

Problem Solving

2-11 Heap & Heapsort

MA Jun

Institute of Computer Software

May 7, 2020

Contents

- 1 Heaps
- 2 Heapsort
- 3 Priority Queue

Contents

1 Heaps

2 Heapsort

3 Priority Queue



Heap

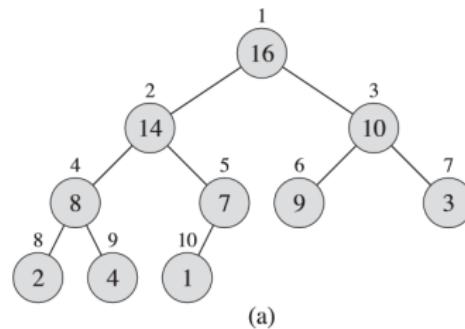
Heaps

The (binary) heap data structure is **an array object** that we can view as a **nearly complete binary tree**

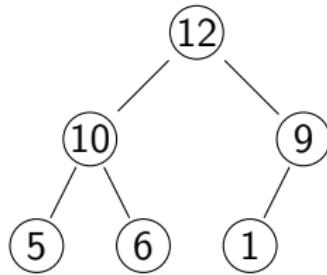
Heaps

The (binary) heap data structure is **an array object** that we can view as a **nearly complete binary tree**

- The tree is **completely** filled on all levels **except possibly** the **lowest**

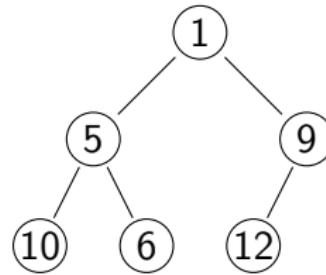


Heaps: Max-heap VS Min-heap



Max-heap property

$$A[\text{Parent}(i)] \geq A[i]$$



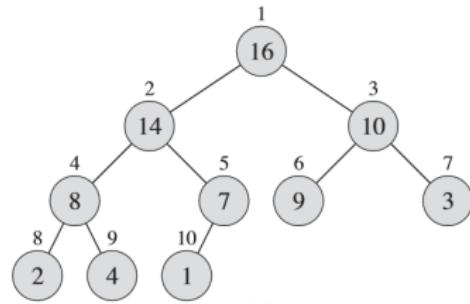
Min-heap property

$$A[\text{Parent}(i)] \leq A[i]$$

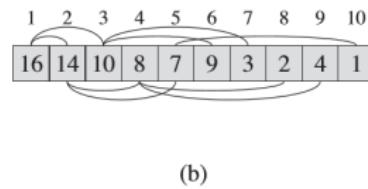


Heaps: Storage

Q-1: Why do we implement a heap with an array?



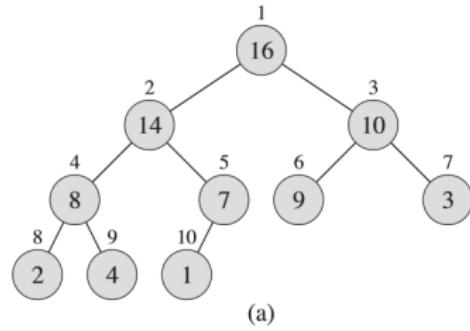
(a)



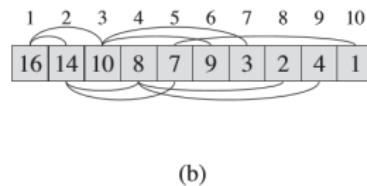
(b)

Heaps: Storage

Q-1: Why do we implement a heap with an array?



(a)



(b)

PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

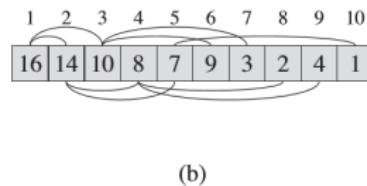
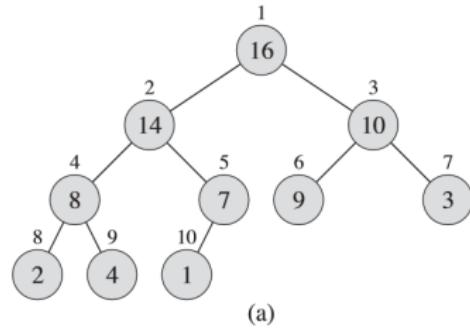
RIGHT(i)

1 **return** $2i + 1$

- Easy to index

Heaps: Storage

Q-1: Why do we implement a heap with an array?



PARENT(i)

1 **return** $\lfloor i/2 \rfloor$

LEFT(i)

1 **return** $2i$

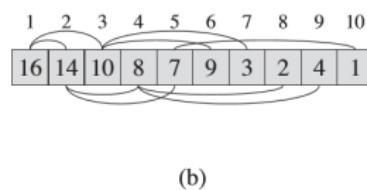
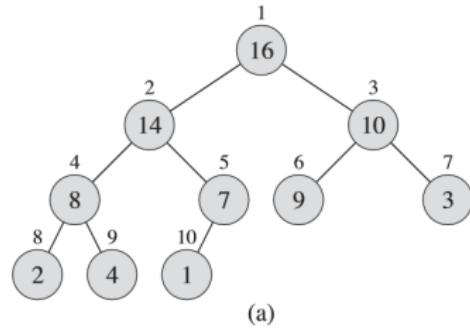
RIGHT(i)

1 **return** $2i + 1$

- Easy to index
- Save memory

Heaps: Storage

Q-1: Why do we implement a heap with an array?



PARENT(i)

1 **return** $[i/2]$

LEFT(i)

1 **return** $2i$

RIGHT(i)

1 **return** $2i + 1$

- Easy to index
- Save memory
- Better cache locality

Heaps: Height

The height of a node

- The number of edges on the **longest simple downward path** from the node to a leaf.

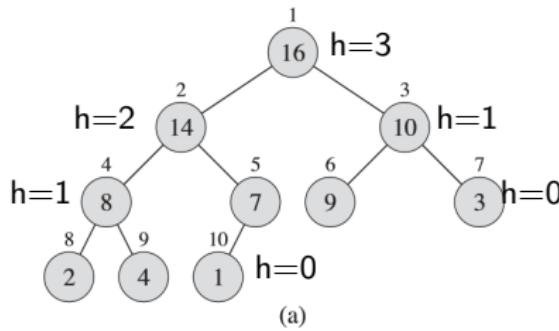
Heaps: Height

The height of a node

- The number of edges on the **longest simple downward path** from the node to a leaf.

The height of a heap

- The height of its root, $\Theta(\lg n)$.
- A heap of n elements is based on a **nearly complete binary tree**



Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

Heaps: basic operations

- The **MAX-HEAPIFY** procedure, which runs in $O(\lg n)$ time, is the key to maintaining the max-heap property.
- The **BUILD-MAX-HEAP** procedure, which runs in linear time, produces a max-heap from an unordered input array.
- The **HEAPSORT** procedure, which runs in $O(n \lg n)$ time, sorts an array in place.
- The **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, and **HEAP-MAXIMUM** procedures, which run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue.

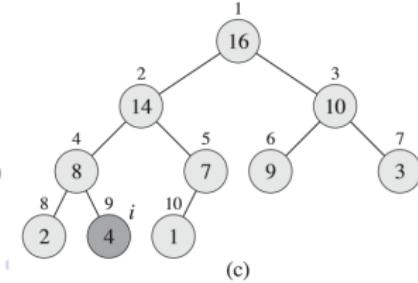
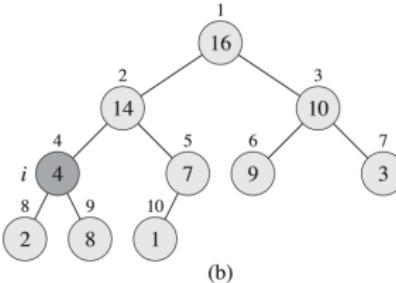
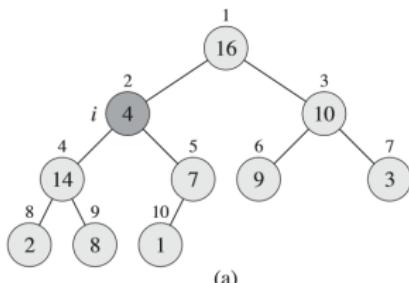


Maintaining the heap property: MAX-HEAPIFY

Q-2: Can you explain the process of MAX-HEAPIFY

MAX-HEAPIFY(A, i)

- 1 $l = \text{LEFT}(i)$
- 2 $r = \text{RIGHT}(i)$
- 3 **if** $l \leq A.\text{heap-size}$ and $A[l] > A[i]$
- 4 $\text{largest} = l$
- 5 **else** $\text{largest} = i$
- 6 **if** $r \leq A.\text{heap-size}$ and $A[r] > A[\text{largest}]$
- 7 $\text{largest} = r$
- 8 **if** $\text{largest} \neq i$
- 9 exchange $A[i]$ with $A[\text{largest}]$
- 10 MAX-HEAPIFY($A, \text{largest}$)



Maintaining the heap property: MAX-HEAPIFY

Q-2: Can you explain the process of MAX-HEAPIFY

MAX-HEAPIFY(A, i)

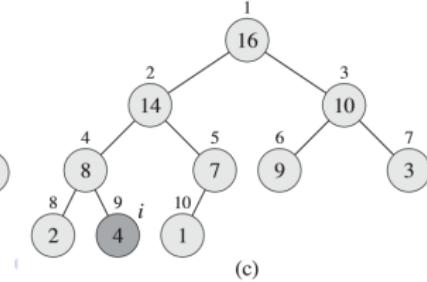
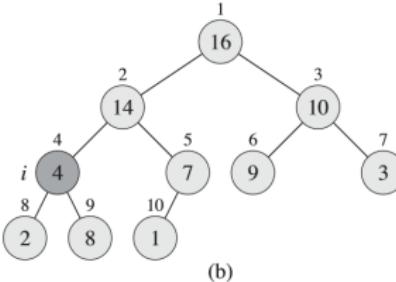
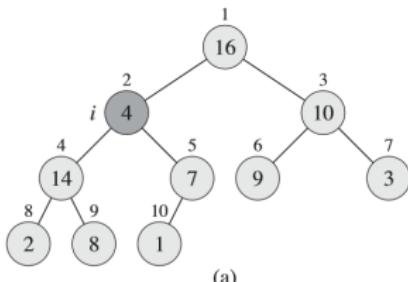
```

1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4     $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7     $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9    exchange  $A[i]$  with  $A[\text{largest}]$ 
10   MAX-HEAPIFY( $A, \text{largest}$ )

```

Pre-condition: $\text{Left}(i)$ and $\text{Right}(i)$ are maxheaps.

Post-condition: the tree rooted at i is a maxheap.



Maintaining the heap property: MAX-HEAPIFY

Q-2: Can you explain the process of MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```

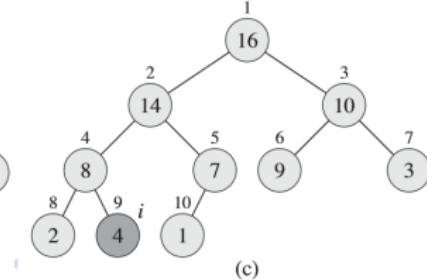
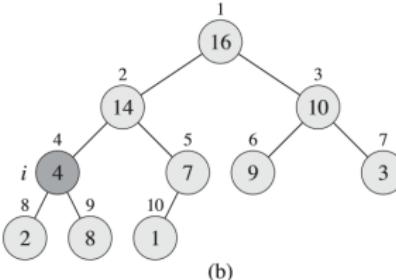
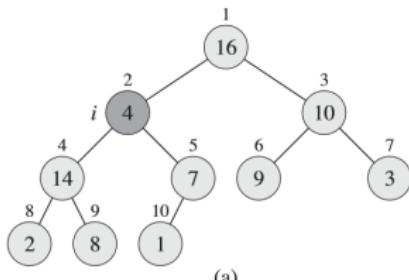
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
   largest =  $l$ 
4  else largest =  $i$ 
5  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
   largest =  $r$ 
6  if  $\text{largest} \neq i$ 
7    exchange  $A[i]$  with  $A[\text{largest}]$ 
8    MAX-HEAPIFY( $A, \text{largest}$ )
9
10

```

Pre-condition: $\text{Left}(i)$ and $\text{Right}(i)$ are maxheaps.

Post-condition: the tree rooted at i is a maxheap.

index of the largest element in the tree rooted at i



Maintaining the heap property: MAX-HEAPIFY

Q-2: Can you explain the process of MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```

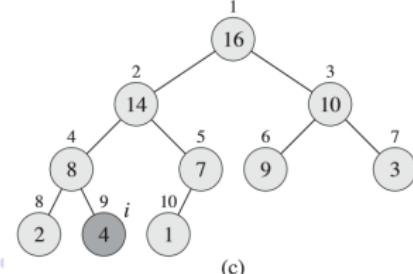
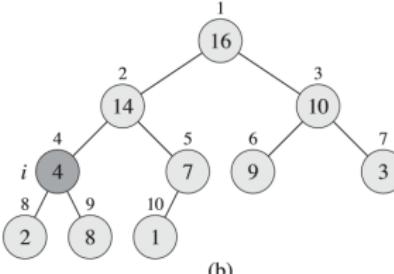
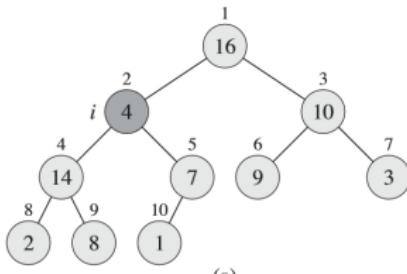
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
   largest =  $l$ 
4  else largest =  $i$ 
5  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
   largest =  $r$ 
6  if  $\text{largest} \neq i$ 
7    exchange  $A[i]$  with  $A[\text{largest}]$ 
8
9  MAX-HEAPIFY( $A, \text{largest}$ )
10

```

Pre-condition: $\text{Left}(i)$ and $\text{Right}(i)$ are maxheaps.

Post-condition: the tree rooted at i is a maxheap.

index of the largest element in the tree rooted at i



Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the sum of:

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
    $largest = l$ 
4  else  $largest = i$ 
5  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
    $largest = r$ 
6  if  $largest \neq i$ 
7    exchange  $A[i]$  with  $A[largest]$ 
8    MAX-HEAPIFY( $A, largest$ )
9
10   MAX-HEAPIFY( $A, largest$ )
```

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the sum of:

- Time to find the **largest**, $\Theta(1)$

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4     $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7     $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9    exchange  $A[i]$  with  $A[\text{largest}]$ 
10   MAX-HEAPIFY( $A, \text{largest}$ )
```

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the sum of:

- Time to find the largest, $\Theta(1)$
- Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$



Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4     $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7     $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9    exchange  $A[i]$  with  $A[\text{largest}]$ 
10   MAX-HEAPIFY( $A, \text{largest}$ )
```

The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the sum of:

- Time to find the largest, $\Theta(1)$
- Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$

$$T(n) \leq T(2n/3) + \Theta(1) = O(\lg n) = O(h)$$

Maintaining the heap property: MAX-HEAPIFY

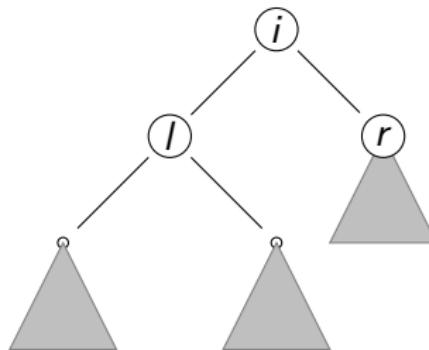
Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

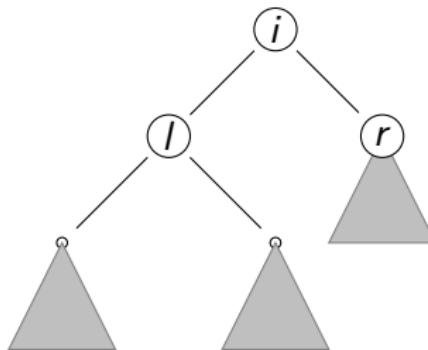
Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$



Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$

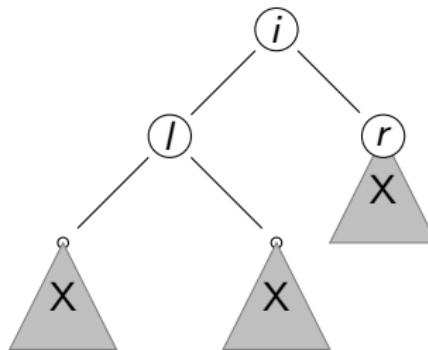


The subtree rooted at l is full

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$

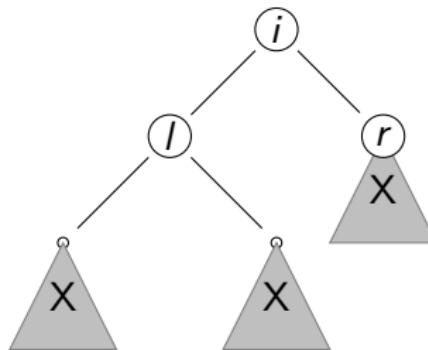


The subtree rooted at l is full

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$



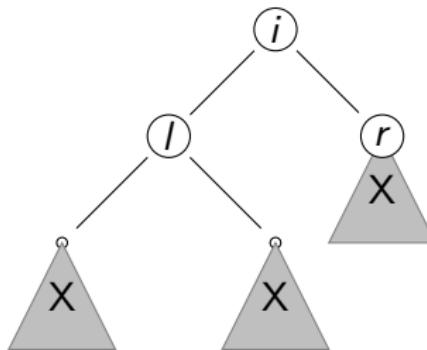
The subtree rooted at l is full

- Total number of nodes $n = 3X + 2$

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$



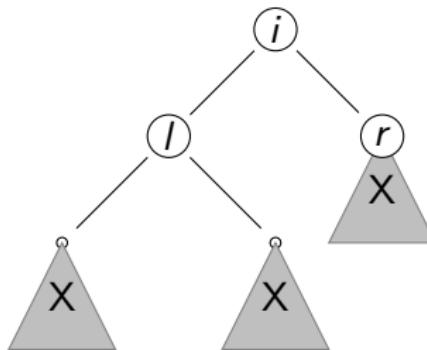
The subtree rooted at l is full

- Total number of nodes $n = 3X + 2$
- Total number of nodes in the left subtree $2X + 1$

Maintaining the heap property: MAX-HEAPIFY

Worst-case for MAX-HEAPIFY

Time to run MAX-HEAPIFY recursively, $\leq T(2n/3)$



The subtree rooted at l is full

- Total number of nodes $n = 3X + 2$
- Total number of nodes in the left subtree $2X + 1$
- $\frac{2X+1}{3X+2} < 2/3$

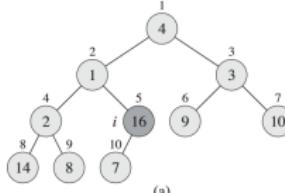
Building a heap: BUILD-MAX-HEAP

Q-3: Can you explain the process of
BUILD-MAX-HEAP

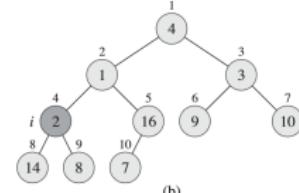
BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \lfloor A.\text{length}/2 \rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

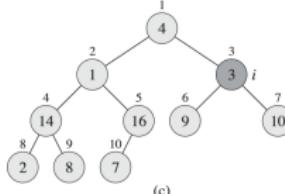
A [4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7]



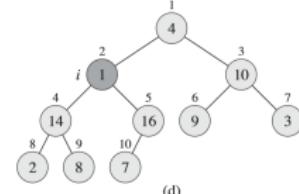
(a)



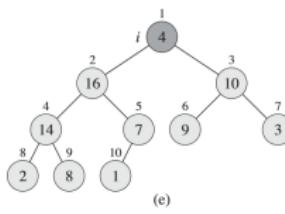
(b)



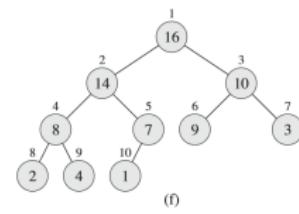
(c)



(d)



(e)



(f)

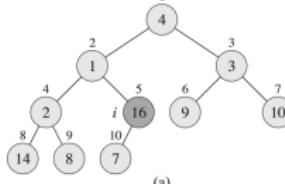
Building a heap: BUILD-MAX-HEAP

Q-3: Can you explain the process of
BUILD-MAX-HEAP

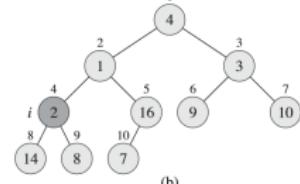
BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 for $i = \lfloor A.\text{length}/2 \rfloor$ downto 1
- 3 MAX-HEAPIFY(A, i)

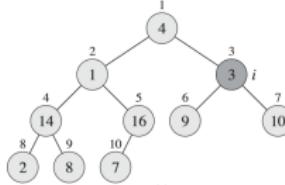
A	4	1	3	2	16	9	10	14	8	7
---	---	---	---	---	----	---	----	----	---	---



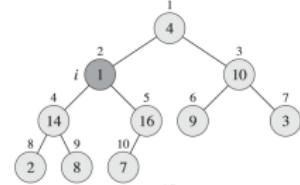
(a)



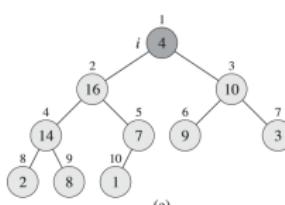
(b)



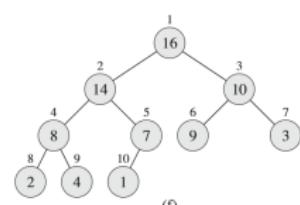
(c)



(d)



(e)



(f)

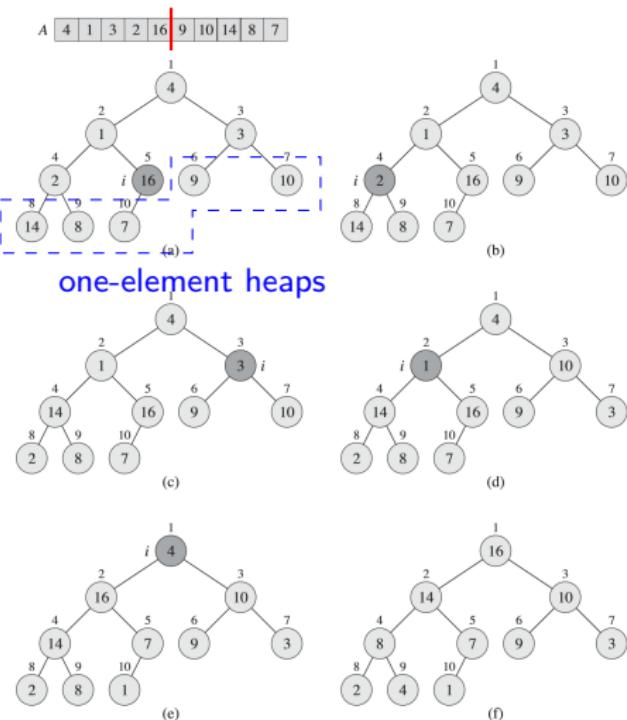


Building a heap: BUILD-MAX-HEAP

Q-3: Can you explain the process of
BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 for $i = \lfloor A.\text{length}/2 \rfloor$ down to 1
 - MAX-HEAPIFY(A, i)



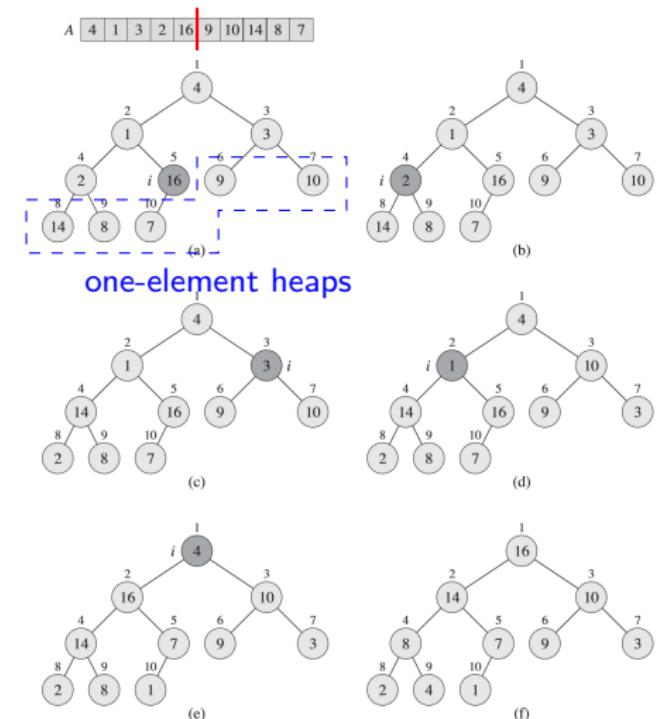
Building a heap: BUILD-MAX-HEAP

Q-3: Can you explain the process of
BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 for $i = \lfloor A.\text{length}/2 \rfloor$ down to 1
- 3 MAX-HEAPIFY(A, i)

Q-4: Can you prove the correctness of
BUILD-MAX-HEAP?



Correctness of BUILD-MAX-HEAP

Invariant

At the start of each iteration of the for loop of lines 2-3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

Correctness of BUILD-MAX-HEAP

Invariant

At the start of each iteration of the for loop of lines 2-3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

Proof.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call $\text{MAX-HEAPIFY}(A, i)$ to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \dots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap. In particular, node 1 is.



Correctness of BUILD-MAX-HEAP

Invariant

At the start of each iteration of the for loop of lines 2-3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

Proof.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call $\text{MAX-HEAPIFY}(A, i)$ to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \dots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap. In particular, node 1 is.



Correctness of BUILD-MAX-HEAP

Invariant

At the start of each iteration of the for loop of lines 2-3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

Proof.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \dots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap. In particular, node 1 is.



Correctness of BUILD-MAX-HEAP

Invariant

At the start of each iteration of the for loop of lines 2-3, each node $i + 1, i + 2, \dots, n$ is the root of a max-heap.

Proof.

Initialization: Prior to the first iteration of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf and is thus the root of a trivial max-heap.

Maintenance: To see that each iteration maintains the loop invariant, observe that the children of node i are numbered higher than i . By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes $i + 1, i + 2, \dots, n$ are all roots of max-heaps. Decrementing i in the **for** loop update reestablishes the loop invariant for the next iteration.

Termination: At termination, $i = 0$. By the loop invariant, each node $1, 2, \dots, n$ is the root of a max-heap. In particular, node 1 is.



Running time of BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \lfloor A.\text{length}/2 \rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

Running time of BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

- 1 $A.\text{heap-size} = A.\text{length}$
- 2 **for** $i = \lfloor A.\text{length}/2 \rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

A poor upper bound

- Each call to MAX-HEAPIFY costs $O(\lg n)$
- At most $O(n)$ calls
- Thus, $O(n \lg n)$

Running time of BUILD-MAX-HEAP

BUILD-MAX-HEAP(A)

```
1   $A.\text{heap-size} = A.\text{length}$ 
2  for  $i = \lfloor A.\text{length}/2 \rfloor$  downto 1
3    MAX-HEAPIFY( $A, i$ )
```

A poor upper bound

- Each call to MAX-HEAPIFY costs $O(\lg n)$
- At most $O(n)$ calls
- Thus, $O(n \lg n)$

Q-5: Can you give a better one?

Running time of BUILD-MAX-HEAP

A tighter linear upper bound

Running time of BUILD-MAX-HEAP

A tighter linear upper bound

- An n -element heap has height $\lfloor \lg n \rfloor$.

Running time of BUILD-MAX-HEAP

A tighter linear upper bound

- An n -element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height h .

Running time of BUILD-MAX-HEAP

A tighter linear upper bound

- An n -element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height h .
- Thus,

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

Running time of BUILD-MAX-HEAP

A tighter linear upper bound

- An n -element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height h .
- Thus,

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

Running time of BUILD-MAX-HEAP

A tighter linear upper bound

- An n -element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height h .
- Thus,

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

Running time of BUILD-MAX-HEAP

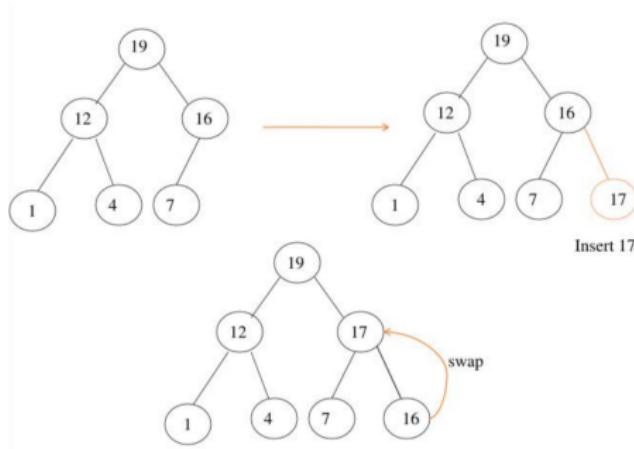
A tighter linear upper bound

- An n -element heap has height $\lfloor \lg n \rfloor$.
- At most $\lceil n/2^{h+1} \rceil$ nodes of any height h .
- Thus,

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

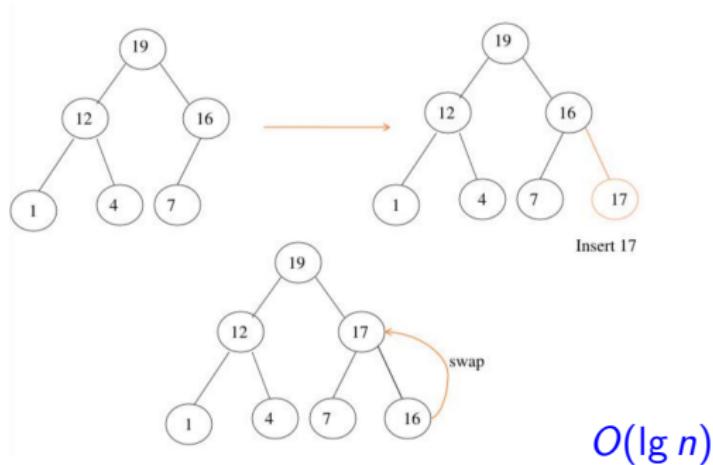
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \leq \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

Inserting an element into a Heap



- **Step-1:** Add the new element to the end of the heap
- **Step-2:** Compare the new element to its parent, if it is greater than its parent, swap the two elements
- **Step-3:** Repeat step-2 until the new element is smaller than its parent or it is the root.

Inserting an element into a Heap



- **Step-1:** Add the new element to the end of the heap
- **Step-2:** Compare the new element to its parent, if it is greater than its parent, swap the two elements
- **Step-3:** Repeat step-2 until the new element is smaller than its parent or it is the root.



Build heap with insertion?

- Insert $A[1, \dots, n]$ to a heap one by one.
- Complexity?

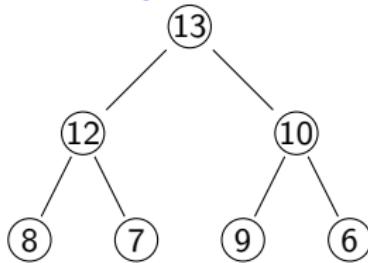
Deleting an element from a Heap

Assume that we try to delete a node i

Deleting an element from a Heap

Assume that we try to delete a node i

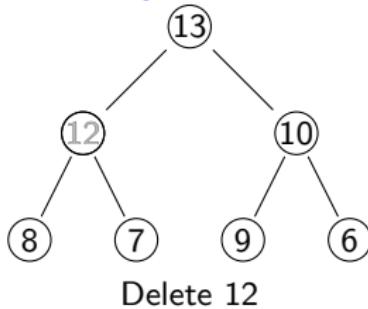
- **Step-1:** Copy the value of the last node to node i
- **Step-2:** Remove the last node
- **Step-3:** Call MAX-HEAPIFY on node i



Deleting an element from a Heap

Assume that we try to delete a node i

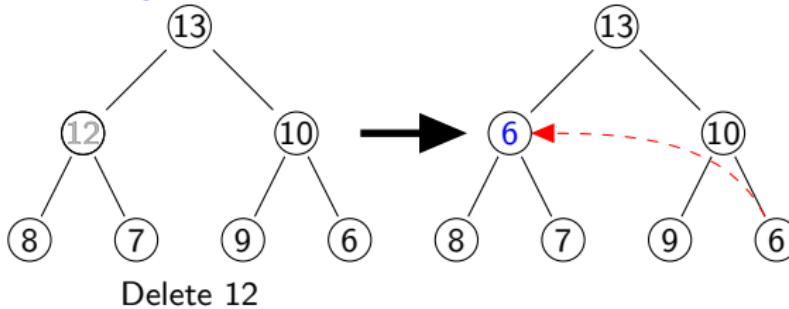
- **Step-1:** Copy the value of the last node to node i
- **Step-2:** Remove the last node
- **Step-3:** Call MAX-HEAPIFY on node i



Deleting an element from a Heap

Assume that we try to delete a node i

- **Step-1:** Copy the value of the last node to node i
- **Step-2:** Remove the last node
- **Step-3:** Call MAX-HEAPIFY on node i

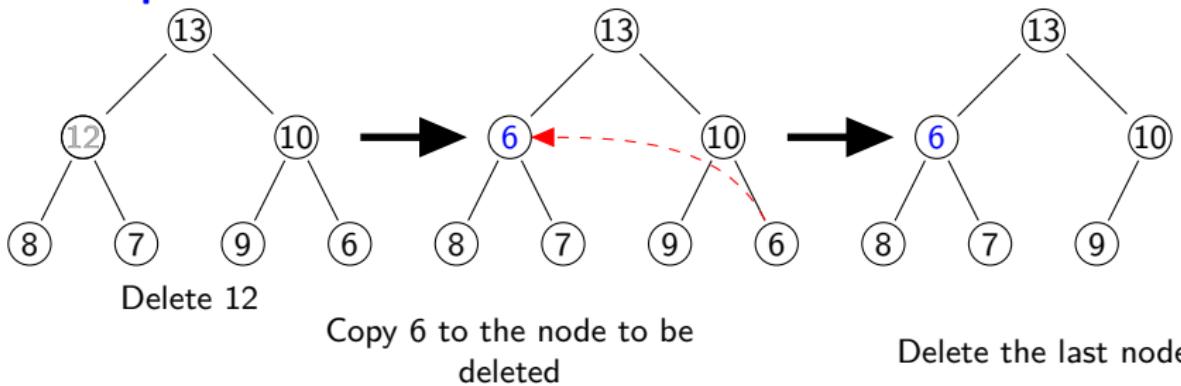


Copy 6 to the node to be deleted

Deleting an element from a Heap

Assume that we try to delete a node i

- **Step-1:** Copy the value of the last node to node i
- **Step-2:** Remove the last node
- **Step-3:** Call MAX-HEAPIFY on node i



Contents

1 Heaps

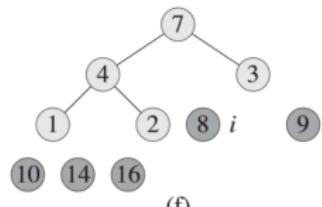
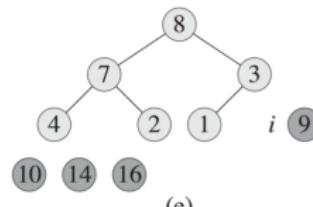
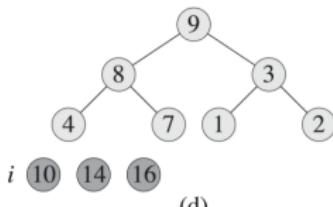
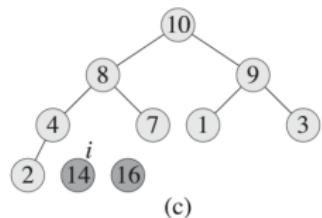
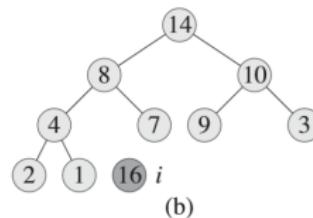
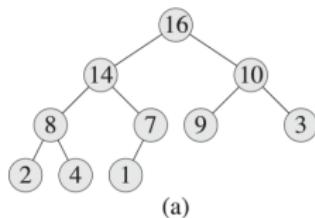
2 Heapsort

3 Priority Queue

Heapsort

$\text{HEAPSORT}(A)$

- 1 $\text{BUILD-MAX-HEAP}(A)$
- 2 **for** $i = A.\text{length}$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.\text{heap-size} = A.\text{heap-size} - 1$
- 5 $\text{MAX-HEAPIFY}(A, 1)$



Heapsort: Correctness

HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 **for** $i = A.length$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap-size = A.heap-size - 1$
- 5 MAX-HEAPIFY($A, 1$)

Q-6: How to prove the correctness of HEAPSORT?

Heapsort: Correctness

HEAPSORT(A)

- 1 BUILD-MAX-HEAP(A)
- 2 **for** $i = A.length$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.heap-size = A.heap-size - 1$
- 5 MAX-HEAPIFY($A, 1$)

Q-6: How to prove the correctness of HEAPSORT?

Loop Invariant (Exercise 6.4-2)

At the start of each iteration of the for loop of lines 2-5,

- the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$,
- the subarray $A[i + 1..n]$ contains the $n - i$ largest elements of $A[1..n]$, sorted.

In-place sorting

In-place sorting algorithms

Algorithms require $O(1)$ extra space and sorting is said to be happened in-place, or for example, within the array itself.

In-place sorting

In-place sorting algorithms

Algorithms require $O(1)$ extra space and sorting is said to be happened in-place, or for example, within the array itself.

- BUBBLE-SORT ✓
- INSERTION-SORT ✓
- HEAPSORT ✓
- MERGESORT ✓
- QUICKSORT X

Review QUICKSORT

Q-7: Why is QUICKSORT more efficient in practice?

Review QUICKSORT

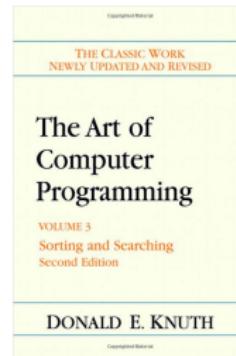
Q-7: Why is QUICKSORT more efficient in practice?

Based on a **fix computer model!**

- QUICKSORT: $11.667(n + 1) \ln n - 1.74n - 18.74$
- MERGESORT: $12.5n \ln n$
- HEAPSORT: $16n \ln n + 0.01n$
- INSERTION SORT: $2.25n^2 + 7.75n - 3 \ln n$



Donald Knuth



Review QUICKSORT

Q-7: Why is QUICKSORT more efficient in practice?

Analyze abstract basic operations! #swap & #comparison

- QUICKSORT: $2n \ln n$ comparisons and $\frac{1}{3}n \ln n$ swaps on average
- MERGESORT: $1.44n \ln n$ comparisons, but up to $8.66n/n(n)$ array accesses (mergesort is not swap based, so we cannot count that).
- INSERTIONSORT: $\frac{1}{4}n^2$ comparisons and $\frac{1}{4}n^2$ swaps on average.



Robert Sedgewick



<https://algs4.cs.princeton.edu/home/>



Contents

1 Heaps

2 Heapsort

3 Priority Queue

Priority Queue: ADT

Priority queue

A data structure for maintaining a set S of elements, each with an associated value called a **key**.

Priority Queue: ADT

Priority queue

A data structure for maintaining a set S of elements, each with an associated value called a **key**.

A **max-priority queue** supports the following operations:

$\text{INSERT}(S, x)$ inserts the element x into the set S , which is equivalent to the operation $S = S \cup \{x\}$.

$\text{MAXIMUM}(S)$ returns the element of S with the largest key.

$\text{EXTRACT-MAX}(S)$ removes and returns the element of S with the largest key.

$\text{INCREASE-KEY}(S, x, k)$ increases the value of element x 's key to the new value k , which is assumed to be at least as large as x 's current key value.

Priority Queue: ADT

Q-8: What is key difference between a Queue and a Priority-queue?

Queue

FIFO: First-In-First-Out

Priority Queue

- Order does not matter
- Priority matters



Priority Queue: Implementation

Heap → Priority Queue

HEAP-MAXIMUM(A)

- 1 **return** $A[1]$

HEAP-EXTRACT-MAX(A)

- 1 **if** $A.\text{heap-size} < 1$
- 2 **error** “heap underflow”
- 3 $\max = A[1]$
- 4 $A[1] = A[A.\text{heap-size}]$
- 5 $A.\text{heap-size} = A.\text{heap-size} - 1$
- 6 **MAX-HEAPIFY($A, 1$)**
- 7 **return** \max

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 $A[\text{PARENT}(i)] < A[i]$
- 5 exchange $A[i]$ with $A[\text{PARENT}(i)]$
- 6 $i = \text{PARENT}(i)$

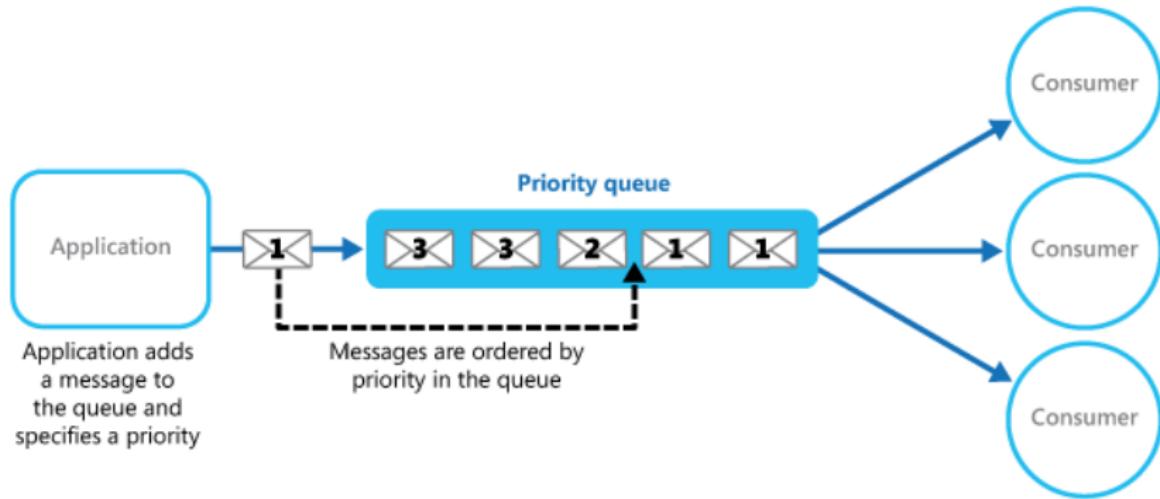
MAX-HEAP-INSERT(A, key)

- 1 $A.\text{heap-size} = A.\text{heap-size} + 1$
- 2 $A[A.\text{heap-size}] = -\infty$
- 3 **HEAP-INCREASE-KEY($A, A.\text{heap-size}, key$)**

Priority Queue: Applications

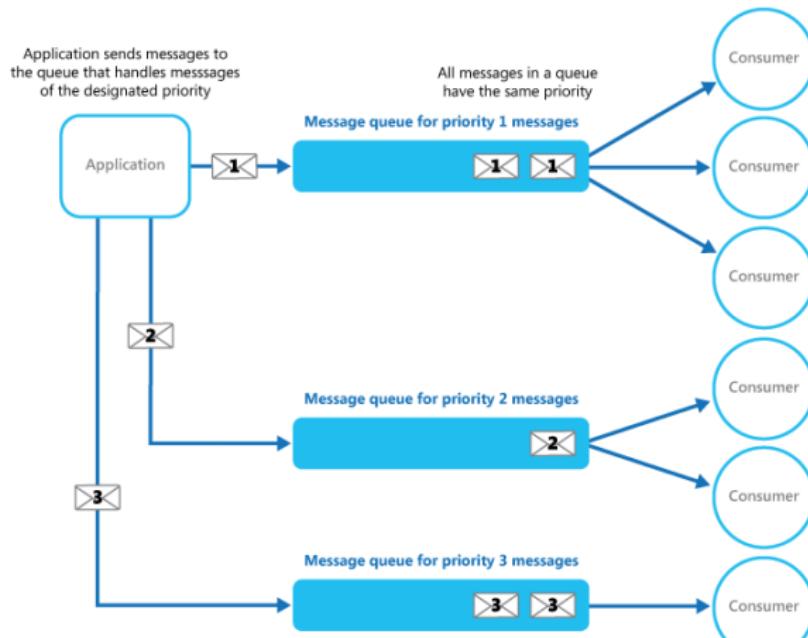
Message Queue

In the cloud, a message queue is typically used to delegate tasks to background processing.



Priority Queue: Applications

Message Queue (Without Priority Queue)



<https://docs.microsoft.com/en-us/azure/architecture/patterns/priority-queue>

Priority Queue: Applications

Dijkstra's Shortest Path Algorithm



Edsger W. Dijkstra.
1972 ACM A.M. Turing
Award winner

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

"I designed in about twenty minutes".

In 1956, "One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path"

Priority Queue: Applications

Prim Algorithm for MST

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Thank You!
Questions?

Office 819
majun@nju.edu.cn