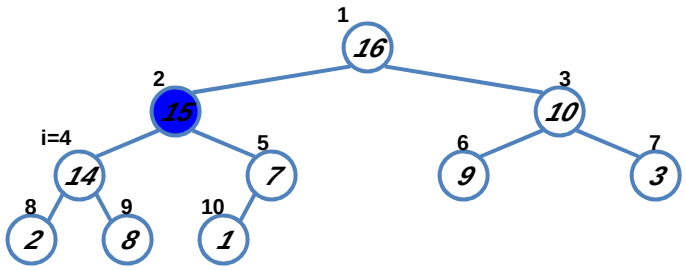
- A = {16, 14, 10, 15, 7, 9, 3, 2, 8, 1}
- After one iteration of the while loop of lines 4-6, the node and its parent have exchanged keys (values), and the index i moves up to the parent.



59

Increase-Key(A, 4, 15) Example

- A = {16, 15, 10, 14, 7, 9, 3, 2, 8, 1}
- After one more iteration of the while loop.
- The max-heap property now holds and the procedure terminates.



60

Insert(*A*, *key*)

// insert a value at the end of the binary tree then move it in the right position

1. *heap-size*[*A*] ← *heap-size*[*A*] + 1
2. *A*[*heap-size*[*A*]] ← -∞
3. Increase-Key(*A*, *heap-size*[*A*], *key*)

61

Example: Operation of Heap Insert

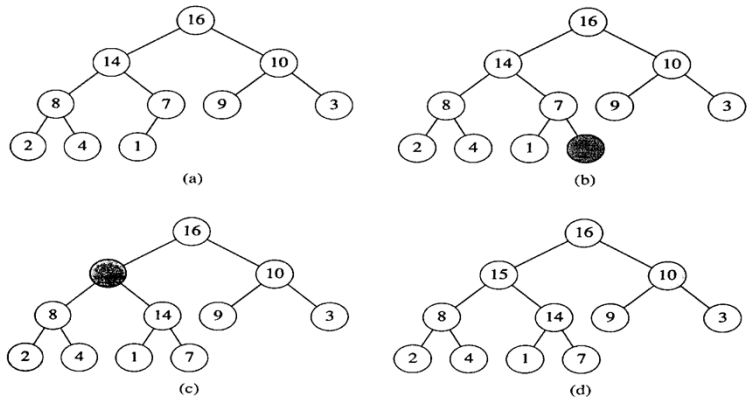


Figure 7.5 The operation of HEAP-INSERT. **(a)** The heap of Figure 7.4(a) before we insert a node with key 15. **(b)** A new leaf is added to the tree. **(c)** Values on the path from the new leaf to the root are copied down until a place for the key 15 is found. **(d)** The key 15 is inserted.

62

Running Time

- Running time of Maximum is $\Theta(1)$
- Running time of Extract-Max is $O(\lg n)$.
  - Performs only a constant amount of work + time of Heapify, which takes $O(\lg n)$ time
- Running time of Increase-Key is $O(\lg n)$.
  - The path traced from the new leaf to the root has length $O(\lg n)$.
- Running time of Insert is $O(\lg n)$.
  - Performs only a constant amount of work + time of Increase-Key, which takes $O(\lg n)$ time
- In Summary, a heap can support any max-priority queue operation on a set of size n in $O(\lg n)$ time.

63