WIA2005: Algorithm Design and Analysis

Semester 2, Session 2016/17

Lecture 6: Heaps and Heap Sort

Learning Objectives

- Heaps
  - Max-heapify
  - Build-Max-Heap
  - Heapsort
  - Priority queue operation

Heapsort

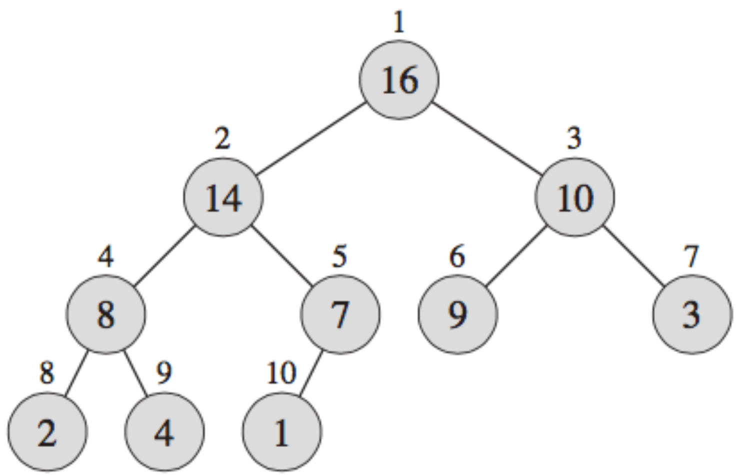
- Like merge sort, but unlike insertion sort, heapsort's running time is O(n lg n).
- Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.
- Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.
- Heapsort also introduces another algorithm design technique: using a data structure, in this case one we call a "heap," to manage information.

Heap

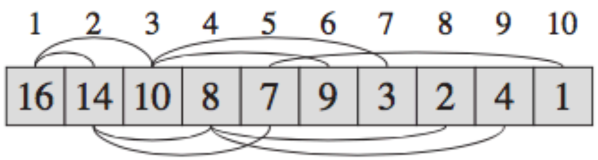
- The ***(binary) heap*** data structure is an array object that we can view as a nearly complete binary tree.
- Each node of the tree corresponds to an element of the array.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.
- An array A that represents a heap is an object with two attributes:
  - A.*length*, which (as usual) gives the number of elements in the array.
  - A.*heap-size*, which represents how many elements in the heap are stored within array A.

Max-heap

- A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.



(a)



(b)

Computing indices

- The root of the tree is A[1], and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

- The LEFT procedure can compute 2i in one instruction by simply shifting the binary representation of i left by one bit position.
- Similarly, the RIGHT procedure can quickly compute 2i+1 by shifting the binary representation of i left by one bit position and then adding in a 1 as the low-order bit.
- The PARENT procedure can compute floor(i/2) by shifting i right one bit position.

Heaps property

- There are two kinds of binary heaps:
  - max-heaps
  - min-heaps
- Max-heaps property:

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

- Min-heaps property:

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

Basic operation on heaps

- MAX-HEAPIFY
- BUILD-MAX-HEAP
- HEAPSORT
- MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM – for priority queue

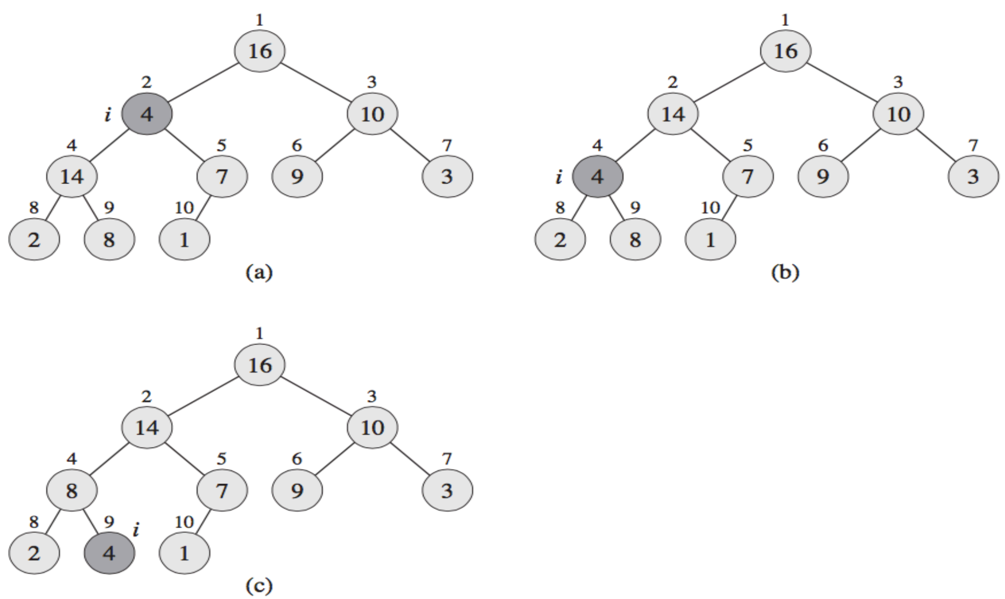
Maintaining the heap property

- In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY.

```
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)
```

Visualizing MAX-HEAPIFY

- The action of MAX-HEAPIFY(A,2), where A.*heap-size* = 10. **(a)** The initial coniguration, with A[2] at node i = 2 violating the max-heap property since it is not larger than both children.
- The max-heap property is restored for node 2 in **(b)** by exchanging A[2] with A[4], which destroys the max-heap property for node 4.
- The recursive call MAX-HEAPIFY(A,4) now has i = 4. After swapping A[4] with A[9], as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY(A,9) yields no further change to the data structure.



Running time for MAX-HEAPIFY

- The running time of MAX-HEAPIFY on a subtree of size n rooted at a given node i is the Θ(1) time to fix up the relationships among the elements A[i], A[LEFT(i)], and A[RIGHT(i)], plus the time to run MAX-HEAPIFY on a subtree rooted at one of the children of node i (assuming that the recursive call occurs).
- The running time of MAX-HEAPIFY by the recurrence:

$$T(n) \le T(2n/3) + \Theta(1)$$

T(n) = O(lg n)

Case 2 of Masters Theorem

- Alternatively, we can characterize the running time of MAX-HEAPIFY on a node of height h as O(h).

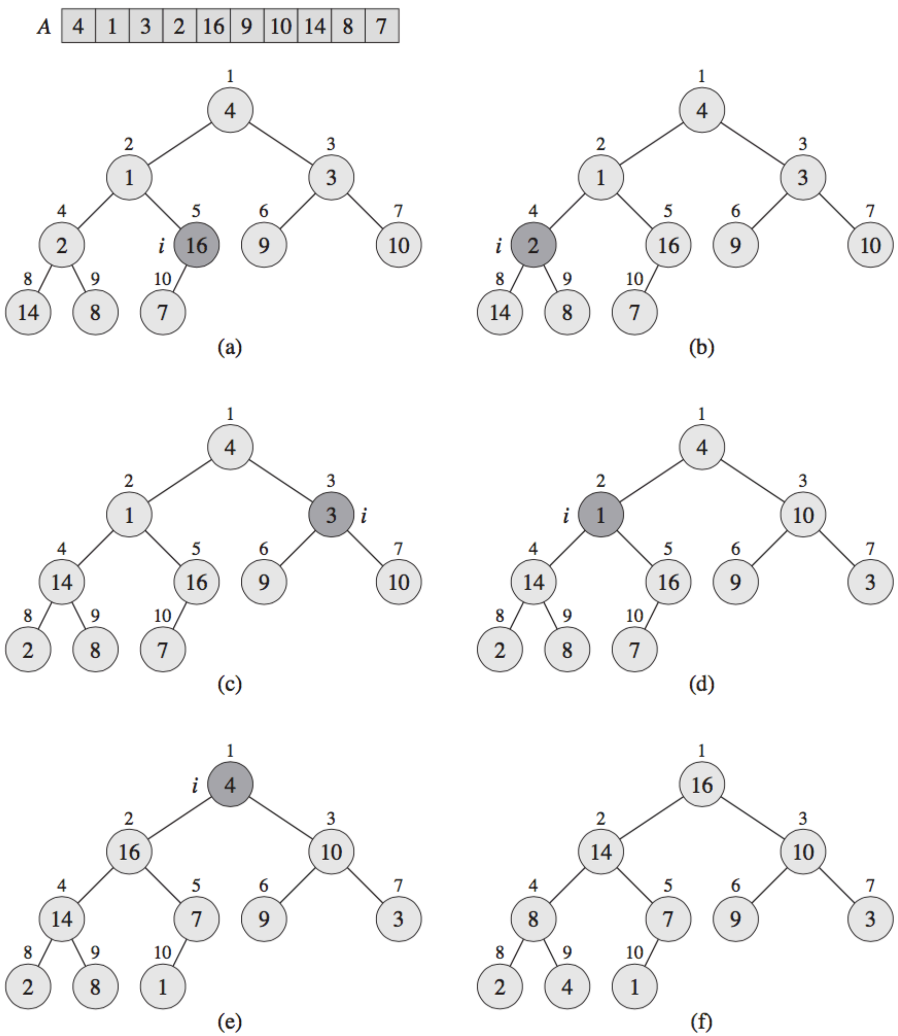
Building a heap

- We can use the procedure MAX-HEAPIFY in a bottom-up manner to convert an array A[1,..,n], where n = A.*length*, into a max-heap.

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

Visualizing BUILD-MAX-HEAP

The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A,i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.



Running time for BUILD-MAX-HEAP

- The time required by MAX-HEAPIFY when called on a node of height h is O(h), and so we can express the total cost of BUILD-MAX-HEAP as being bounded from above by:

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

- We evalaute the last summation by substituting x = 1/2 in the formula (**Integrating and differentiating series**) yielding

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

- Thus, we can bound the running time of BUILD-MAX-HEAP as

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

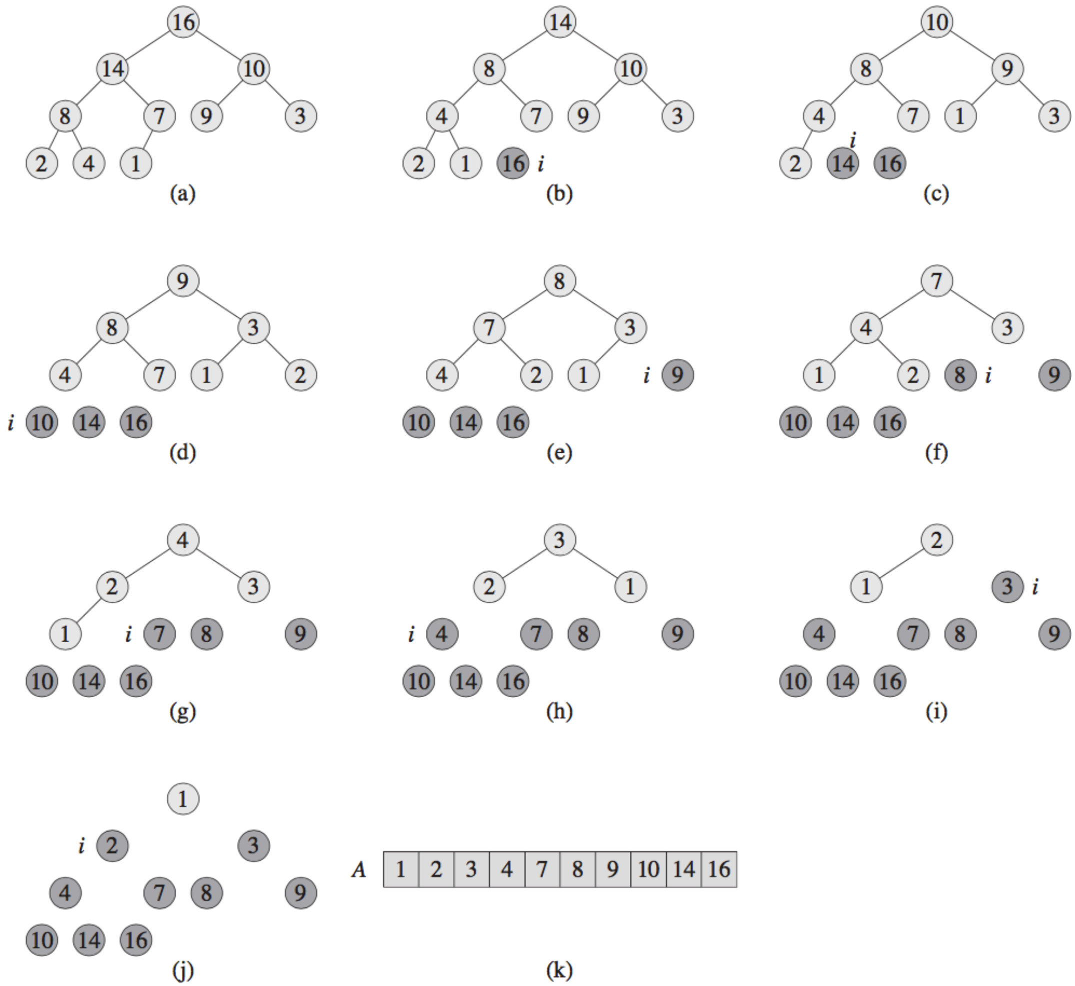
Heapsort Algorithm

- The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A[1,..,n], where n = A.*length*.
- Since the maximum element of the array is stored at the root A[1], we can put it into its correct final position by exchanging it with A[n].

```
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)
```

Visualizing HEAPSORT

The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX- HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.



Running time for Heapsort

- The HEAPSORT procedure takes time O(n lg n), since the call to BUILD-MAX- HEAP takes time O(n) and each of the n - 1 calls to MAX-HEAPIFY takes time O(lg n).

Priority Queue

- Heapsort is an excellent algorithm, but a good implementation of quicksort, usually beats it in practice.
- Nevertheless, the heap has its speciality:
  - most popular applications of a heap: as an efficient priority queue.
- Priority queues come in two forms:
  - max-priority queues.
  - min-priority queues.
- A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***.

Max-priority and min-priority Queue

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

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

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) 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.

- Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY.

Max-priority queue – HEAP-MAXIMUM

- The procedure HEAP-MAXIMUM implements the MAXIMUM operation in O(1) time.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

Running time of HEAP-MAXIMUM

- The procedure HEAP-MAXIMUM implements the MAXIMUM operation in O(1) time.

Max-priority queue - HEAP-EXTRACT-MAX

- The procedure HEAP-EXTRACT-MAX implements the EXTRACT-MAX operation.
- It is similar to the **for** loop body (lines 3–5) of the HEAPSORT procedure.

```
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
```

Running time of HEAP-EXTRACT-MAX

- The running time of HEAP-EXTRACT-MAX is O(lg n), since it performs only a constant amount of work on top of the O(lg n) time for MAX-HEAPIFY.

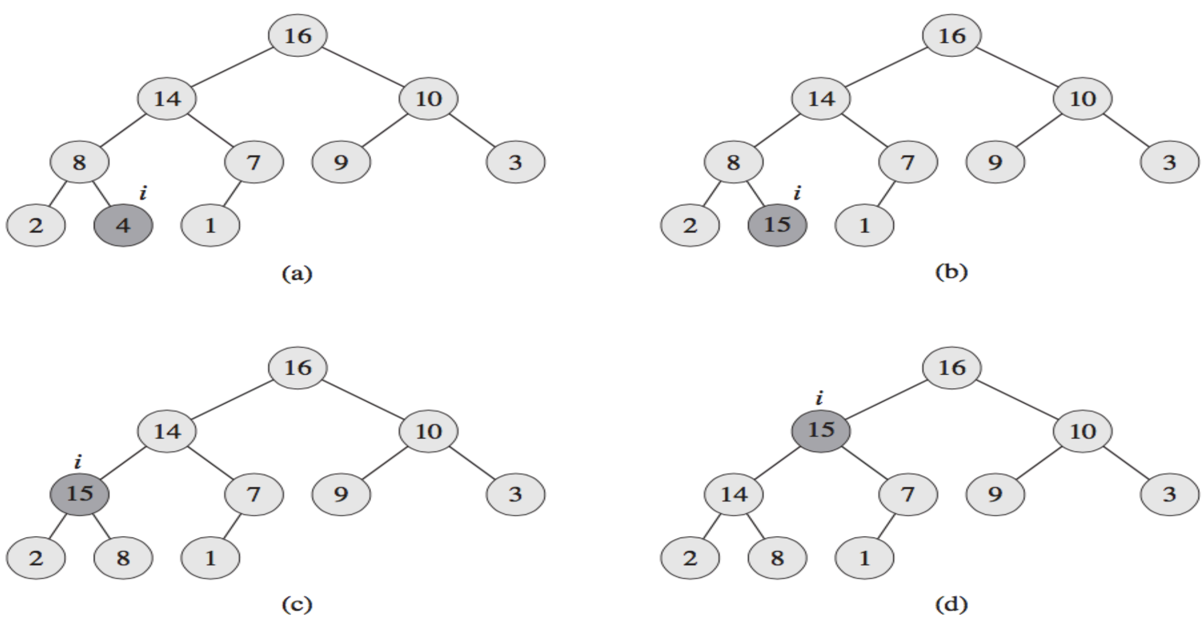
Max-priority queue - HEAP-INCREASE-KEY

- The procedure HEAP-INCREASE-KEY implements the INCREASE-KEY operation.

```
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)
```

Visualizing HEAP-INCREASE-KEY

The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of (a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, A[PARENT(i)]≥ A[i]. The max-heap property now holds and the procedure terminates.



Running time for HEAP-INCREASE-KEY

- The running time of HEAP-INCREASE-KEY on an n-element heap is O(lg n), since the path traced from the node updated in line 3 to the root has length O(lg n).

Max-priority queue - MAX-HEAP-INSERT

- The procedure MAX-HEAP-INSERT implements the INSERT operation.

MAX-HEAP-INSERT(A, key)

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

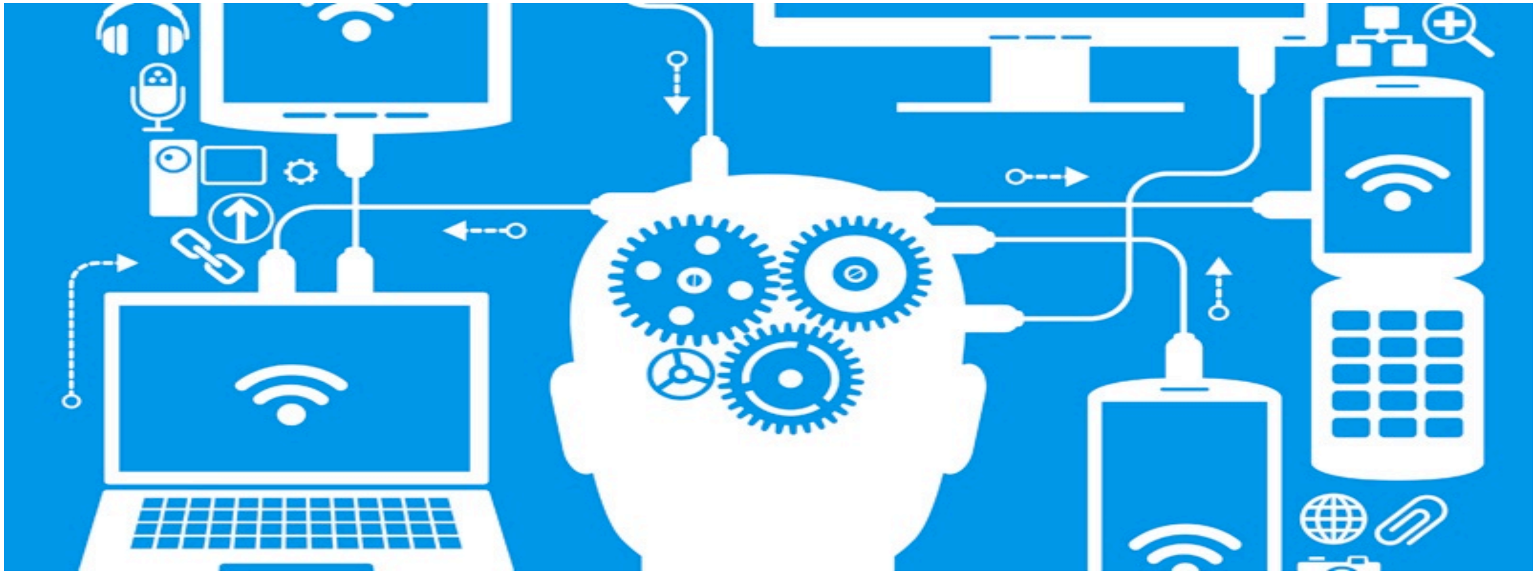
Running time for MAX-HEAP-INSERT

- The running time of MAX-HEAP-INSERT on an n-element heap is $O(\lg n)$.

Summary

- In summary, a heap can support any priority-queue operation on a set of size n in $O(\lg n)$ time.

In the next lecture..



Lecture 7: Data Structure I – Hash Table