Heapsort

Textbook: Chapter 7, Section 7.1, 7.2 and 7.3

CSCE310: Data Structures and Algorithms

`www.cse.unl.edu/~choueiry/S01-310/`

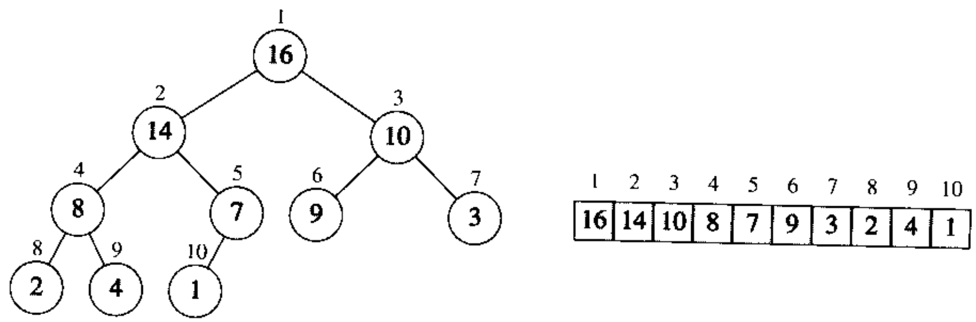
Berthe Y. Choueiry (Shu-we-ri)

Ferguson Hall, Room 104

`choueiry@cse.unl.edu`, Tel: (402)472-5444

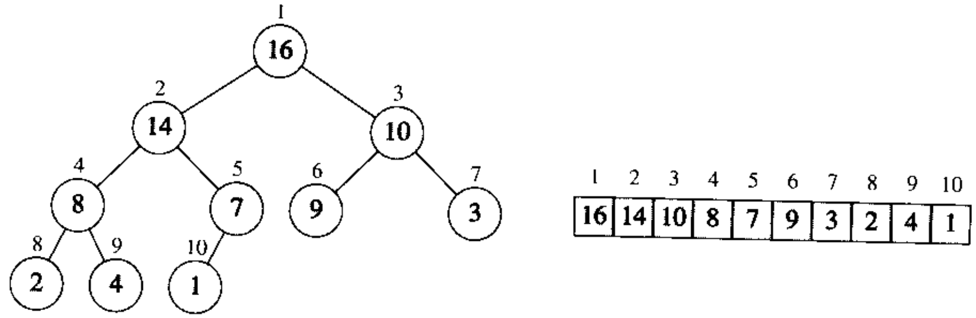
- Insertion sort $\surd$
  Sorts in $\Omega(n), O(n^2)$
  Sorts in place, in array
- Merge-sort $\surd$
  Sorts $\Theta(n \lg n)$
  Uses space, external to array
- Heapsort, $\longleftarrow$
  Sorts $O(n \lg n)$
  Sorts in place, in array
  Constant number of array elements stored outside input array at any time
  Uses **heap** $\longleftarrow$ new data structure

(Binary) Heap: Array that can be viewed as a <u>complete</u> <u>binary</u> <u>tree</u>



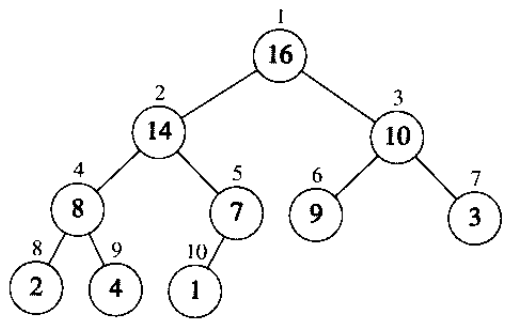
(a) (b)

- Node in the tree $\leadsto$ element in array storing same value
- Tree complete: filled on all levels, except possibly lowest level (from left up to a point)
- A is the array, $heap-size[A]$, $A[1\ldots length[A]]$, $heap-size[A] \leq length[A]$, elements in A beyond $A[heap-size[A]]$ are not elements of the heap



(a) (b)

- Root of tree is $A[1]$
- Node index i, indices of parent/children: $Parent(i)$, $Left(i)$, and $Right(i)$
- Index of $Parent(i)$ is $\lfloor i/2 \rfloor$
- Index of left child, $Left(i)$ is $2i$
- Index of right child, $Right(i)$ is $2i + 1$
- Binary representation: shift left one bit (+ one), shift right one bit



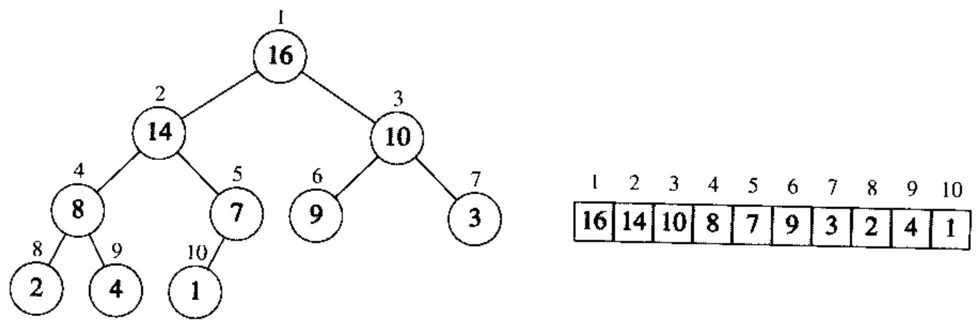
(a)

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

(b)

- **Heap property**: $\forall\, i, A[Parent(i)] \geq A[i]$, except root
  Value of a node at most value of its parent
- Largest element in a heap is stored at the root
- Subtrees rooted at a node contain smaller values than the node's

Exercise: 7.1-6. Is $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a heap?



(a) (b)

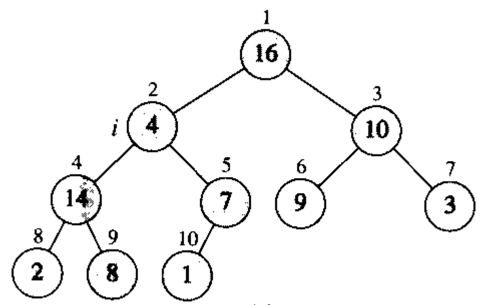
- **Height** of a node: #edges on longest simple path from node to a leaf
- Height of the tree = height of its root
- Heap of n elements, height is $\Theta(\lg n)$
- Basic operations (remember?) on heaps run in time proportional to height, thus $O(\lg n)$

Exercises: 7.1-1 and 7.1-2

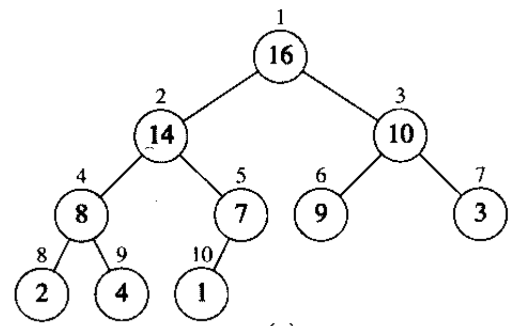
Five basic procedures

1. `Heapify` maintains heap property
   runs in $O(\lg n)$
2. `Build-Heap` produces a heap from an unordered input array
   linear in time
3. `Heapsort` sorts an array in place
   runs in $O(n \lg n)$
4. `Extract-Max` and 5. `Insert` allow heap to be used as priority queue
   run in $O(\lg n)$

`Heapify` maintains heap property



(a)



(c)

Input: An array A, and an index i in array

Assumption: Binary tree rooted at $Left(i)$ and $Right(i)$ are heaps

but $A[i]$ smaller than its children (i.e., heap property violated)

Output: heap A, heap property restored

$A[i]$ pushed down so that subtree rooted at i becomes a heap

```
HEAPIFY(A, i)
  l ← LEFT(i)
  r ← RIGHT(i)
  if l ≤ heap-size[A] and A[l] > A[i]
     then largest ← l
     else largest ← i
  if r ≤ heap-size[A] and A[r] > A[largest]
     then largest ← r
  if largest ≠ i
     then exchange A[i] ↔ A[largest]
          HEAPIFY(A, largest)
```

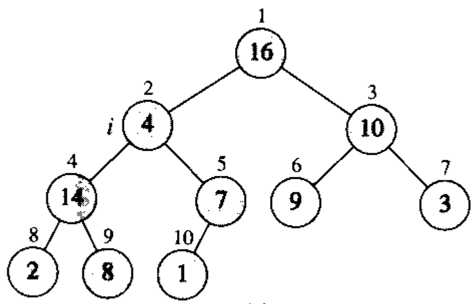
Each call stores in *largest* index of largest of

$A[i], A[Left(i), A[Right(i)]$

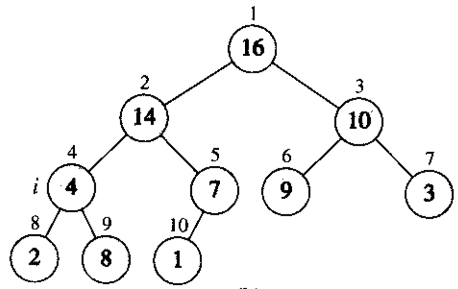
If $A[i]$ is largest, we have a heap!

Otherwise, $A[i]$ swapped with $A[largest] \rightarrow i$ satisfies heap property

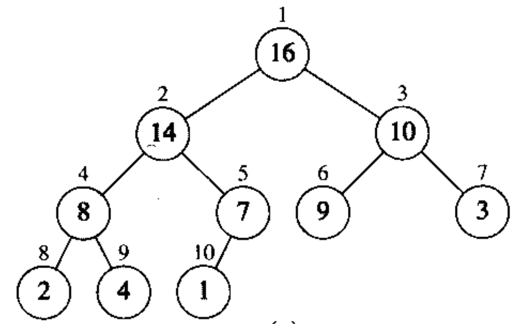
However, *largest* may violate heap property → recursive call



(a)



(b)



(c)

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else  largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        HEAPIFY(A, largest)
```

Exercise: 7.2-1. `Heapify`$(A, 3)$ on

$A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$

Running time of `Heapify`

```
HEAPIFY(A, i)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        HEAPIFY(A, largest)
```

Subtree of size n, rooted at node i:

- $\Theta(1)$ to fix up relationship between $A[i], A[Left(i), A[Right(i)]$
- time to Heapify a subtree of at most 2n/3 nodes
- Running time $T(n) \leq T(2n/3) + \Theta(1)$
- Solution: case 2 of Master theorem $T(n) = O(\lg n)$
- Alternatively, in terms of h, $T(n) = O(h)$

Building a heap: use `Heapify`

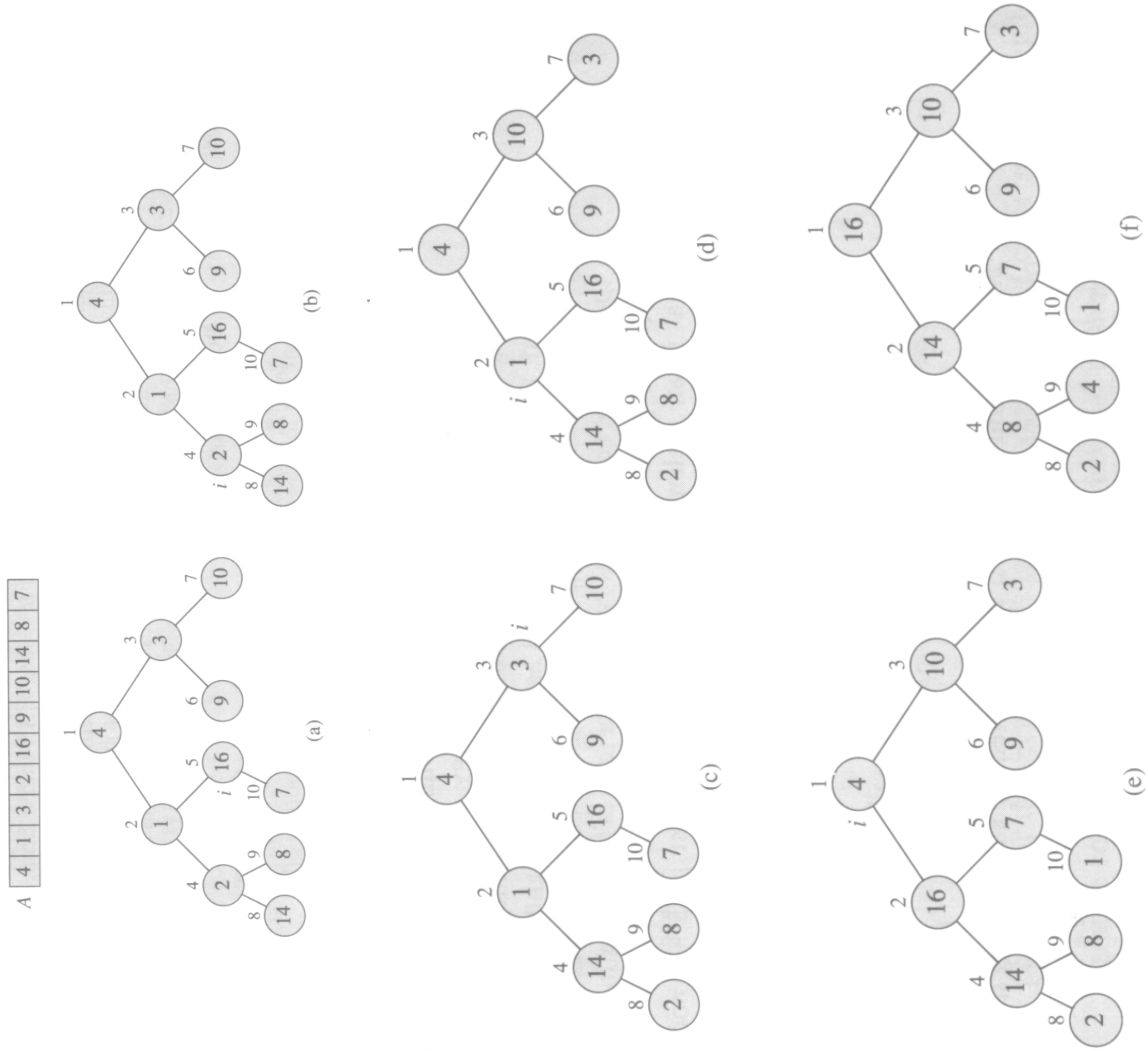
Converts array $A[1 \ldots n]$ ($n = length[A]$) into a heap.

Elements in subarray $A[(\lfloor n/2 \rfloor) + 1 \ldots n]$ are all leaves, 1-element heap, no need to heapify them (they'll stay where there are)

Heapify the remaining elements, from $\lfloor n/2 \rfloor$ downto 1

```
BUILD-HEAP(A)
  heap-size[A] ← length[A]
  for i ← ⌊length[A]/2⌋ downto 1
       do HEAPIFY(A, i)
```

Build a heap with $A = \langle 4, 1, 3, 2, 16, 9, 10, 14, 8, 7 \rangle$


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

Running time of `Build-Heap`

Simple upper bound on running time:

- Each call to `Heapify` costs $O(\lg n)$
- There are at most $O(n)$ calls
- Running time at mot $O(n \lg n)$
- Upper bound correct, but not tight:
  tighter upper bound can be computed.
  Observation: Cost of call to `Heapify` depends on height of node and heights of most nodes are small!
  One can prove running time is in $O(n)$