Note for the students:

- These slides are meant for the lecturers to conduct lectures only. It is NOT suitable to be used as a study material.
- Students are expected to study by reading the textbook for this course:
Sorting algorithm 3

Chapter 6

Sorting algorithm 1 → insertion sort
Sorting algorithm 2 → merge sort
Objectives

* Priority Queues
* Heaps
* Heapsort
Priority Queue

* A data structure implementing a set $S$ of elements, each associated with a key, supporting the following operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{insert}(S, x)$</td>
<td>insert element $x$ into set $S$</td>
</tr>
<tr>
<td>$\text{max}(S)$</td>
<td>return element of $S$ with largest key</td>
</tr>
<tr>
<td>$\text{extract_max}(S)$</td>
<td>return element of $S$ with largest key and remove it from $S$</td>
</tr>
<tr>
<td>$\text{increase_key}(S, x, k)$</td>
<td>increase the value of element $x$’ s key to new value $k$</td>
</tr>
</tbody>
</table>
* Like merge sort, but unlike insertion sort.
* Like insertion sort, but unlike merge sort.
* Heap sort’s running time is $O(n \lg n)$.

Like merge sort, the worst case time of heap sort is $O(n \lg n)$ and like insertion sort, heap sort sorts in-place.
The (binary) heap data structure is an array object that can be viewed as a nearly complete binary tree.

Each node of the tree corresponds to an element of the array that stores the value in the node.

A heap is an almost-complete binary tree.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
An array $A$ that represents a heap is an object with two attributes: $\text{length}[A]$, which is the number of elements in the array, and $\text{heap-size}[A]$, the number of elements in the heap stored within array $A$.

That is, although $A[1 \ldots \text{length}[A]]$ may contain valid numbers, no element past $A[\text{heap-size}[A]]$, where $\text{heap-size}[A] \leq \text{length}[A]$, is an element of the heap.
Heap as a Tree

- **Parent(i)**: return \( \lfloor i/2 \rfloor \)
- **Left(i)**: return \( 2i \)
- **Right(i)**: return \( 2i + 1 \)

root of tree: first element in the array, corresponding to \( i = 1 \)
parent(i) = \( \lfloor i/2 \rfloor \): returns index of node's parent
left(i) = 2i: returns index of node's left child
right(i) = 2i + 1: returns index of node's right child
Kinds of binary heaps

* There are two kinds of binary heaps:
  * max-heaps and
  * min-heaps
* In both kinds, the values in the nodes satisfy a **heap property, the specifics of which depend on** the kind of heap.
* In a **max-heap, the max-heap property is that for every node** \( i \) other than the root 
  \[
  A[\text{PARENT}(i)] \geq A[i],
  \]
In a max-heap, the max-heap property is that for every node other than the root

\[ A[\text{PARENT}(i)] \geq A[i], \]

the value of a node is at most the value of its parent.

the largest element in a max-heap is stored at the root, and the sub-tree rooted at a node contains values no larger than that contained at the node itself.
A **min-heap** is organized in the opposite way; the **min-heap property is that for every node other than the root**, 

\[ A[\text{PARENT}(i)] \leq A[i] \]

* The smallest element in a min-heap is at the root.
The height of a node in a heap

- The **height of a node in a heap** to be the number of edges on the longest simple downward path from the node to a leaf, and
- The **height of the heap** to be the height of its root.
- Since a heap of *n* elements is based on a complete binary tree, its height is $\Theta(lg \, n)$.
- the basic operations on heaps run in time at most proportional to the height of the tree and thus take $O(lg \, n)$ time.
Minimum number of nodes in a binary tree whose height is $h$.

At least one node at each level
Maximum Number Of Nodes

All possible nodes at first $h$ levels are present

Maximum number of nodes

$$= 1 + 2 + 4 + 8 + \ldots + 2^h = 2^{h+1} - 1$$
6.1-1
What are the minimum and maximum numbers of elements in a heap of height $h$?

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

6.1-2
Show that an $n$-element heap has height $\lceil \lg n \rceil$.

Given an $n$-element heap of height $h$, we know from Exercise 6.1-1 that

$$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}.$$ 

Thus, $h \leq \lg n < h + 1$. Since $h$ is an integer, $h = \lceil \lg n \rceil$ (by definition of $\lceil \cdot \rceil$).
### HEAP 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_heapify</td>
<td>Maintaining the max-heap property</td>
</tr>
<tr>
<td>build_max_heap</td>
<td>produce a max-heap from an unordered array</td>
</tr>
</tbody>
</table>
Maintaining the heap property

MAX-HEAPIFY(A, i)
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if \( l \leq \text{heap-size}[A] \) and \( A[l] > A[i] \)
4    then largest ← l
5  else largest ← i
6  if \( r \leq \text{heap-size}[A] \) and \( A[r] > A[\text{largest}] \)
7    then largest ← r
8  if largest ≠ i
9    then exchange \( A[i] \) ↔ \( A[\text{largest}] \)
10  MAX-HEAPIFY(A, largest)
The children’s sub-trees each have size at most $2n/3$.

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

The solution to this recurrence, by case 2 of the master theorem (Theorem 4.1), is $T(n) = O(lg n)$. 

Computer Sciences Department
Build_Max_Heap(A) Analysis
Building a heap (a bottom-up manner)

BUILD-MAX-HEAP(A)
1 heap-size[A] ← length[A]  
2 for i ← [length[A]/2] downto 1  
3 do MAX-HEAPIFY(A, i)

Why start at $\left\lfloor \text{length}/2 \right\rfloor$?
\[ \sum_{h=0}^{[\lg n]} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O \left( n \sum_{h=0}^{[\lg n]} \frac{h}{2^h} \right). \]

The last summation can be evaluated by substituting \( x = 1/2 \) in the formula (A.8), which yields
\[
\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2.
\]

Thus, the running time of \textsc{Build-Max-Heap} can be bounded as
\[
O \left( n \sum_{h=0}^{[\lg n]} \frac{h}{2^h} \right) = O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n).
\]
The heap-sort algorithm

HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
4   heap-size[A] ← heap-size[A] − 1
5  MAX-HEAPIFY(A, 1)

BUILD-MAX-HEAP(A) takes $\Theta(n)$, MAX-HEAPIFY(A,1) takes $\Theta(\lg n)$ and repeated $n-1$ times.???
Sorting Strategy:

1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[n] and A[1]: now max element is at the end of the array!
4. Discard node n from heap (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
6. Go to Step 2 unless heap is empty.
(a) 16
   14 10
     8 7 9 3
   2 4 1

(b) 14
   8 10
     7 9 3
   2 1 16

(c) 10
   8 9
   4 7 1 3
   2 14 16

(d) 9
   8 3
     7 1 2
   4

(e) 8
   7 3
     2 1 9
   4

(f) 7
   4 3
     1 2 8
   10 14 16
Why does heapsort run in $\Theta(n \log n)$ time?

Let us count operations line by line.

1. Construct the heap in linear time. “In particular, the BuildHeap procedure actually runs in $\Theta(n)$ time”.
2. Execute the loop and perform a logarithmic time operation $n-1$ times.
3. Other operations take constant time.

Hence, your running time is

$$n + (n - 1) \log n + O(1)$$
$$= n + n \log n - \log n + O(1)$$
$$= \Theta(n \log n).$$

- The For loop is $(n-1)\log n$ and BUILD-MAX-HEAP(A) should have been just added
  - BUILD-MAX-HEAP(A) takes $\Theta(n)$, MAX-HEAPIFY(A,1) takes $\Theta(\log n)$ and repeated $n-1$ times
Heapsort

The running time of BUILD-MAX-HEAP = $O(n)$

Recursion-tree method
\[
\sum_{h=0}^{\left\lfloor \log n \right\rfloor} h \times 2^{\left\lfloor \log n \right\rfloor - h} = \sum_{h=0}^{\left\lfloor \log n \right\rfloor} h \times \frac{2^{\left\lfloor \log n \right\rfloor}}{2^h} \leq \sum_{h=0}^{\left\lfloor \log n \right\rfloor} h \times \frac{2^{\log n}}{2^h} = \sum_{h=0}^{\left\lfloor \log n \right\rfloor} h \times \frac{n^{1/2}}{2^h}
\]
\[ \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left( \frac{n}{2^{h+1}} \right) \leq O(h) \]

\[ O \left( n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n) \]

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

\[ = \frac{1/2}{(1-1/2)^2} = 2 \]

using formula (A.8) Integrating and differentiating series

for \( x = 1/2 < 1 \)
Conclusion

Sorting Strategy:

1. Build Max Heap from unordered array;

2. Find maximum element A[1];

3. Swap elements A[n] and A[1]:
   now max element is at the end of the array!

4. Discard node $n$ from heap
   (by decrementing heap-size variable)

5. New root may violate max heap property, but its
   children are max heaps. Run max_heapify to fix this.

6. Go to Step 2 unless heap is empty.
Conclusion

There are two factors at work: the time it takes to create a heap by adding each element and the time it takes to remove all of the elements from a heap.

Running time:

\[
\text{after } n \text{ iterations the Heap is empty every iteration involves a swap and a max_heapify operation; hence it takes } O(\log n) \text{ time}
\]

worst-case efficiency

Overall \(O(n \log n)\)

A naive implementation requires additional space, but it is possible to do a heap sort in place. Heap sort has guaranteed \(O(n \log n)\) performance, though the constant factor is typically a bit higher than for other algorithms such as quicksort. Heap sort is not a stable sort.
For additional information see:

http://www.cprogramming.com/tutorial/computersciencetheory/heapsort.html

http://www.cprogramming.com/tutorial/computersciencetheory/heapcode.html