

Heapsort

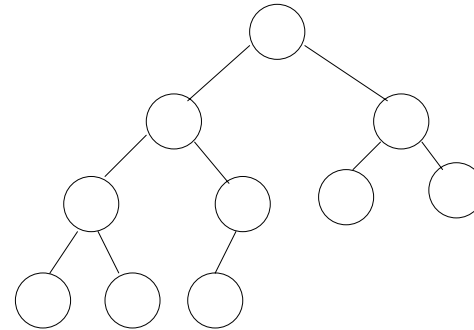
Reviews: Tree

- Tree
 - Rooted tree
 - parent, child, sibling, ancestor
- Binary tree
 - Left child, right child
- Some concepts
 - Height: # of edges on a longest simple path from node to a leaf

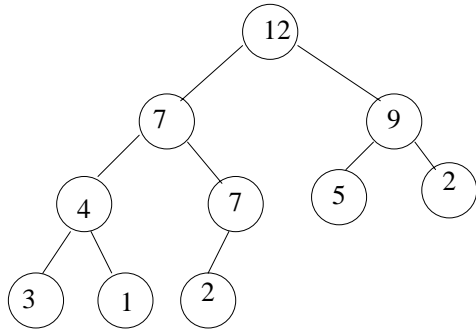
Heap Definition

- A heap is
 - An *essentially complete binary* tree which satisfies heap property.
- Binary tree
- Essentially complete binary tree
- Heap property
 - **max-heap**
 - The value (key) of each node in the heap is greater than or equal to the values (keys) of its children, if any.
 - **min-heap**
 - The value (key) of each node in the heap is less than or equal to the values (keys) of its children, if any.

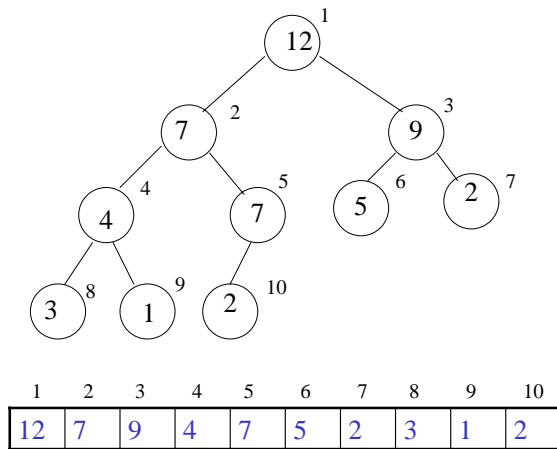
An essentially complete binary tree



A max-heap



A heap can be represented as an array



Some important properties of heaps

- Given a node $A[i]$
 - It's parent is $A[\lfloor i/2 \rfloor]$, if $i > 1$.
 - It's left child is $A[2*i]$, if $2*i \leq n$.
 - It's right child is $A[2*i+1]$, if $2*i+1 \leq n$.
- The height of a heap containing n nodes is $\lfloor \lg n \rfloor$
- There are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes with height h

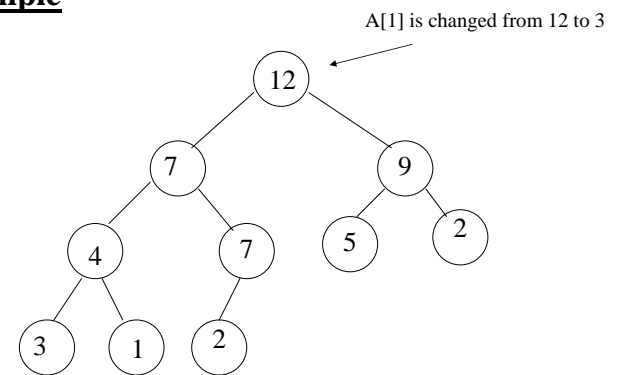
Heapify

- Assume that the left and right subtrees of $A[i]$ are already max-heaps
- $A[i]$ may be less than its children a violation
- Goal: Make the subtree rooted at index i a max-heap
- Application
 - Call `heapify(i)` when the value of $A[i]$ is decreased

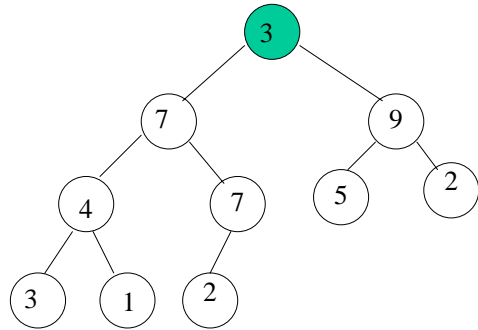
Heapify

```
MAX-HEAPIFY( $A, i, n$ )  
 $l \leftarrow \text{LEFT}(i)$   
 $r \leftarrow \text{RIGHT}(i)$   
if  $l \leq n$  and  $A[l] > A[i]$   
  then  $largest \leftarrow l$   
  else  $largest \leftarrow i$   
if  $r \leq n$  and  $A[r] > A[largest]$   
  then  $largest \leftarrow r$   
if  $largest \neq i$   
  then exchange  $A[i] \leftrightarrow A[largest]$   
  MAX-HEAPIFY( $A, largest, n$ )
```

Example

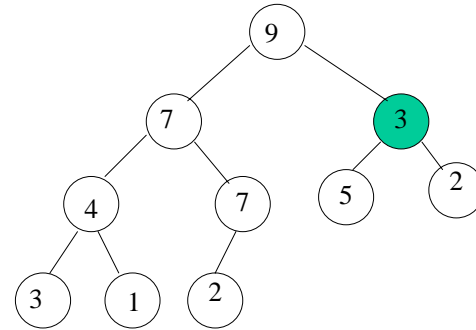


Example

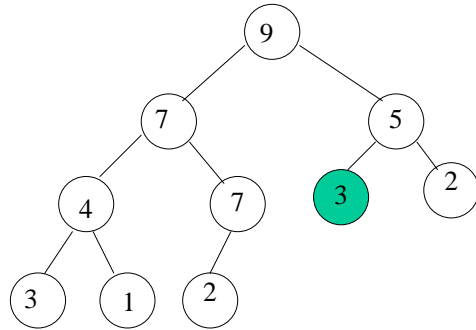


Call heapify(1)

Example



Example



Cost: $O(\log(n))$

buildHeap

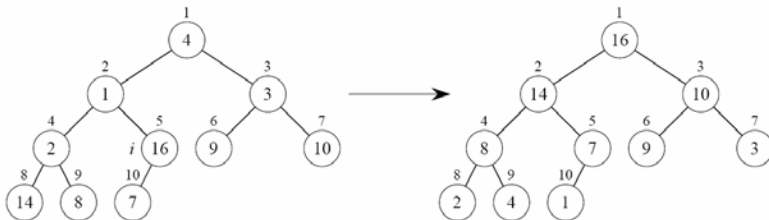
```
BUILD-MAX-HEAP( $A, n$ )  
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1  
  do MAX-HEAPIFY( $A, i, n$ )
```

- What's the idea here?
- Proof

Example

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

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



Correctness

Correctness

Loop invariant: At start of every iteration of **for** loop, each node $i + 1, i + 2, \dots, n$ is root of a max-heap.

Initialization: By Exercise 6.1-7, we know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i , so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i + 1, i + 2, \dots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i reestablishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis

- $O(n)$ calls to MAX-HEAPIFY
- Each takes $O(\log n)$

Total time: $O(n \log n)$, not tight!

Analysis

- The cost of heapify(i) for a node at height h is $O(h)$
- Number of nodes at height h is $\left\lceil \frac{n}{2^{h+1}} \right\rceil$
- The total cost is bounded by

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

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

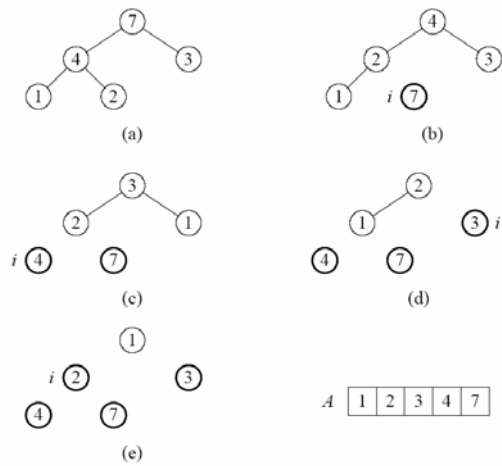
Heapsort Algorithm

- build a max-heap from the array
- starting from root, place maximum element into the correct place in array by swapping it with the element in the last position in the array
- “discard” the last node (knowing that it is in its correct place) by decreasing the heap size
- call MAX-HEAPIFY on new root
- repeat the “discarding” process until only one node remains

Heapsort Algorithm

```
HEAPSORT( $A, n$ )  
  BUILD-MAX-HEAP( $A, n$ )  
  for  $i \leftarrow n$  downto 2  
    do exchange  $A[1] \leftrightarrow A[i]$   
        MAX-HEAPIFY( $A, 1, i - 1$ )
```

Example



Analysis

- BUILD-MAX-HEAP: $O(n)$
- For loop: $n-1$ times
- Exchange elements: $O(1)$
- MAX-HEAPFY'' $O(\log n)$

Total time: $O(n \log n)$

Application: Priority Queues

Priority queue: a data structure for maintaining a set S of elements, each with an associated value called a key.

Applications: job scheduling, event-driven simulations

Operations of Priority Queues

- **Insert(S, x)**
 - Insert element x into S
- **Maximum(S)**
 - Return the element with the largest key
- **Extract-Max(L)**
 - Remove and return the largest element
- **Increase-Key(S, x, k)**
 - Increase the value of element x 's key to the new value k

Finding the maximum element

- It's the root

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

Time: $\Theta(1)$

Extracting maximum element

- Make sure heap is not empty
- Make a copy of the maximum element
- Make the last node in the tree the new root
- re-heapify the heap, with one fewer node
- Return the copy of the maximum element

```
HEAP-EXTRACT-MAX(A, n)
    if  $n < 1$ 
        then error "heap underflow"
     $max \leftarrow A[1]$ 
     $A[1] \leftarrow A[n]$ 
    MAX-HEAPIFY(A, 1, n - 1)  $\triangleright$  remakes heap
    return  $max$ 
```

Time: constant time assignments plus time for MAX-HEAPIFY: $O(\log n)$

Increasing key value

- make sure $k \geq x$'s current key
- Update x 's key value to k
- Traverse tree upward comparing x to its parent and swapping keys if necessary, until x 's key is smaller than its parent's key

```
HEAP-INCREASE-KEY( $A, i, key$ )
if  $key < A[i]$ 
  then error "new key is smaller than current key"
 $A[i] \leftarrow key$ 
while  $i > 1$  and  $A[PARENT(i)] < A[i]$ 
  do exchange  $A[i] \leftrightarrow A[PARENT(i)]$ 
   $i \leftarrow PARENT(i)$ 
```

Time: $O(\log n)$ as upward path has length $O(\log n)$

Inserting into heap

- Insert a new node in the last position of tree with key $-\infty$
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure

```
MAX-HEAP-INSERT( $A, key, n$ )
 $A[n + 1] \leftarrow -\infty$ 
HEAP-INCREASE-KEY( $A, n + 1, key$ )
```

Time: constant time assignments plus time for HEAP-INCREASE-KEY
 $O(\log n)$