Notes on Heapsort

Geoffrey Matthews

April 25, 2018

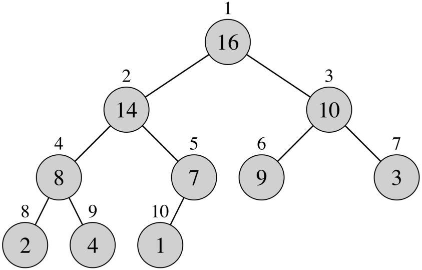
Heapsort

- $O(n \lg n)$ worst case — like merge sort
- Sorts in place — like insertion sort
- Combines best of both algorithms

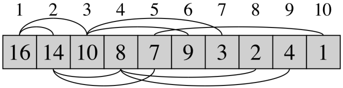
Heaps

- A nearly complete binary tree.
- **Height:** number of edges on longest path from node to leaf
- Stored as an array A
  - Root at $A[1]$
  - Left child at $A[2i]$
  - Right child at $A[2i+1]$
  - Parent of $A[i]$ at $A[\lfloor i/2 \rfloor]$
- Computing very fast

Example Max-heap



(a)



(b)

- For max-heaps: $A[\text{PARENT}(i)] \geq A[i]$
- Induction can prove largest element is at root.

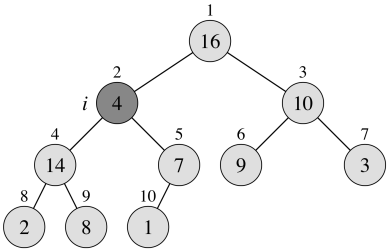
Max-Heapify

- Preconditions:
  - $A[i]$ may be smaller than its children.
  - Left and right subtrees of i are max-heaps.
- Postcondition: subtree rooted at i is a max-heap.

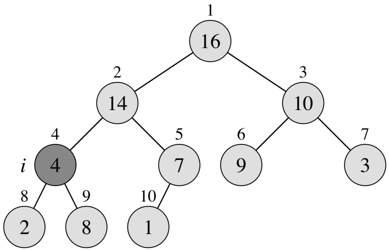
MAX-HEAPIFY(A, i, n)

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

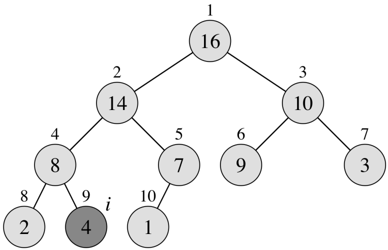
Max-Heapify



(a)



(b)



(c)

Time for `Max-Heapify`

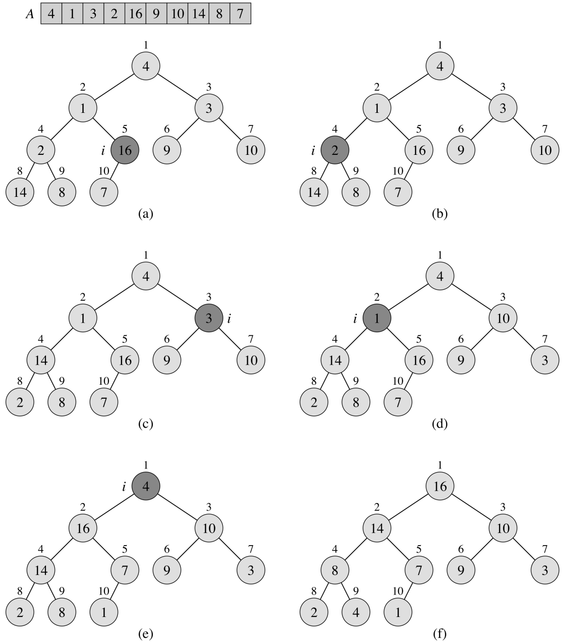
- $O(\lg n)$
- Why?

BUILD-MAX-HEAP

- If A is not a max-heap, this will make it one.

BUILD-MAX-HEAP(A, n)

```
for i = ⌊n/2⌋ downto 1
    MAX-HEAPIFY(A, i, n)
```


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

Loop invariant for Build-Max-Heap

- At start of every iteration of **for** loop, each node $i+1, i+2, \ldots, n$ is root of a max-heap.
  - Initialization?
  - Maintenance?
  - Termination?

```
BUILD-MAX-HEAP(A)
for i = ⌊n/2⌋ downto 1
    MAX-HEAPIFY(A, i, n)
```

Running time of BUILD-MAX-HEAP

- Loose bound: $O(n)$ calls to MAX-HEAPIFY, which is $O(\lg n)$, gives $O(n \lg n)$.
- We can get a tighter bound.
  - n element heap has height $\lfloor \lg n \rfloor$
  - n element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h
  - Time for MAX-HEAPIFY on a node of height h is $O(h)$
  - Total time for BUILD-MAX-HEAP:

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

Note:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \text{(A.8)}$$

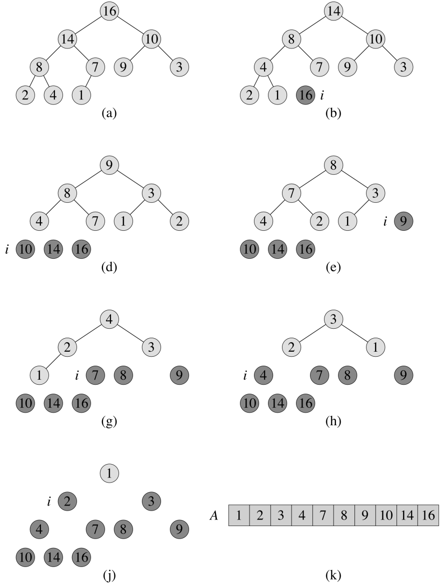
$$\sum_{k=0}^{\infty} k(1/2)^k = \frac{1/2}{(1-1/2)^2} = 2$$

Heapsort

- Builds max-heap in the array.
- Swaps the root (the maximum) with the element at the end.
- Heapifies the result, with one less element.
- Repeat until only one element left.

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

HEAPSORT



HEAPSORT

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

- Analysis: $O(n \lg n)$

Priority queue

- Heaps efficiently implement priority queues.
- Maintains a dynamic set S of elements.
- Each element has a *key*
- Operations:
  - INSERT(S, x)
  - MAXIMUM(S)
  - EXTRACT-MAX(S)
  - INCREASE-KEY(S, x, k)
- Min priority queue similar

Heap-Maximum

- Trivial
- Should probably check for empty heap

Heap-Maximum(A)

```
1  return A[1]
```

- $O(1)$

Heap-Extract-Max

```
Heap-Extract-Max(A)
if n < 1
    error "heap underflow"
max = A[1]
A[1] = A[n]
n = n - 1
Max-Heapify(A, 1, n)
return max
```

- $O(\lg n)$

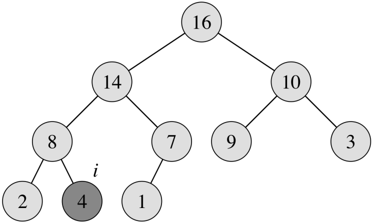
Heap-Increase-Key

- Make sure $k \geq x$'s current key
- Update x's key to k
- Traverse tree upward, swapping keys if necessary.

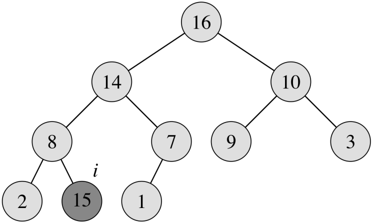
```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
        error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)
```

- $O(\lg n)$

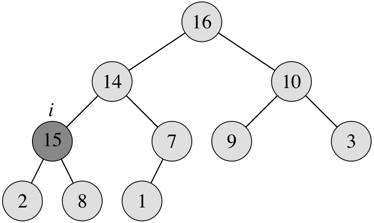
Heap-Increase-Key



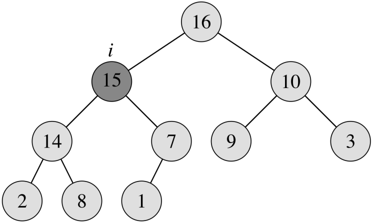
(a)



(b)



(c)



(d)

Max-Heap-Insert

```
Max-Heap-Insert(A, key, n)
  n = n + 1
  A[n] = −∞
  Heap-Increase-Key(A, n, key)
```

- $O(\lg n)$