ILLINOIS INSTITUTE OF TECHNOLOGY

Transforming Lives. Inventing the Future. www.iit.edu

CS430

Introduction to Algorithms

Lec 6

Lan Yao

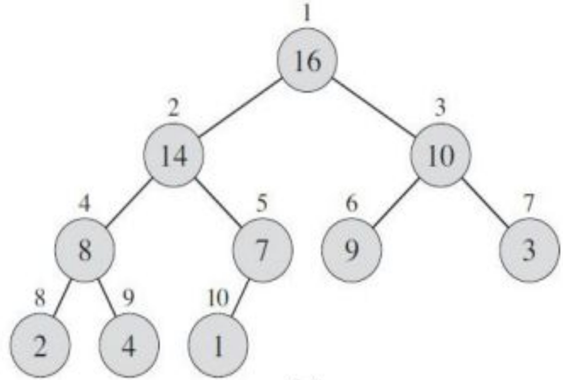
Outlines

- **Heap Sort**

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

Max-Heap

- (Binary)Heap
  - an almost complete binary tree
  - If a child's index is i, its parent's index= $\left\lfloor \frac{i}{2} \right\rfloor$;
  - If a parent's index is i, its left child's index=2i; its right child's index=2i+1.

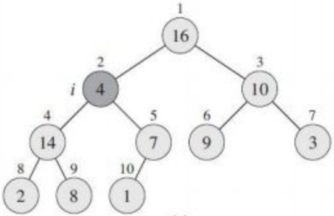


(a)

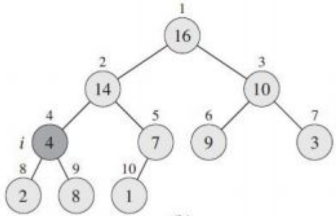
- Max-Heap: for any node i other than the root, A[parent(i)]>=A[i]

Max-Heapify

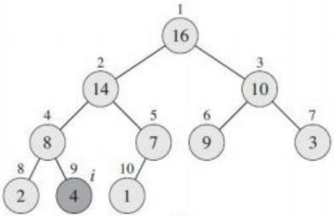
- a procedure to maintain a Max-Heap when a random element inserted
- input: a array A and newly inserted element A[i]. Both of A[i]'s children left(i) and right(i) are Max-Heaps. However, we do not know A[i] is greater than either of its children to satisfy Max-Heap property.
- output: a Max-Heap with A[i] and its children.



(a)



(b)



(c)

MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

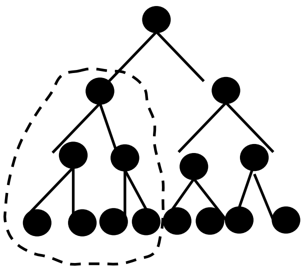
By Lan Yao

Complexity of Max-Heapify

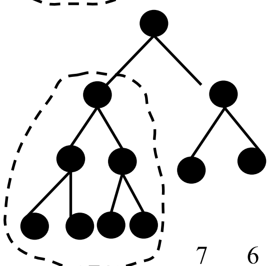
- Complexity: $$T(n) \leq T(\frac{2n}{3}) + \Theta(1)$$
- Proof
  - Consider the worst case--the the input size to a sub-tree is relatively greatest over n
  - The greater the size of the sub-tree's input is, the more complex the whole tree will be.

Heapsort

- The complexity of Max Heapify



The ratio of sub-problem to heap is 7:15=7/(7+7+1);
The general form of the ratio when the size of sub-problem is denote by k is k/(k+k+1) with the limit of 1/2.



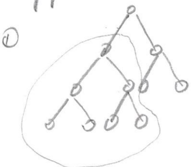
To maximize the ratio, maximize the size of sub-problem and minimize the size of the heap. It is 7/11=7/(7+4). The general form is 2^(n-1)/[2^(n−1) −1+2^n] with the limit of 2/3 when .

$$\frac{7}{11} > \frac{6}{10} > \frac{5}{9} > \frac{4}{8}$$

$$\frac{7}{11} > \frac{7}{12} > \frac{7}{13} > \frac{7}{14} > \frac{7}{15}$$

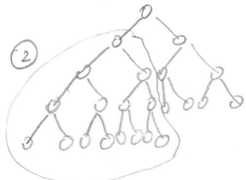
proof: (Recursion Tree)

suppose that we have the worst case :

① 

The tree is divided into a sub-tree with 7 nodes out of 11 nodes. The size of input is $\frac{7}{7+4}$.

g

② 

In this case, the size of input is : $\frac{15}{15+8}$.

Then we have : when height is 3 : $\frac{\text{sub}}{\text{tree}} = \frac{7}{7+4}$

; when height is 4 : $\cdots = \frac{15}{15+8}$

which gives us : when height is h : $\frac{2^h-1}{2^h-1+2^{h-1}}$,

while $h = \lfloor \lg n \rfloor$, plus it in $\Rightarrow$

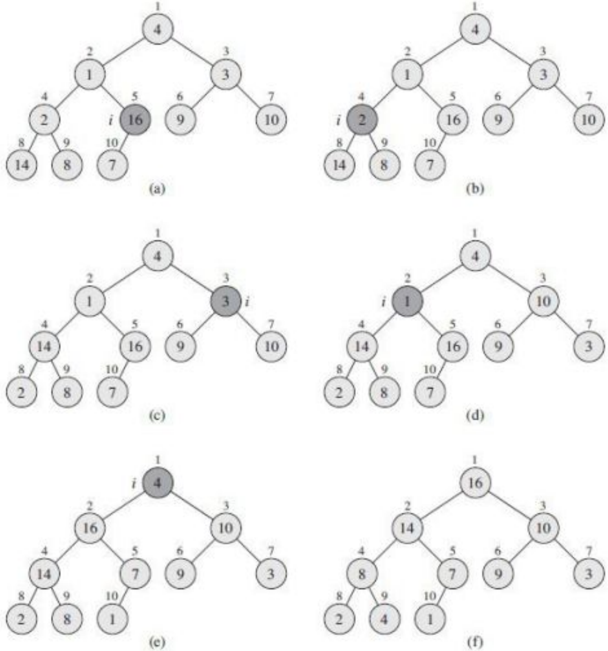
$$\frac{\text{sub}}{\text{tree}} = \frac{2^{\lfloor \lg n \rfloor}-1}{2^{\lfloor \lg n \rfloor}-1+2^{\lfloor \lg n \rfloor -1}} = \frac{2-\frac{1}{2^{\lfloor \lg n \rfloor -1}}}{2-\frac{1}{2^{\lfloor \lg n \rfloor -1}}+1}.$$

When $n \to \infty$, that is $\frac{2}{2+1} = \frac{2}{3}$.

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

Build a Max-Heap with Max-Heapify

- Suppose that we have a heap tree other than a max-heap
- Max-Heapify some nodes to adjust it into a Max-Heap
- You don't have to heapify leaves, because they don't have any child
- Heapifying starts with the index for the last parent node, which is floor of n/2




```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

By Lan Yao

- **Complexity of Build-Max-Heap**

- n-element heap has height of $\lfloor \lg n \rfloor$
- at any level, let h be the height of that level, then there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes.

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

$$when \quad h \to \infty, \quad \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2\ .$$

- **Complexity of Build-Max-Heap**

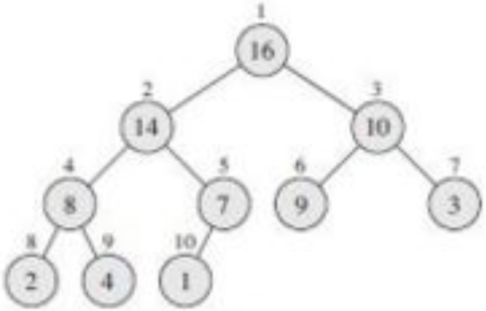
$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})$$

$$when \quad h \to \infty, \quad \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h(\frac{1}{2})^h = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 .$$

$$O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n \times 2) = O(n) \ .$$

Heapsort

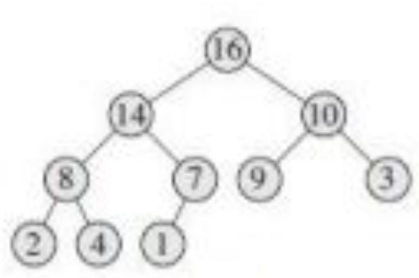
- Input: a Max-Heap;
- Output: sorted array;
- from the root down to leaves--why?



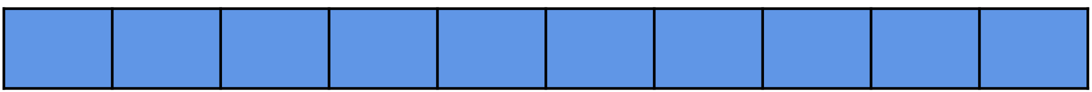
- **Heapsort**
  - direct thought
    - ✔ take the root off from the Max-Heap as the current maximum element of the array;
    - ✔ put it to the head of the array;
    - ✔ adjust the remined sub-trees to a Max-Heap;
    - ✔ recursively do previous steps.

- **Heapsort**

  Example:

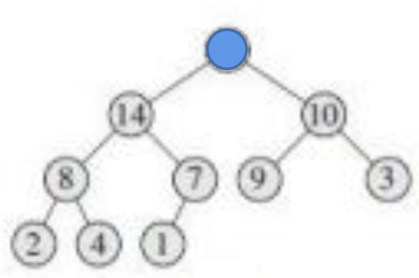


A



- **Heapsort**

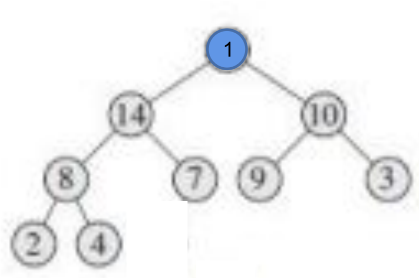
  Example:



A										16

- **Heapsort**

  Example:

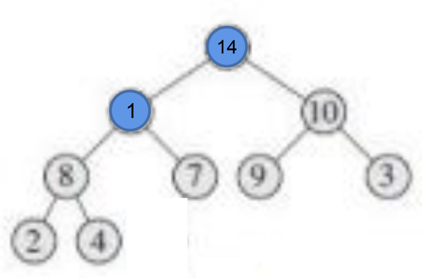


←Max-Heapify it!

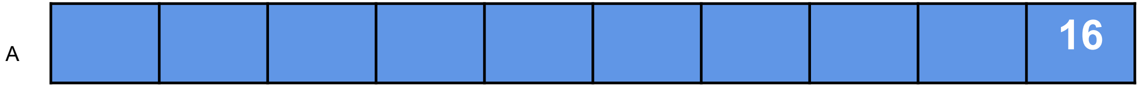
A										16

- **Heapsort**

  Example:

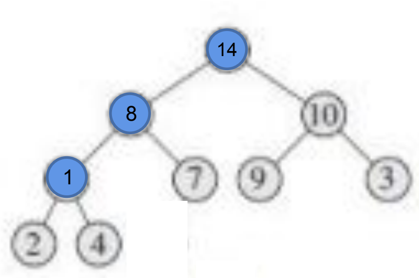


←Max-Heapify it!



- **Heapsort**

  Example:

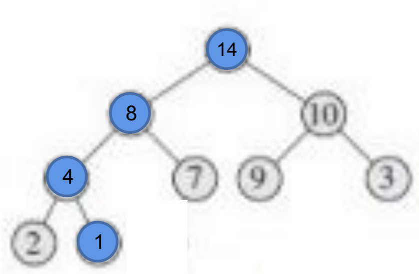


←Max-Heapify it!

A										16

- **Heapsort**

  Example:



←Max-Heapify it!

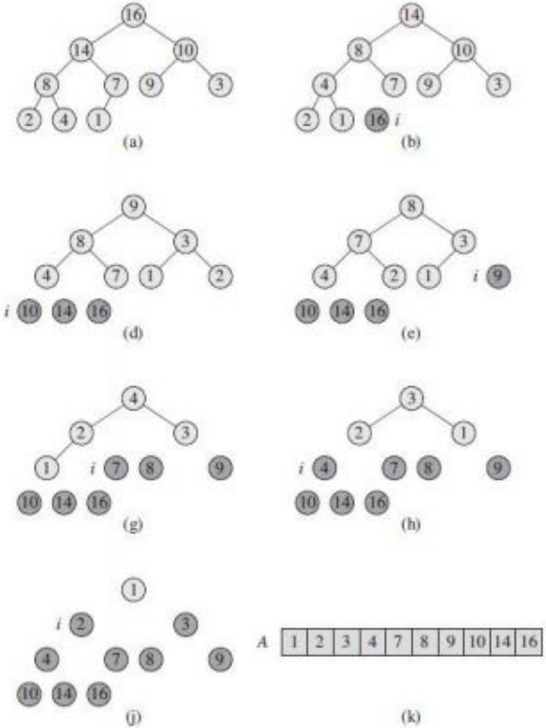
A										**16**

- **Heapsort**

Algorithm:

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
```

- **Heapsort**



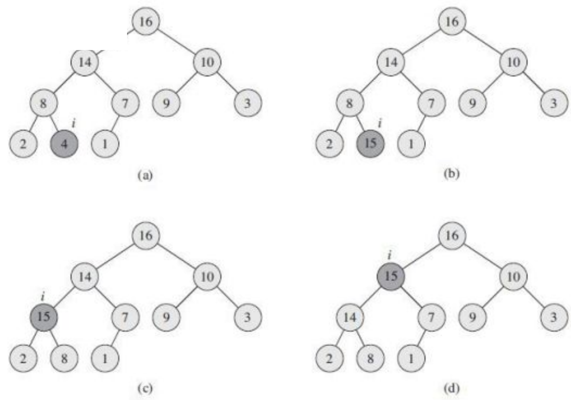
- Heap is a priority queue
- It supports the following operations:
    - Insert (S,x)--insert the element x into the set S, S=S U {x};
    - Maximum(S) returns the element of S with the largest key;
    - Extract-Max (S) removes and returns the element of S with the largest key;
    - Increase-Key (S, x, k) increase 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: Removing and Returning the Largest Element

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Priority Queue: Increasing the Value of an Element

```
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)
```



Priority Queue: Inserting a New Element

MAX-HEAP-INSERT(A, key)

$A.heap\text{-}size = A.heap\text{-}size + 1$

$A[A.heap\text{-}size] = -\infty$

HEAP-INCREASE-KEY$(A, A.heap\text{-}size, key)$