

COMP2013

Data Structures and Algorithms

Lecture 5

Binary Heap

Dr. Jieming Shi

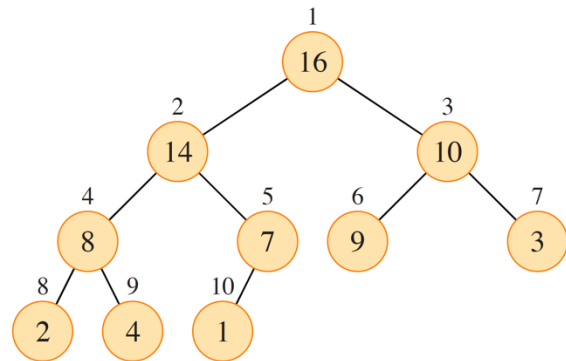


- **Heap**
 - **Heap concepts**
 - Build a heap
 - Heap analysis
- Heapsort
- Priority queue

(Binary) Heap

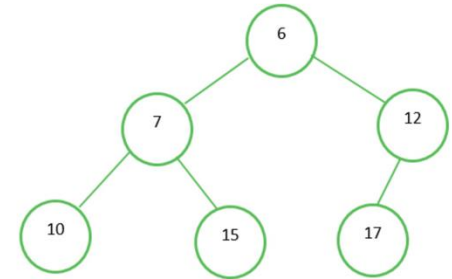
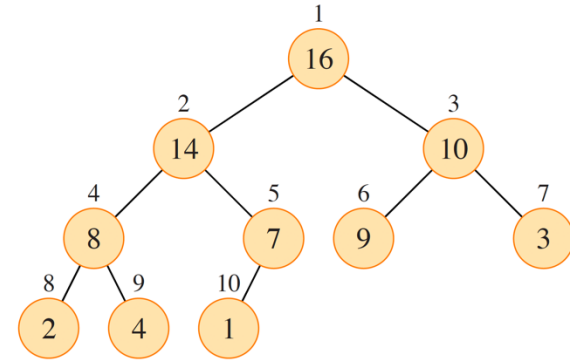


- *Understand* a heap as a complete binary tree
 - Nodes: Each with a key
 - Parent-child relationships
 - A node has 1 parent, except root
 - Subtree rooted at a node
 - Binary tree if every node has at most two children
 - The tree is completely filled on all levels (i.e., every node has two children) except possibly the lowest, which is filled from the left up to a point.
- **Max-Heap:**
 - The value of each node $\geq$ the values of its children
 - The root node contains the maximum value
 - As you move down the tree, the values decrease



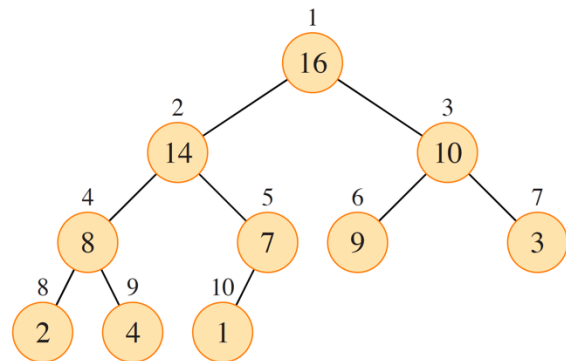


- **Max-Heap:**
 - The value of each node $\geq$ the values of its children
 - The root node contains the maximum value
 - As you move down the tree, the values decrease.
- **Min-Heap:**
 - The value of each node $\leq$ the values of its children
 - The root node contains the minimum value
 - As you move down the tree, the values increase.





- How to organize the heap?
 - Do we need to use complicated structures to maintain it?
- Just by an *Array*
- How?
 - Where are a node's children?
 - Where is a node's parent?

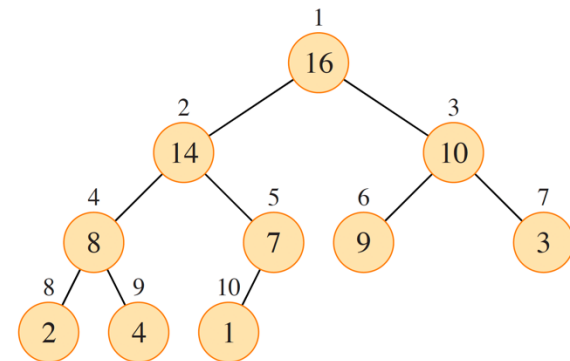


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap



- The array stores the values level by level
- Every level is completely filled, except the lowest filled from left up to a point right.
 - The number of values in a level is 2 times of its parent level, except the lowest level
- $A[1]$ is root
- For $A[i]$, where are its parent, left child, right child?



PARENT(i)

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

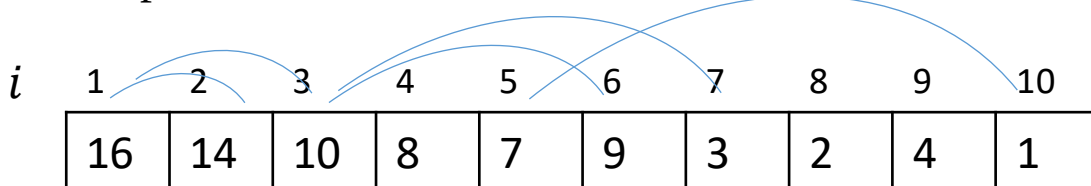
LEFT(i)

1 **return** $2i$

RIGHT(i)

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

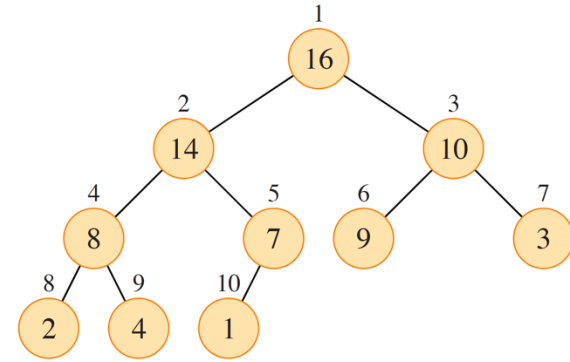
- Heap size: the number of elements in A



Max-heap and Min-heap

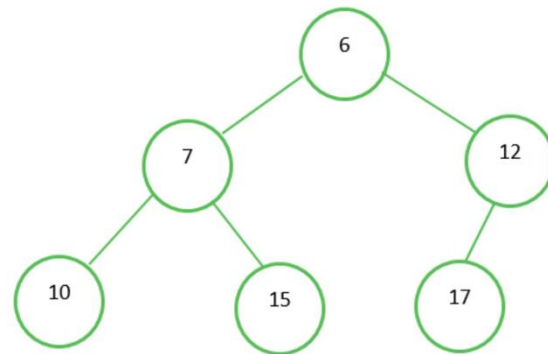


- Max-heap property
 - For every node i other than the root, its parent is no smaller than it
 - $A[\lfloor \frac{i}{2} \rfloor] \geq A[i]$
 - The largest element is at the root





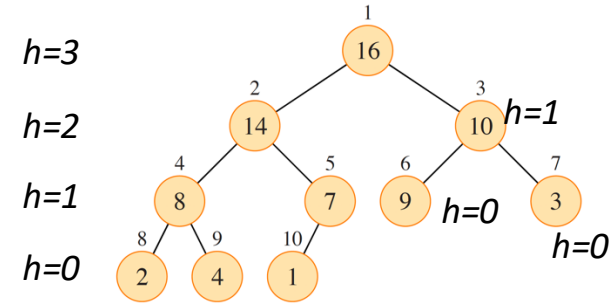
- Min-heap property
 - For every node i other than the root, its parent is no larger than it
 - $A[\lfloor \frac{i}{2} \rfloor] \leq A[i]$
 - The smallest element is at the root



Height of a Heap



- Height of a node is the number of edges on the *longest* simple downward path from it to a leaf
- Height of a tree is the height of its root
- Heap is regarded as a *complete* binary tree
- The height of a heap with size n is $\Theta(?)$
 - $\Theta(\lg n)$
 - How to prove this?



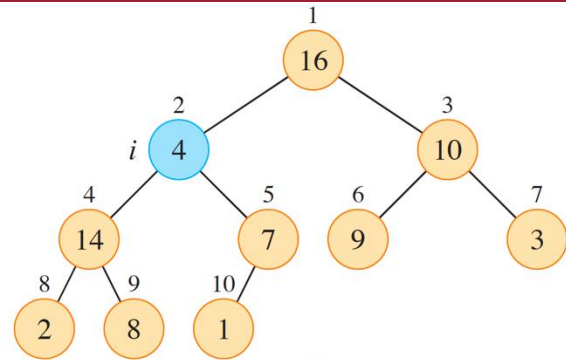


- Heap
 - Heap concepts
 - **Build a heap**
 - Heap analysis
- Heapsort
- Priority queue

Maintain the heap property



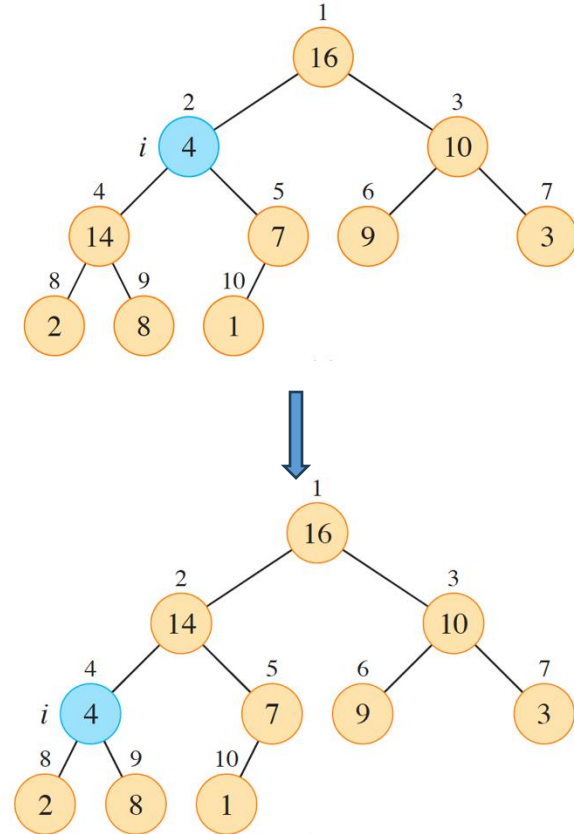
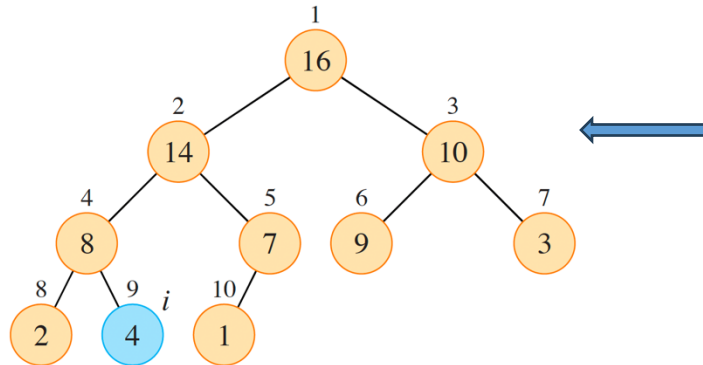
- $Max\text{-Heapify}(A, i)$
 - Left child $Left(i)$
 - The subtree rooted at $Left(i)$ is *max-heap*
 - Right child $Right(i)$
 - The subtree rooted at $Right(i)$ is *max-heap*
- $A[i]$ might violate the max-heap property
 - $A[i]$ might be smaller than its children or descendants
- This function fixes the issue to obey max-heap property



Maintain the heap property



- *Max-Heapify*(A, i)
 - $A[i]$ might violate the max-heap property
 - $A[i]$ might be smaller than its children or descendants
 - Move value at $A[i]$ downward
 - Swap it with its largest child.
 - Is this recursive?

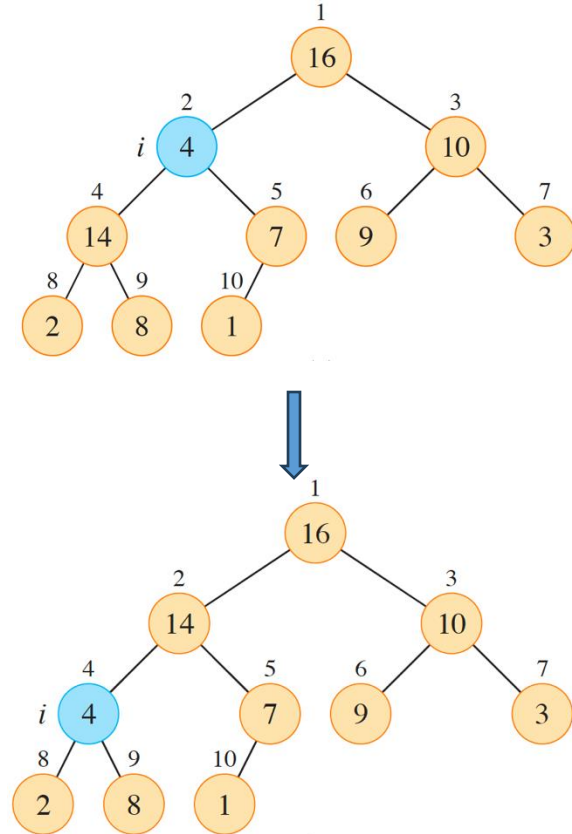
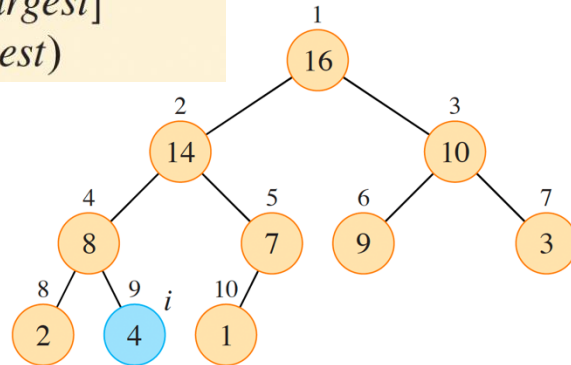


Maintain the heap property



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( $A, \text{largest}$ )
```



Maintain the heap property



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( $A, \text{largest}$ )
```

- How many levels should it move down at most ?
 - At most the height of the heap
 - $O(\lg n)$
- What is the cost per move?
 - $O(1)$
- Max-Heapify has running time
 - $O(\lg n)$

Build a Heap



- In heap $A[1:n]$, where are the leaves of the tree for the heap?
 - $A[\lfloor \frac{n}{2} \rfloor + 1 : n]$ are all leaves of the tree. Why?
 - Since for $i = \lfloor \frac{n}{2} \rfloor + 1$ to n , $2i$ and $2i + 1$ both are beyond n , meaning it does not have any child.
 - For $i = 1$ to $\lfloor \frac{n}{2} \rfloor$, $2i$ or $2i + 1$ or both are within n , meaning it has child(ren).
- A leaf node itself is a max-heap
- How to build a heap?
 - Bottom up and maintain the heap property

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

Build a Heap



BUILD-MAX-HEAP(A, n)

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

- Given an input unordered array
 - Regard it as a heap to recover its max-heap property
 - From the *lowest internal* level, to maintain the property upwards by calling *Max-Heapify*(A, i)

Build a Heap



Initial input A with $n=10$ elements

A

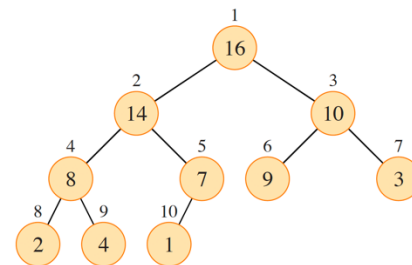
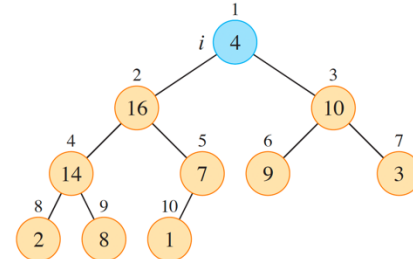
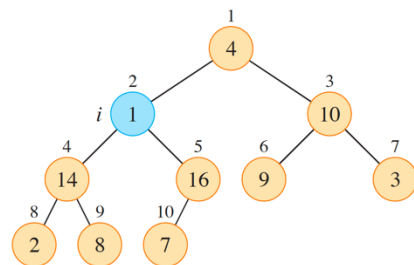
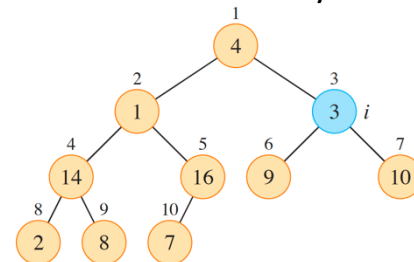
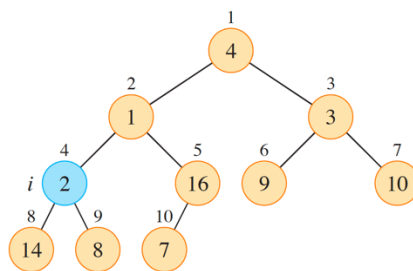
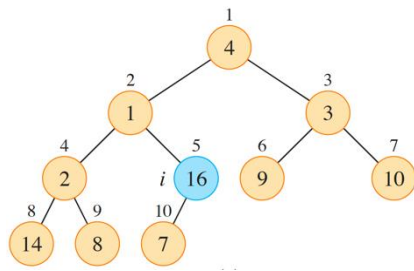
4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---

BUILD-MAX-HEAP(A, n)

```

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

Whenever *Max-Heapify* is called on a node, the two subtrees of that node are already both max-heaps





- Heap
 - Heap concepts
 - Build a heap
 - **Heap analysis**
- Heapsort
- Priority queue

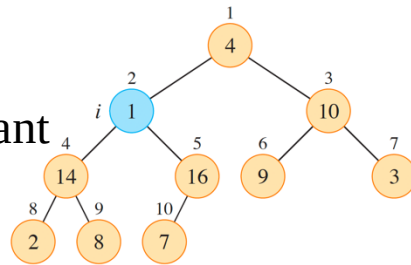
Prove an algorithm is correct by loop invariant



- Loop invariant
 - At the start of each iteration of the for loop (Lines 2-3),
 - each node $i + 1, \dots, n$ is the root of a max-heap
 - i.e. the nodes numbered higher than i are roots of max-heaps
- Initialization: show loop invariant is true before first loop
- Maintenance: show each i -th iteration maintains the loop invariant
- Termination: show the loop invariant is true after termination

BUILD-MAX-HEAP(A, n)

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



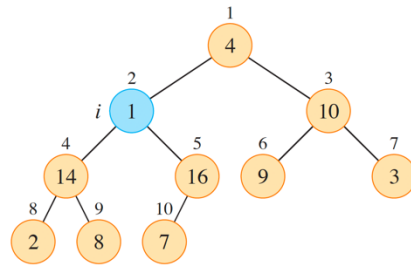
Prove an algorithm is correct by loop invariant



- Loop invariant
 - At the start of each iteration of the for loop (Lines 2-3),
 - each node $i + 1, \dots, n$ is the root of a max-heap
 - i.e. the nodes numbered higher than i are roots of max-heaps
- Initialization
 - Prior to the 1st iteration of the loop, $i = \lfloor n/2 \rfloor$,
 - each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is the root of a max-heap. Why?
 - They are leaf nodes
- Maintenance
 - The children of node i are with indexes larger. They are both roots of max-heaps at the start of i -th iteration based on the loop invariant.
 - This is precisely the condition to call *Max-Heapify*, after which, node i is made a max-heap root.
 - After i -th iteration, i is decreased by 1, which reestablishes the loop invariant for next iteration.
- Termination
 - $i=0$. By the loop invariant, each node numbered larger than i are roots of max-heaps.

BUILD-MAX-HEAP(A, n)

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



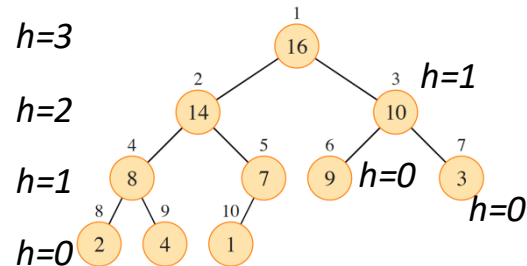
Complexity to build a heap



- Max-Heapify has running time $O(\log n)$
- Max-Heapify is called $O(n)$ times,
- The complexity is $O(n \log n)$
- Is this correct?
 - Yes
- Is this tight?
 - No
- Can we have a tighter $O(?)$
 - The time of Max-Heapify varies with the height of the node i
 - The heights of most nodes are small since it is a tree

BUILD-MAX-HEAP(A, n)

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



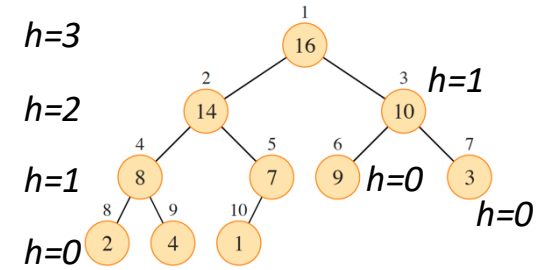
Complexity to build a heap



- Can we have a tighter $O(?)$
 - The time of Max-Heapify varies with the height of the node i
 - The heights of most nodes are small since it is a tree
- Let H be the height of the heap
- Let n_h be the number of nodes with height h
- Max-heapify has complexity $O(h)$ for a node with height h
- Total time to build a heap is
 - $\sum_{h=0}^H n_h \cdot ch$
 - What are H and n_h ?

BUILD-MAX-HEAP(A, n)

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



Complexity to build a heap

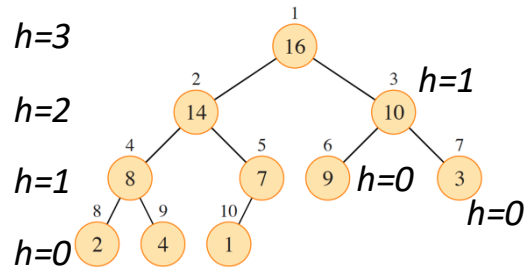


- Can we have a tighter $O(?)$
- Total time to build a heap is $\sum_{h=0}^H n_h \cdot ch$
 - What are H and n_h ?
- A heap is a complete binary tree
 - Height H is $\lfloor \lg n \rfloor$
 - In a heap of size n , at most $\lfloor n/2^{h+1} \rfloor$ nodes are of height h
 - Prove this by induction, contradiction, etc
 - $n_h \leq \lfloor n/2^{h+1} \rfloor$
- Max-heapify has complexity $O(h)$
- Total time to build a heap is
 - $\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor ch$
 - $\leq \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{n}{2^h} ch$
 - $\leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} = cn \frac{1/2}{(1-1/2)^2}$
 - **$O(n)$**

BUILD-MAX-HEAP(A, n)

```

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



$$\lfloor x \rfloor \leq 2x, \text{ for any } x \geq 1/2$$

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

for $|x| < 1$.



- Heap
 - Heap concepts
 - Build a heap
 - Heap analysis
- **Heapsort**
- Priority queue

Heapsort



HEAPSORT(A, n)

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

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

Make it a heap in linear time $O(n)$



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

- The max element of a heap is at root $A[1]$
- The idea is to get the max element out from the heap and restore the remaining elements to be a heap again

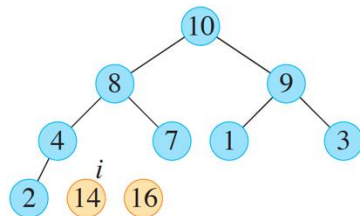
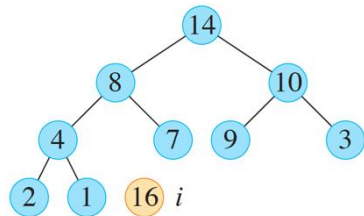
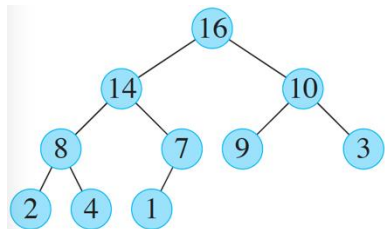
Heapsort



HEAPSORT(A, n)

```

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



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

Make it a heap in linear time $O(n)$



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

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

$i=10$

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

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

$i=9$

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

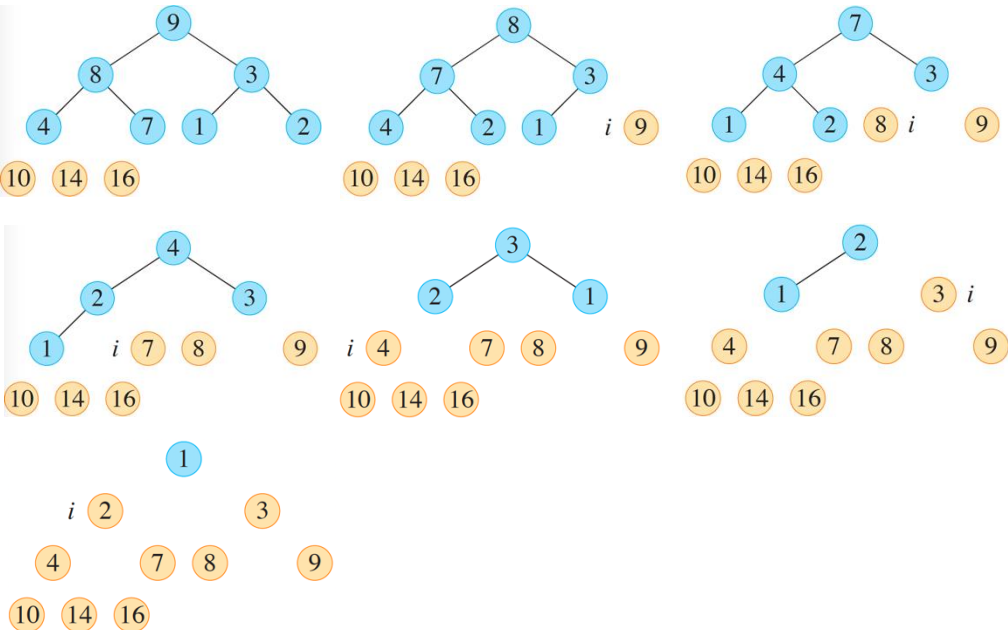
Heapsort



HEAPSORT(A, n)

```

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



After $i=9$

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

$i=8$

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

$i=7$

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

$i=6$

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

$i=5$

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

$i=4$

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

$i=3$

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

$i=2$

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

Heapsort



HEAPSORT(A, n)

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

Running time of Heapsort:

Line 1: ?

Line 2:

Line 3:

Line 4:

Line 5:?

$O(n \log n)$

Heapsort



- Heapsort is in place
- Merge sort and heapsort are asymptotically optimal comparison sorts; Quicksort achieves this on average. (worst-case lower-bound running time $\Omega(n \log n)$)
- *Heap as a data structure helps to get the max element in $O(\lg n)$ time, which is more efficient than insertion sort, bubble sort, and selection sort that scan arrays in a vanilla way*

Algorithm	Worst-case running time	Average/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$

Selection Sort (in C++)



- For $i=0, \dots, n-2$,
 - Find the minimum elements in $A[i+1:n-1]$ and swap it with $A[i]$

```
void selectionSort(int arr[], int n)
{
    // One by one move boundary of
    // unsorted subarray
    for (int i = 0; i < n - 1; i++)
    {
        // Find the minimum element in
        // unsorted array
        int min_idx = i;
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }

        // Swap the found minimum element
        // with the first element
        if (min_idx != i)
            swap(arr[min_idx], arr[i]);
    }
}
```

	0	1	2	3	4
i=0	3	8	4	2	6
i=1	2	8	4	3	6
i=2	2	3	4	8	6
i=3	2	3	4	8	6
	2	3	4	6	8

Running time?

$\Theta(n^2)$

Bubble Sort (in C++)



- Scan the array sequentially
 - Swap adjacent elements if they are not in the ascending order
- Repeat the above until no swaps are needed

```
// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // If no two elements were swapped
        // by inner loop, then break
        if (swapped == false)
            break;
    }
}
```

Running time?

$$\Theta(n^2)$$

3	8	4	2	6
---	---	---	---	---

3	4	8	2	6
---	---	---	---	---

3	4	2	8	6
---	---	---	---	---

3	4	2	6	8
---	---	---	---	---

3	4	2	6	8
---	---	---	---	---

3	2	4	6	8
---	---	---	---	---

3	2	4	6	8
---	---	---	---	---

3	2	4	6	8
---	---	---	---	---

2	3	4	6	8
---	---	---	---	---

2	3	4	6	8
---	---	---	---	---

$i=0$

$i=1$

$i=2$



- Selection sort and bubble sort are in place
- Insertion sort, bubble sort and selection sort are not efficient on large inputs

Algorithm	Worst-case running time	Average/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Heapsort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Bubble sort	$\Theta(n^2)$	$\Theta(n^2)$
Selection sort	$\Theta(n^2)$	$\Theta(n^2)$



- Heap
 - Heap concepts
 - Build a heap
 - Heap analysis
- Heapsort
- **Priority queue**



- A set S of elements, each $x \in S$ with priority level $x.key$
- An element with high priority is dequeued before an element with low priority
- Priority examples:
 - age (older with higher priority)
 - price cost (lower with higher priority)
 - ...

Priority queue operations



- Top
 - Get the element with the highest priority
 - *Get the root of the heap*
- Dequeue
 - Get and remove the element with the highest priority from the queue
 - *Get and remove the root of the heap, and then resort heap property*
- Insert
 - Insert an element into the priority queue
 - *Insert an element to the heap and resort heap property*
- How to implement a priority queue?
 - Heap is one possible way



- Max-priority queues
 - Based on max-heaps
 - Larger key has higher priority
- Min-priority queues
 - Based on min-heaps
 - Smaller key has higher priority
- In what follows, we explain max-priority queues

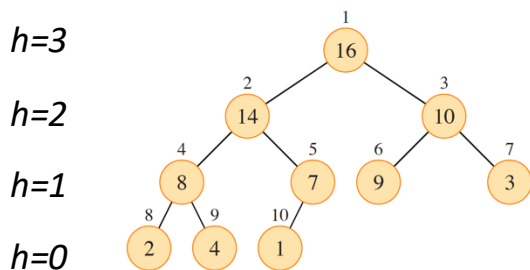


- Get the element with the highest priority
- Given a heap A ,
 - Return $A[1]$ if the heap is not empty
- Running time $O(1)$

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



- Get and remove the element with the highest priority from the queue
- Running time $O(\log n)$



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

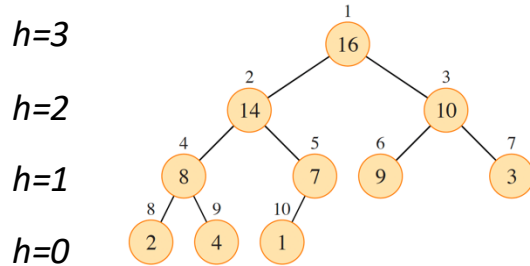
Deque(Heap A)

1. if $A.\text{heapsize} < 1$
2. return -1
3. $\text{max} = A[1]$
4. $A[1] = A[A.\text{heapsize}]$
5. $A.\text{heapsize}--$
6. Max-Heapify(A, 1)
7. return max

Change priority of an element



- `changePriority(i, p)`
 - Change the priority of element i to p
 - The new priority p can be lower or higher than the old priority
 - Example: change $i=2$ from 14 to 4



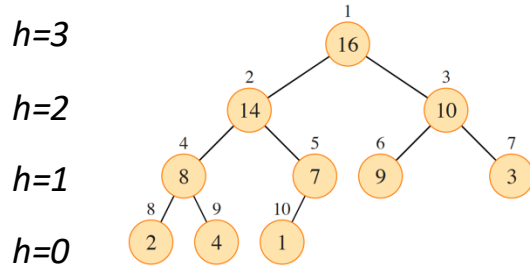
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

```
changePriority(Heap A, i, p)
1. oldp = A[i]
2. A[i]=p
3. if p<oldp //shift down
4.   Max-Heapify(A, i)
5. else if p>oldp //shift up
6.   while i>1 and A[parent(i)]<A[i]
7.     swap(A[parent(i)], A[i])
8.     i=parent(i)
```

Change priority of an element



- `changePriority(i, p)`
 - Change the priority of element i to p
 - The new priority p can be lower or higher than the old priority
 - Example: change $i=9$ from 4 to 15

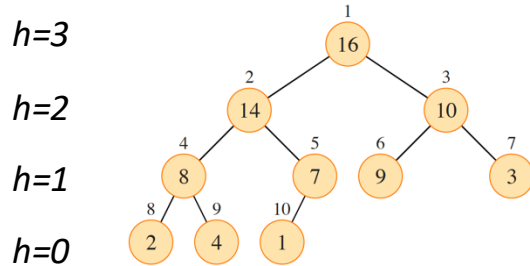


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

```
changePriority(Heap A, i, p)
1. oldp = A[i]
2. A[i]=p
3. if p<oldp //shift down
4.   Max-Heapify(A, i)
5. else if p>oldp //shift up
6.   while i>1 and A[parent(i)]<A[i]
7.     swap(A[parent(i)], A[i])
8.     i=parent(i)
```




- Insert an element with priority p into the priority queue
 - For example, insert 15 into the heap
 - Put it at the end of the heap array and shift up



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

Insert(Heap A, p)

1. $A.\text{heapsize}++$
2. $A[\text{heapsize}] = p$
3. $i = \text{heapsize}$
4. while $i > 1$ and $A[\text{parent}(i)] < A[i]$
5. $\text{swap}(A[\text{parent}(i)], A[i])$
6. $i = \text{parent}(i)$



- Thank you!