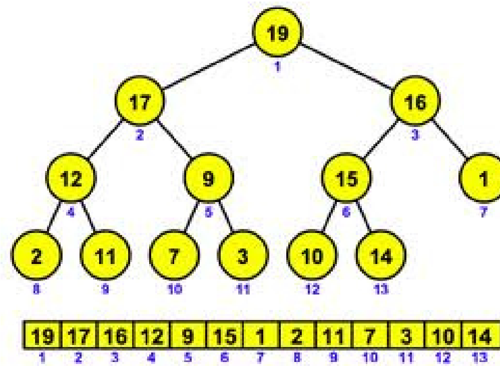


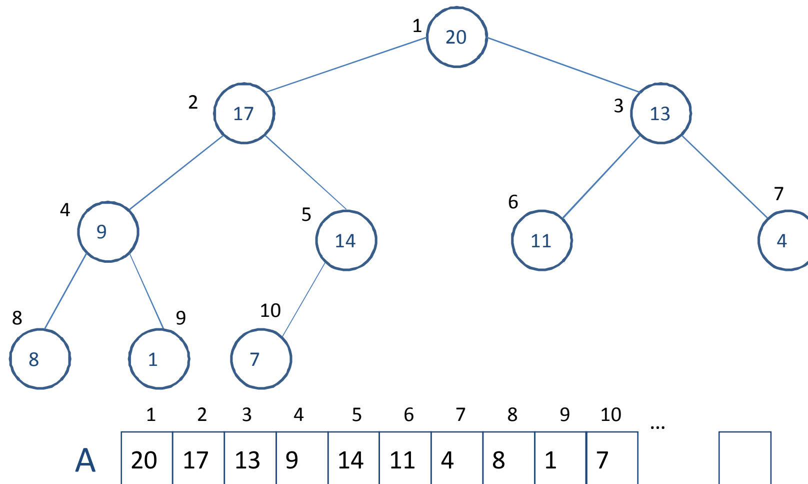
Heaps and Heapsort

Atul Gupta

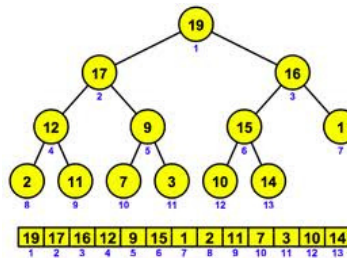
Heap

- A heap
  - is a complete binary tree, i.e. completely filled except possibly last level which is filled from left to right
  - satisfy heap property (max-heap or min-heap)
- Can be implemented in an array without pointers
- Efficient data structure to implement Priority Queues

Example: A Max Heap



Heap



- Can be a Min-heap or Max-heap
- A Heap is represented in an array by
  - *length [A]* – Size of the array A
  - *heapsize [A]* – size of the heap stored in the array
- The root of the heap is *A[1]*
- For an index node i
  - *parent (i)* – return ⌊ i/2 ⌋
  - *left (i)* – return (2*i)
  - *right (i)* – return (2*i + 1)
- Max-heap property – value of a node is greater than or equal to both of its children
  - i.e. A [Parent (i)] >= A [i]

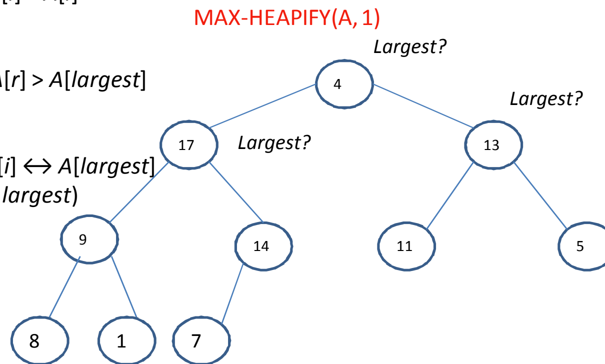
Basic Heap Operations

1. MAX-HEAPIFY procedure
2. BUILD-MAX-HEAP procedure
3. HEAPSORT procedure
4. MAX-HEAP-INSERT procedure
5. HEAP-EXTRACT-MAX procedure

1. Maintaining Heap Property

MAX-HEAPIFY(*A*, *i*)
1 *l* ← left (*i*)
2 *r* ← right (*i*)
3 **if** *l* ≤ *heapsize* [*A*] and *A*[*l*] > *A*[*i*]
4 **then** *largest* ← *l*
5 **else** *largest* ← *i*
6 **if** *r* ≤ *heapsize* [*A*] and *A*[*r*] > *A*[*largest*]
7 **then** *largest* ← *r*
8 **if** *largest* ≠ *i*
9 **then** exchange *A*[*i*] ↔ *A*[*largest*]
10 MAX-HEAPIFY(*A*, *largest*)

MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A(i) might be smaller than its children, thus violating the max-heap property.

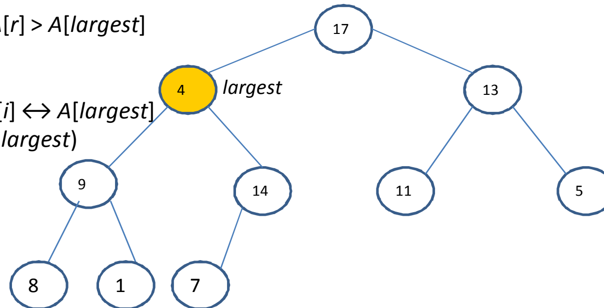


1. Maintaining Heap Property

MAX-HEAPIFY(*A*, *i*)

1 $l \leftarrow$ left (i)
2 $r \leftarrow$ right (i)
3 **if** $l \leq heapsize[A]$ and $A[l] > A[i]$
4 **then** $largest \leftarrow l$
5 **else** $largest \leftarrow i$
6 **if** $r \leq heapsize[A]$ and $A[r] > A[largest]$
7 **then** $largest \leftarrow r$
8 **if** $largest \neq i$
9 **then** exchange $A[i] \leftrightarrow A[largest]$
10 MAX-HEAPIFY(*A*, *largest*)

MAX-HEAPIFY(A, 1)

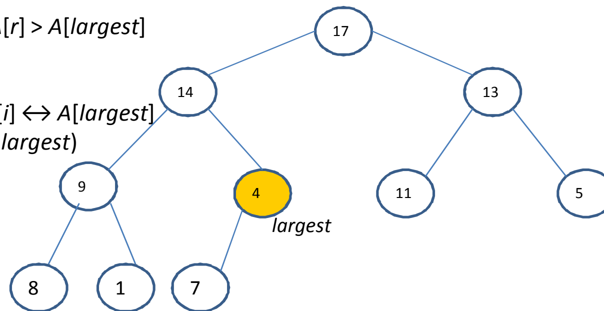


1. Maintaining Heap Property

MAX-HEAPIFY(*A*, *i*)

1 $l \leftarrow$ left (i)
2 $r \leftarrow$ right (i)
3 **if** $l \leq heapsize[A]$ and $A[l] > A[i]$
4 **then** $largest \leftarrow l$
5 **else** $largest \leftarrow i$
6 **if** $r \leq heapsize[A]$ and $A[r] > A[largest]$
7 **then** $largest \leftarrow r$
8 **if** $largest \neq i$
9 **then** exchange $A[i] \leftrightarrow A[largest]$
10 MAX-HEAPIFY(*A*, *largest*)

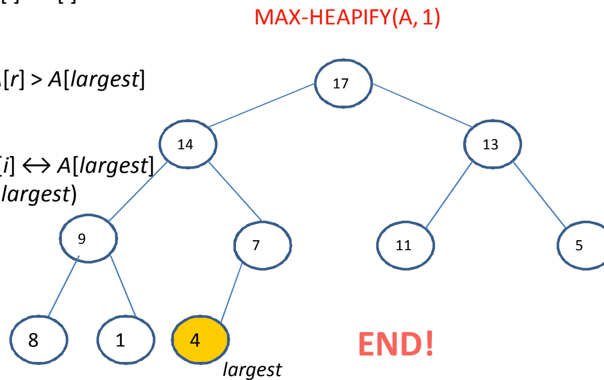
MAX-HEAPIFY(A, 1)



1. Maintaining Heap Property

MAX-HEAPIFY(A, i)
1 $l \leftarrow$ left (i)
2 $r \leftarrow$ right (i)
3 **if** $l \leq heapsize$ [A] and $A[l] > A[i]$
4 **then** $largest \leftarrow l$
5 **else** $largest \leftarrow i$
6 **if** $r \leq heapsize$ [A] and $A[r] > A[largest]$
7 **then** $largest \leftarrow r$
8 **if** $largest \neq i$
9 **then** exchange $A[i] \leftrightarrow A[largest]$
10 MAX-HEAPIFY(A, $largest$)

MAX-HEAPIFY(A, 1)

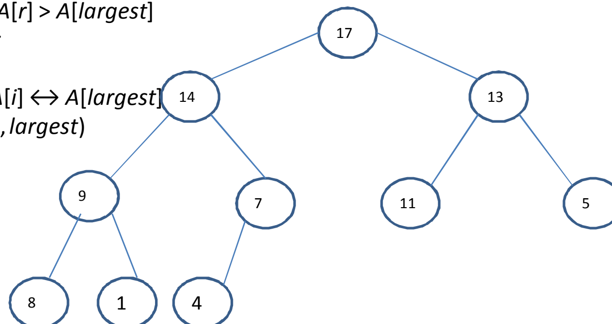


1. Maintaining Heap Property

MAX-HEAPIFY(A, i)
1 $l \leftarrow$ left (i)
2 $r \leftarrow$ right (i)
3 **if** $l \leq heapsize$ [A] and $A[l] > A[i]$
4 **then** $largest \leftarrow l$
5 **else** $largest \leftarrow i$
6 **if** $r \leq heapsize$ [A] and $A[r] > A[largest]$
7 **then** $largest \leftarrow r$
8 **if** $largest \neq i$
9 **then** exchange $A[i] \leftrightarrow A[largest]$
10 MAX-HEAPIFY(A, $largest$)

Time Complexity ?

= O (log n)



Building a Heap

BUILD-MAX-HEAP(*A*)

1 *heapsize*[*A*] ← *length*[*A*] // *n* = *length*[*A*]

2 for *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**

3   MAX-HEAPIFY(*A*, *i*)

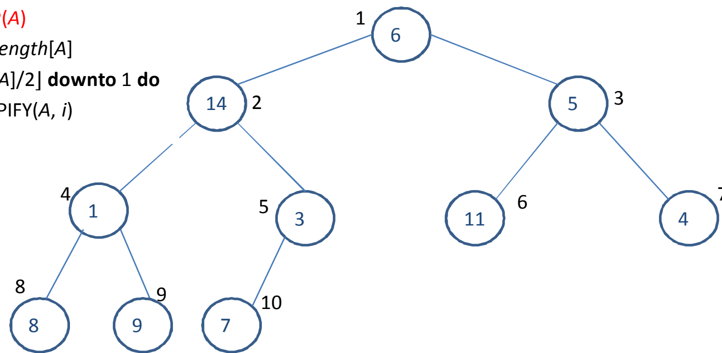
Building a Heap: An Example

BUILD-MAX-HEAP(*A*)

1 *heapsize*[*A*] ← *length*[*A*]

2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**

3 MAX-HEAPIFY(*A*, *i*)

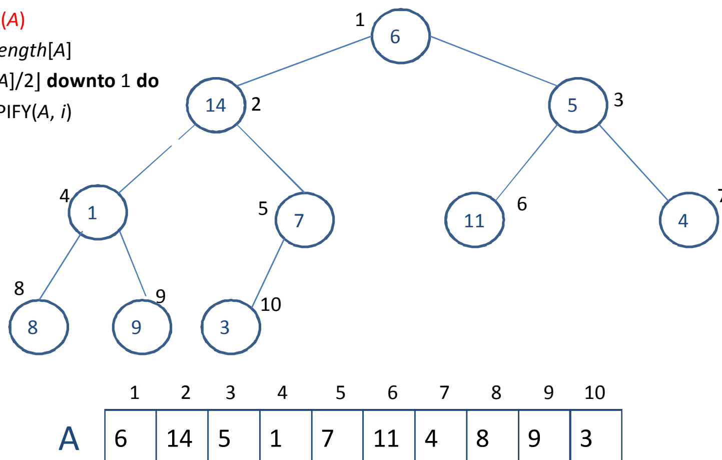


A

1	2	3	4	5	6	7	8	9	10
6	14	5	1	3	11	4	8	9	7

Building a Heap: An Example

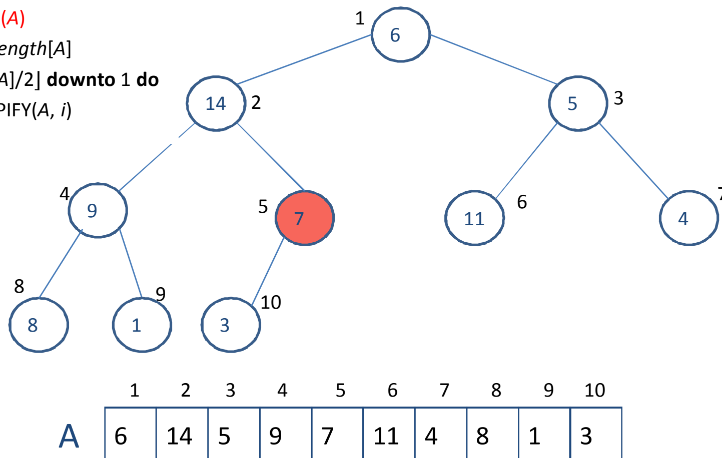
BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)



	1	2	3	4	5	6	7	8	9	10
A	6	14	5	1	7	11	4	8	9	3

Building a Heap: An Example

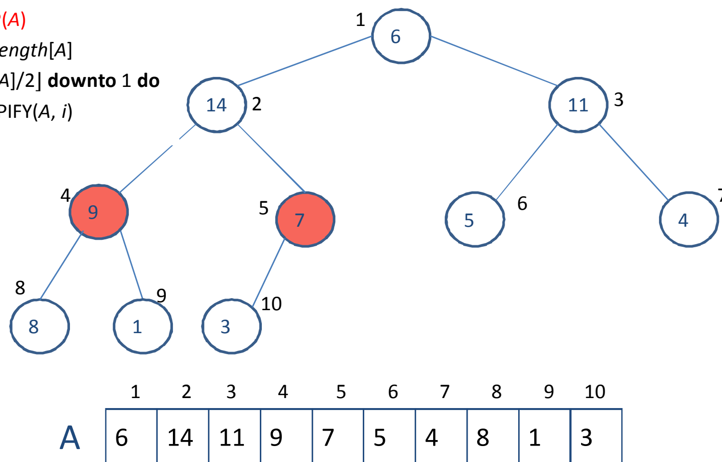
BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)



	1	2	3	4	5	6	7	8	9	10
A	6	14	5	9	7	11	4	8	1	3

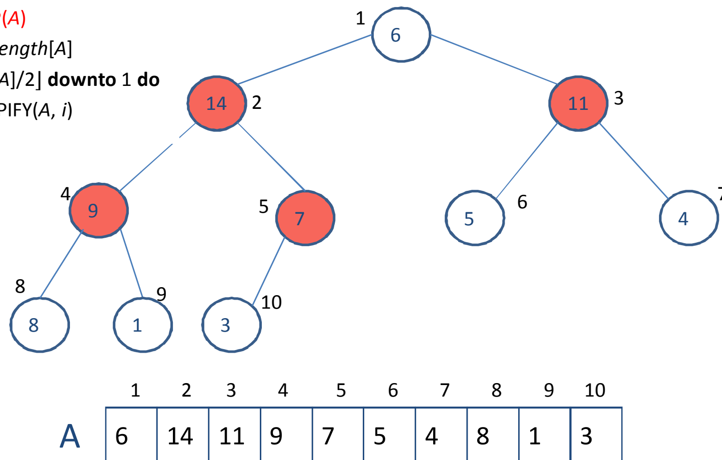
Building a Heap: An Example

BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)



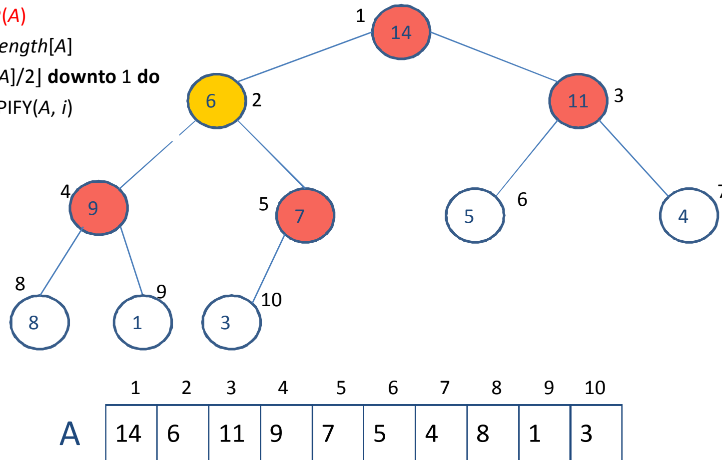
Building a Heap: An Example

BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)



Building a Heap: An Example

BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)

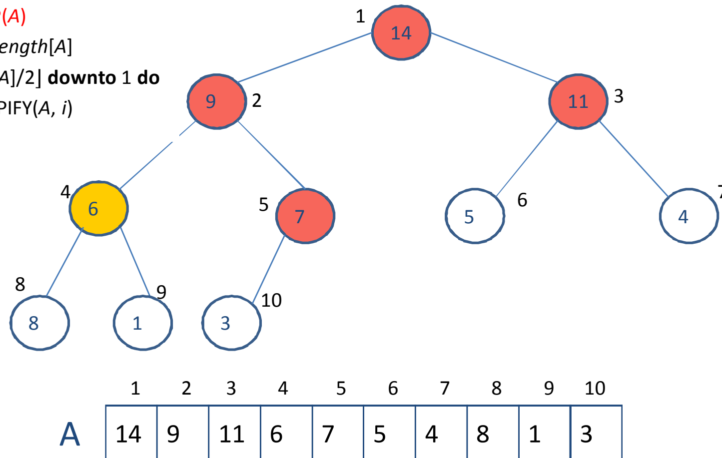


A

1	2	3	4	5	6	7	8	9	10
14	6	11	9	7	5	4	8	1	3

Building a Heap: An Example

BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)

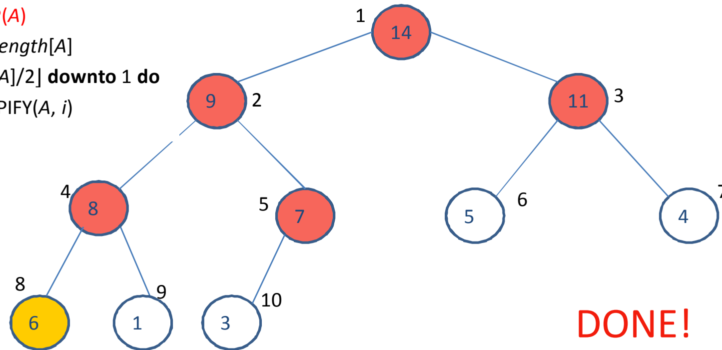


A

1	2	3	4	5	6	7	8	9	10
14	9	11	6	7	5	4	8	1	3

Building a Heap: An Example

BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)



A

1	2	3	4	5	6	7	8	9	10
14	9	11	8	7	5	4	6	1	3

Building a Heap

BUILD-MAX-HEAP(*A*)
1 *heapsize*[*A*] ← *length*[*A*]
2 **for** *i* ← ⌊*length*[*A*]/2⌋ **downto** 1 **do**
3 MAX-HEAPIFY(*A*, *i*)

Time Complexity ?

= O (n log n)?

= O (n)

Heapsort

HEAPSORT(*A*) Time Complexity?

1 BUILD-MAX-HEAP(*A*) // O (n)

2 **for** *i* ← *length*[*A*] **downto** 2 **do** // O (n)

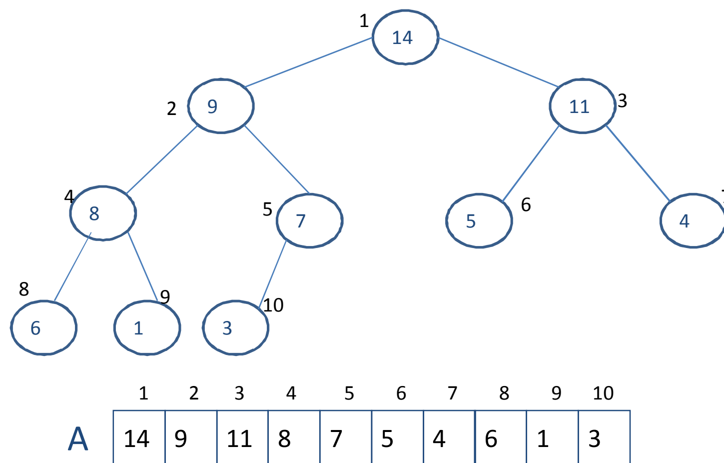
3 exchange *A*[1] ↔ *A*[*i*]

4 *heapsize*[*A*] ← *heapsize*[*A*] - 1

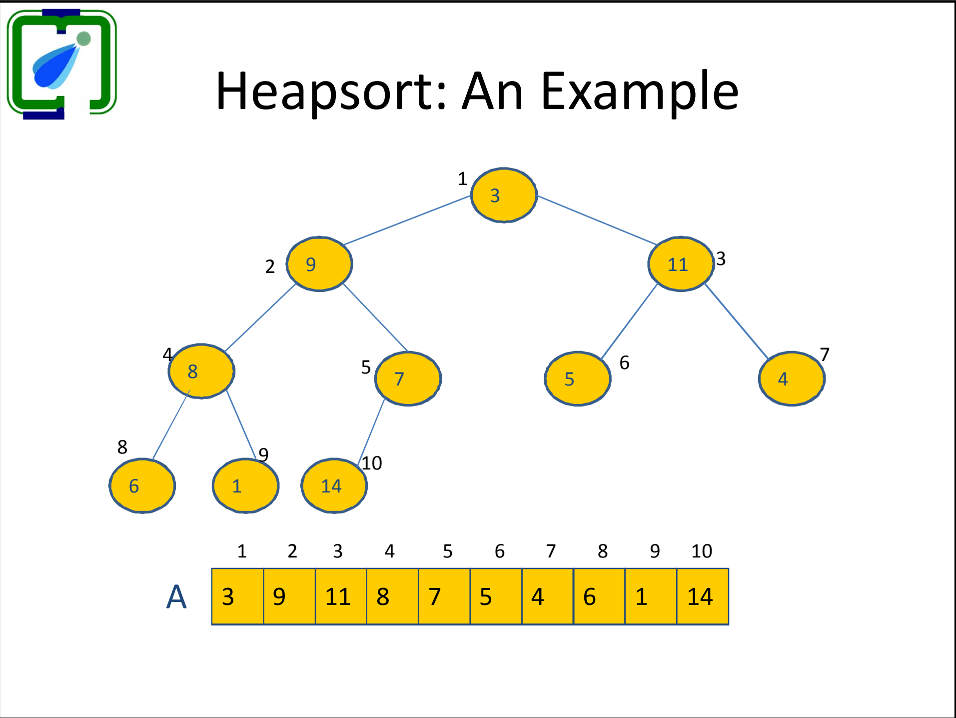
5 MAX-HEAPIFY(*A*, 1) // O (n log n)

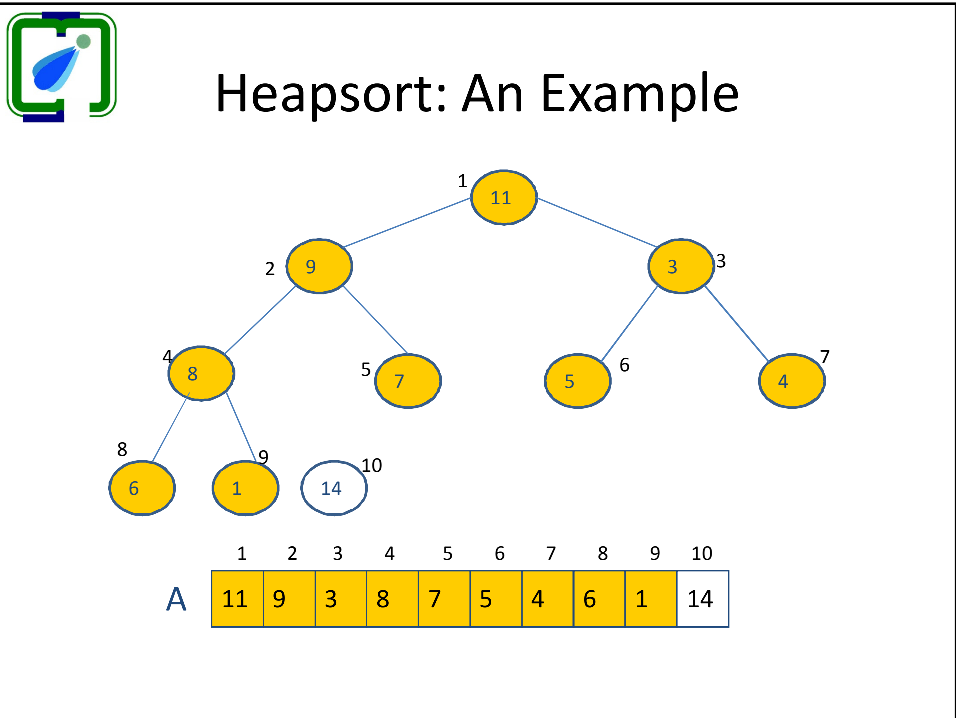
= O (n log n)

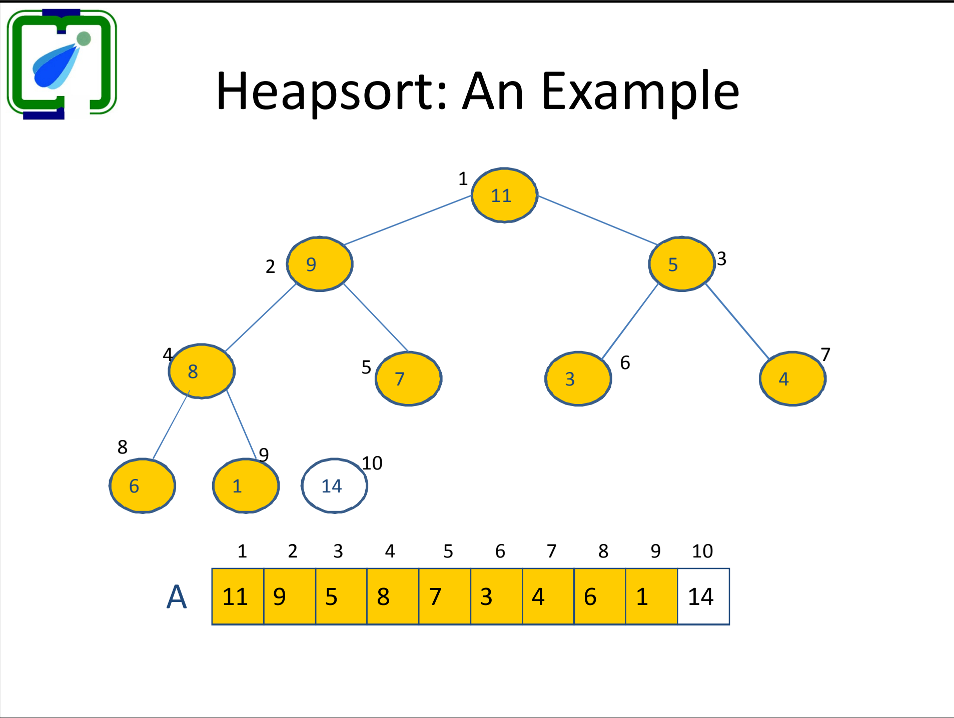
Heapsort: An Example

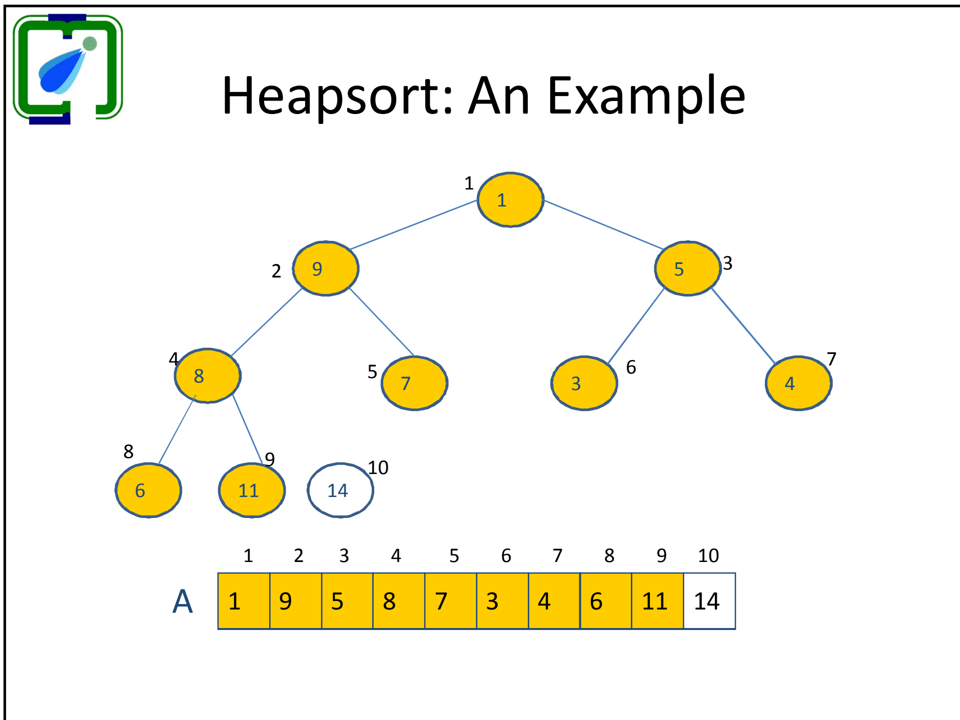


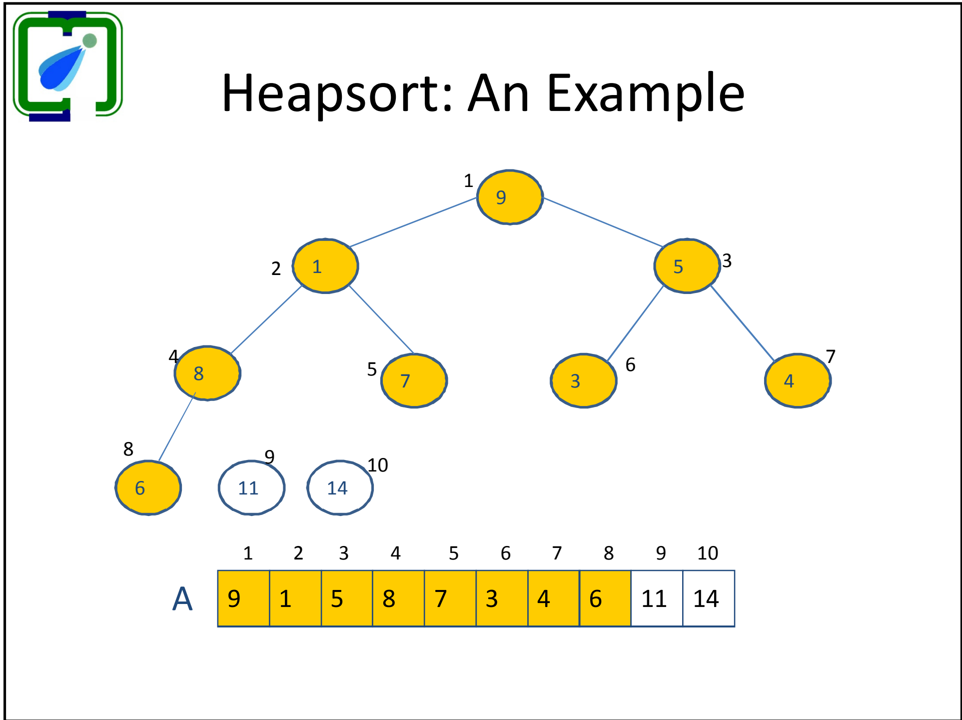
	1	2	3	4	5	6	7	8	9	10
A	14	9	11	8	7	5	4	6	1	3

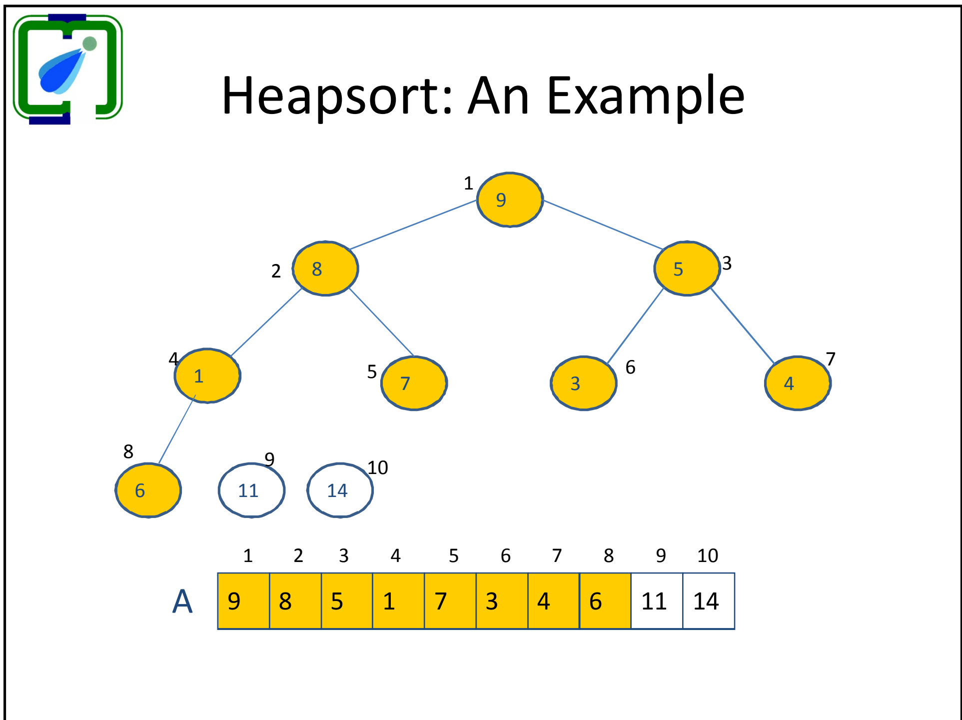

Heapsort: An Example
1
3
2
9
11
3
4
8
5
7
5
6
4
7
8
6
1
9
14
10
1 2 3 4 5 6 7 8 9 10
A
3 9 11 8 7 5 4 6 1 14

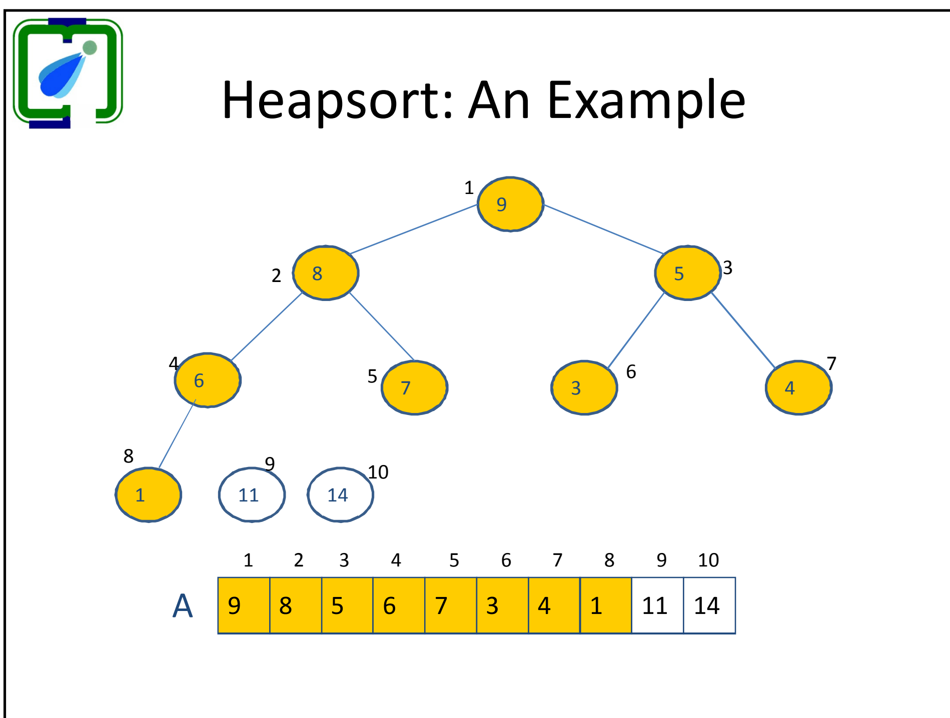

Heapsort: An Example
1
11
2
9
3
3
4
8
5
7
5
6
4
7
8
6
1
9
14
10
1 2 3 4 5 6 7 8 9 10
A
11 9 3 8 7 5 4 6 1 14

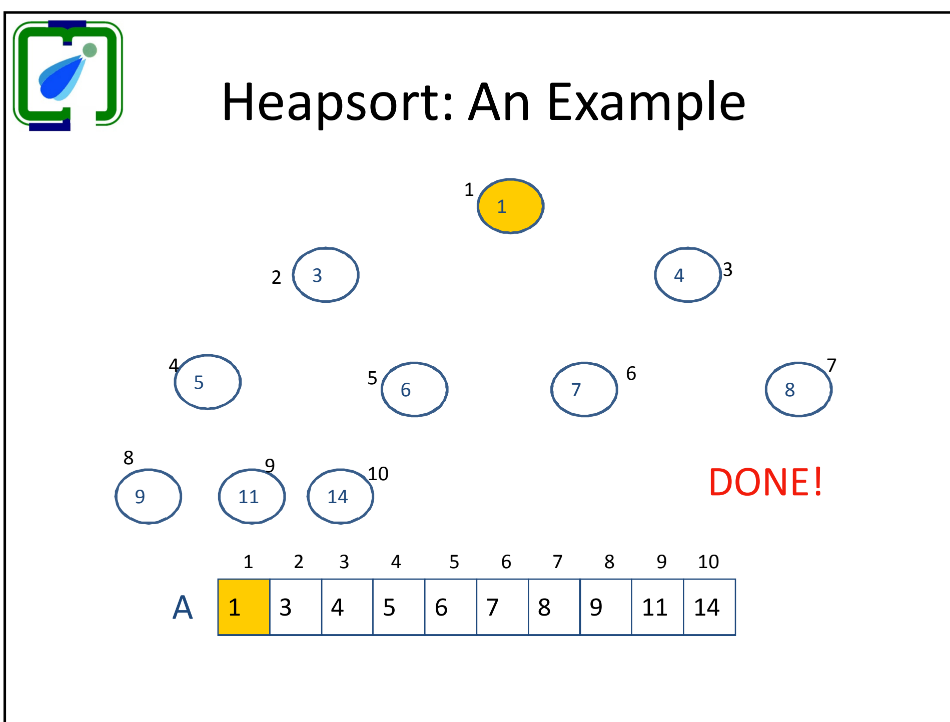

Heapsort: An Example
1
11
2
9
5
3
4
8
5
7
3
6
4
7
8
6
1
9
14
10
1 2 3 4 5 6 7 8 9 10
A
11 9 5 8 7 3 4 6 1 14


Heapsort: An Example
1
1
2
9
5
3
4
8
5
7
3
6
4
7
8
6
11
9
14
10
1 2 3 4 5 6 7 8 9 10
A
1 9 5 8 7 3 4 6 11 14


Heapsort: An Example
1
9
2
1
5
3
4
8
5
7
3
6
4
7
8
6
9
11
14
10
1 2 3 4 5 6 7 8 9 10
A
9 1 5 8 7 3 4 6 11 14


Heapsort: An Example
1
9
2
8
5
3
4
1
5
7
3
6
4
7
8
6
9
11
14
10
1 2 3 4 5 6 7 8 9 10
A
9 8 5 1 7 3 4 6 11 14


Heapsort: An Example
1
9
2
8
5
3
4
6
5
7
3
6
4
7
8
1
9
11
14
10
1 2 3 4 5 6 7 8 9 10
A
9 8 5 6 7 3 4 1 11 14


Heapsort: An Example
1
1
2
3
4
3
4
5
5
6
7
6
8
7
8
9
9
11
14
10
DONE!
1 2 3 4 5 6 7 8 9 10
A
1 3 4 5 6 7 8 9 11 14

Priority Queue

- A Heap is an efficient data structure to implement a priority queue
- Operations (A is the Max-Heap implemented in an array)
  - INSERT (A, x)
  - GET-MAX(A)
  - REMOVE-MAX(A)

Operations: Priority Queue

GET-MAX(A)
    return A[1]

INSERT(A, x)
1 **if** *heapsize*[*A*] < 1 **then**
2   *A*[1] ← *x*
3 **else**
4   *heapsize*[*A*] ← *heapsize*[*A*] + 1
5   A[heapsize] ← x
6   exchange (A[1] ↔ A[heapsize])
7   MAX-HEAPIFY(*A*, 1)

REMOVE-MAX(*A*)
1 **if** *heapsize*[*A*] < 1 **then**
2   **error** "heap underflow"
3 *max* ← *A*[1]
4 *A*[1] ← *A*[*heapsize*[*A*]]
5 *heapsize*[*A*] ← *heapsize*[*A*] - 1
6 MAX-HEAPIFY(*A*, 1)
7 **return** *max*

Correct Operation: INSERT

```
INSERT(A, x)
1  if heapsize[A] < 1 then
2       A[1] ← x
3  else
4     heapsize[A] ← heapsize[A] + 1
5     A[heapsize] ← x
6     i ← heapsixe
7     while (A[i] > parent(A[i])
          swap(A[i], A[parent(A[i])])
          i ← parent(A[i])
```

Applications: Heap

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios
- Selection algorithms: Finding the min, max, both the min and max, median, or even the *k*-th largest element can be done in linear time (often constant time) using heaps
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

References

- Introduction to Algorithms (2nd/3rd Edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein, PHI Press