Lecture 5

COMP 3760

Transform and Conquer algorithms

Text sections 6.1, 6.4

Transform and Conquer

- This technique solves a problem by a transformation to:
  - a more convenient instance of the same problem (aka **instance simplification**)
  - a different representation of the same instance (aka **representation change**)

Transform and Conquer examples

- **Instance simplification (pre-sorting)**
  - Checking element uniqueness in an array
  - Computing the mode
  - Searching (again)

- **Representation change**
  - Heap
    - Implementation
    - Insert and Delete
    - Construction
  - Heap sort

Instance simplification

- Transform a problem into a *more convenient* instance of the same problem

Find a key
in an array

Find a key in
a *sorted array*

Element uniqueness in an array

Example: Element uniqueness in an array

- Problem: Determine if all elements in an array are distinct
  - I.e.: "Are there any duplicated values?"

56	98	11	49	1	45	99	37	33	27	39	33	49

- Brute force algorithm
  - Compare all pairs of elements
  - Efficiency: O(n^2)

Example: Element uniqueness in an array

- Instance simplification (presorting) approach:
  - Part 1: sort by efficient sorting algorithm (e.g. mergesort)

1	11	27	33	33	37	39	45	49	49	56	98	99

  - Part 2: scan array to check pairs of adjacent elements

- Efficiency: O(nlogn) + O(n) = O(nlogn)

Example: Element uniqueness in an array

ALGORITHM *PresortElementUniqueness*$(A[0..n-1])$

//Solves the element uniqueness problem by sorting the array first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Returns "true" if A has no equal elements, "false" otherwise

sort the array A
for $i \leftarrow 0$ **to** $n-2$ **do**
    if $A[i] = A[i+1]$ **return false**
return true

Computing
a mode

Computing a mode

- The ***mode*** is the value that occurs most often in a given list of numbers.

5	1	6	7	6	5	7	6

Mode: 6

Computing a mode

- Brute Force:
  - Scan the list
  - Compute the frequencies of all distinct values
  - Find the value with the largest frequency

5	1	6	7	6	5	7	6

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5
Frequency	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1
Frequency	1	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6
Frequency	1	1	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequency	1	1	1	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequency	1	1	2	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequency	2	1	2	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequency	2	1	2	2

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

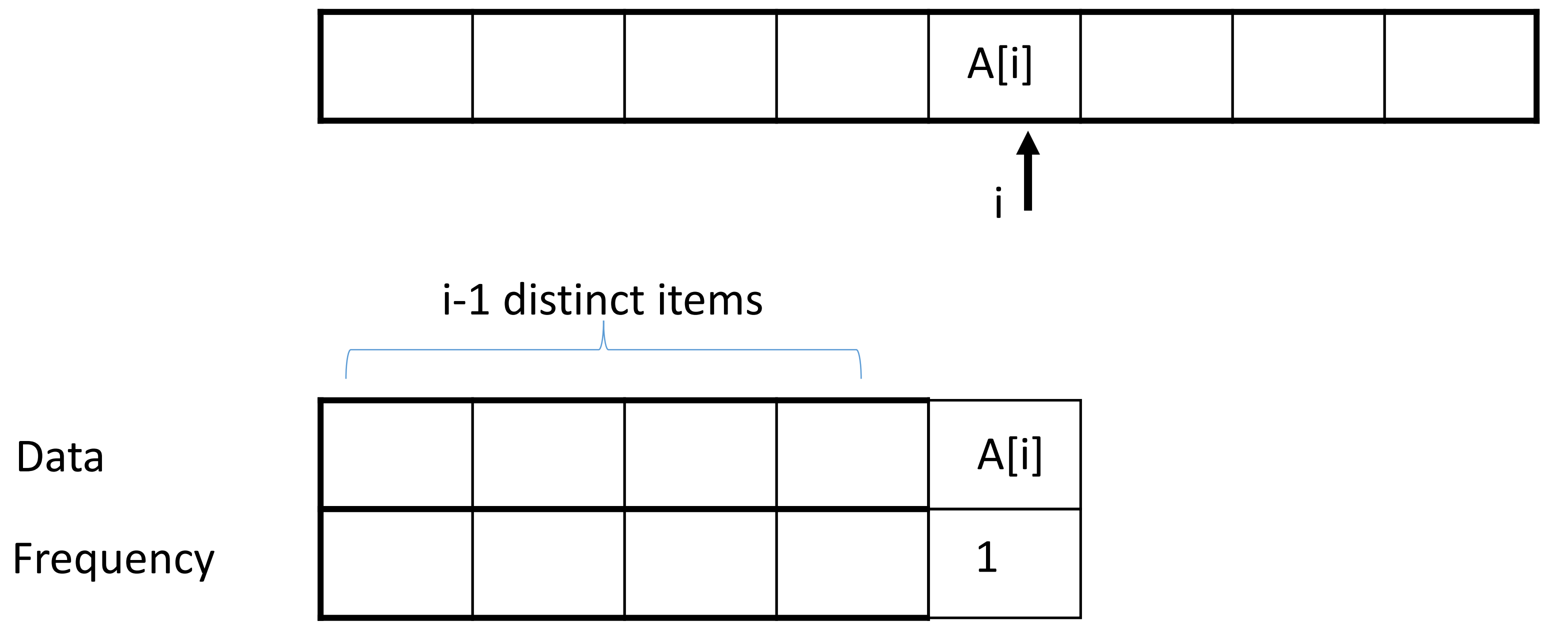
i

Data	5	1	6	7
Frequency	2	1	3	2

Max

Computing a mode

- Efficiency (worst-case):
  - List has no repeated elements
  - i^{th} element must be compared to $i - 1$ existing elements in the "Data" array



Computing a mode

- Efficiency (worst-case):

  - Creating auxiliary list ("Data" array): $0 + 1 + 2 + \cdots + (n-1)$ = O(n^2)
  - Finding max: O(n)

  - Efficiency (worst-case): O(n^2)

Computing a mode (pre-sorting)

- Part 1: Sort the input
  - All equal values will be adjacent to each other

1	5	5	6	6	6	7	7

- Part 2: Find the longest run of adjacent equal values in the sorted array

Computing a mode (pre-sorting)

ALGORITHM *PresortMode*($A[0..n-1]$)

//Computes the mode of an array by sorting it first
//Input: An array $A[0..n-1]$ of orderable elements
//Output: The array's mode

sort the array A
$i \leftarrow 0$ //current run begins at position i
$modefrequency \leftarrow 0$ //highest frequency seen so far
while $i \le n-1$ **do**
  $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
  while $i + runlength \le n-1$ **and** $A[i + runlength] = runvalue$
    $runlength \leftarrow runlength + 1$
  if $runlength > modefrequency$
    $modefrequency \leftarrow runlength$; $modevalue \leftarrow runvalue$
  $i \leftarrow i + runlength$
return $modevalue$

Computing a mode (pre-sorting)

- Efficiency:

$$T(n) = \text{Sorting time} + \text{Scanning time}$$

$$= O(n \log n) + O(n)$$

$$= O(n \log n)$$

Searching with presorting

Searching with presorting

- Problem: Search for a given key K in an array A[0..n-1]

- Presorting-based algorithm:
  - Part 1: Sort the array by an efficient sorting algorithm
  - Part 2: Apply binary search

- Efficiency: $O(n \log n) + O(\log n) = O(n \log n)$

- Good or bad? (Note that sequential search is $O(n)$)
- Why do we have our dictionaries, telephone directories, etc. sorted?

Linear vs Binary Search

- We know that Binary Search is better
  - O(N) vs. O(logN)
- But Binary Search requires a sorted list
  - Sorting is O(NlogN)
- Q: How does this help?

- A: If we have to search MANY times

Why is presorting better?

- What if we have A[1000] and search a million times?
- With Linear/Sequential Search:
  - Search is O(n) → 500 steps per search (average)
  - 1,000,000 searches → 500,000,000 steps
  - Total time: 500,000,000 steps
- With Presort + Binary Search
  - Presort = O(nlogn) = 1000*10 = 10,000 steps
  - Search is O(logn) → 10 steps per search (max)
  - 1,000,000 searches → 10,000,000 steps
  - Total time: 10,010,000 steps
- Presort+BinarySearch is about 50x better
  - Bigger input → even MOAR better!

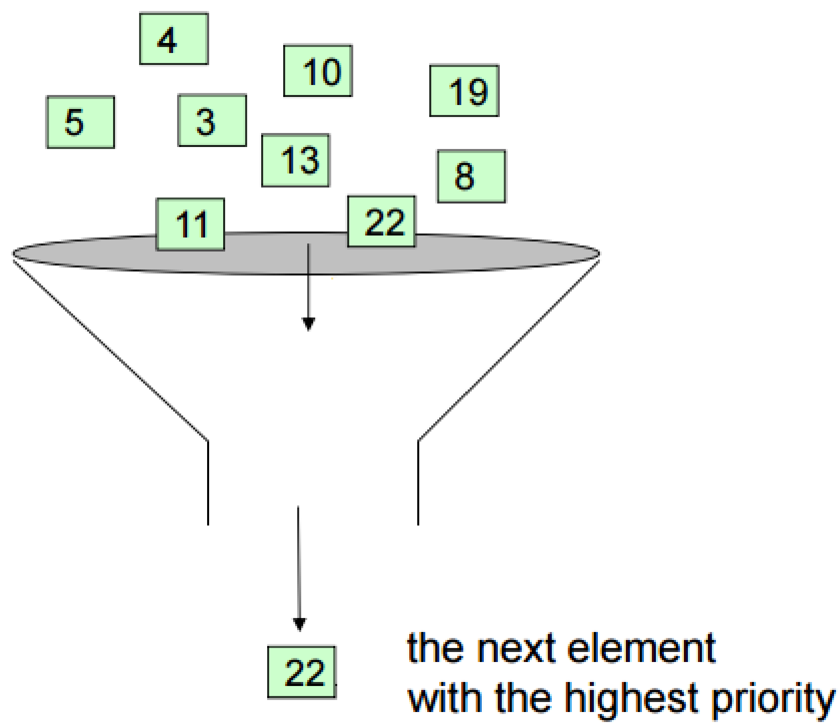
Transform and Conquer

- Previous examples:
  - Instance simplification
  - More specifically, pre-sorting
- Next example:
  - Representation change

Representation change: Heaps and Heapsort

Sample problem

- You're running a hospital ER
- Patients are coming in with different priority



We need two operations

- Insert()
  - Add a new person to the waiting room
  - Each person has a designated priority
- deleteMax()
  - Determine the person with the highest priority
  - Remove them from the waiting room

Simple implementations

- ArrayList
  - Insert: O(1)
  - deleteMax: O(n)

7	5	8	1	9

- SortedArrayList
  - Insert: O(logn + n) = O(n)
  - deleteMax: O(1)

1	5	7	8	9

Representation change

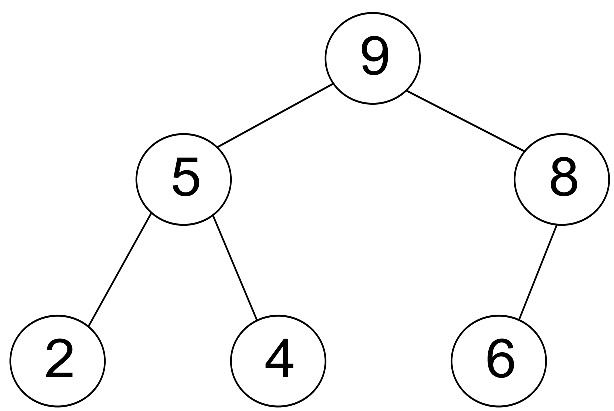
- Idea:
  - Given an array
  - Transform to a new data structure

    (Make a “**heap**” out of it)

- Efficiency of heap:
  - Insert an item: O(logn)
  - deleteMax: O(logn)

Heap definition

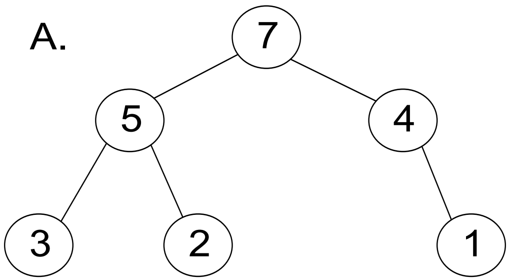
- Almost complete binary tree
  - filled on all levels, except last, where filled from left to right
- Every parent is greater than (or equal to) children



Heap or No Heap?

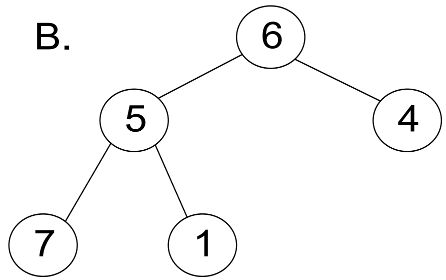
NO

A.



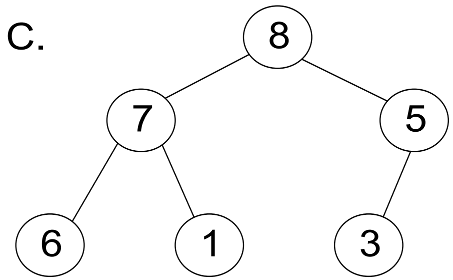
NO

B.



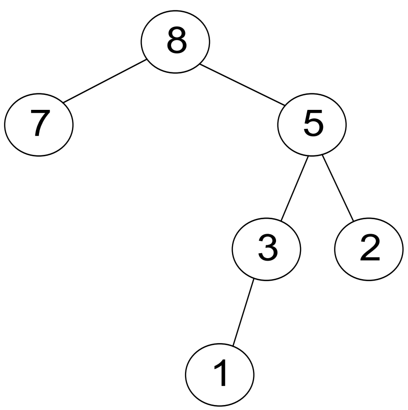
YES

C.



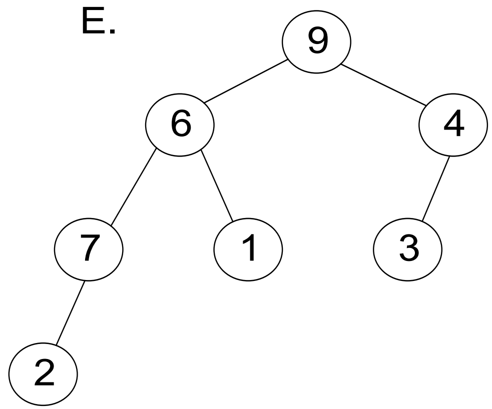
NO

D.



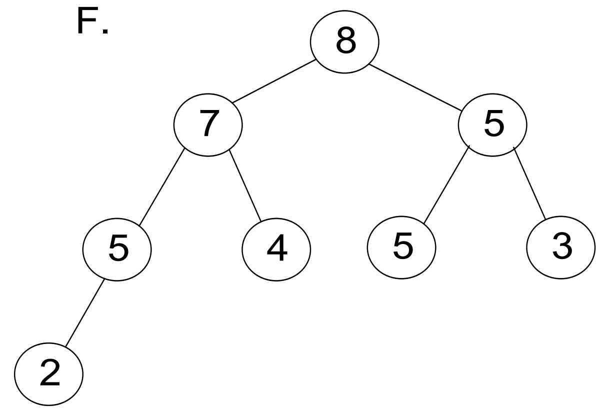
NO

E.



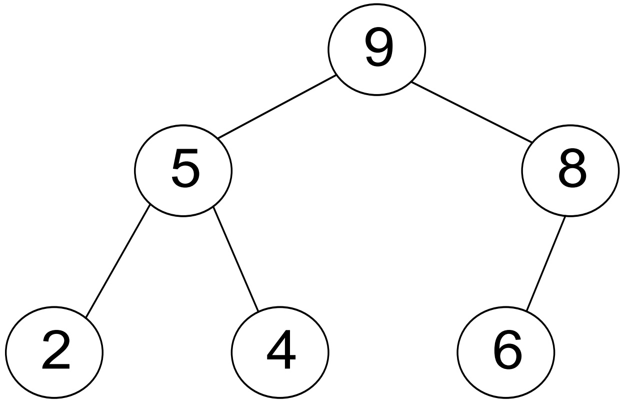
YES

F.



Heap characteristics

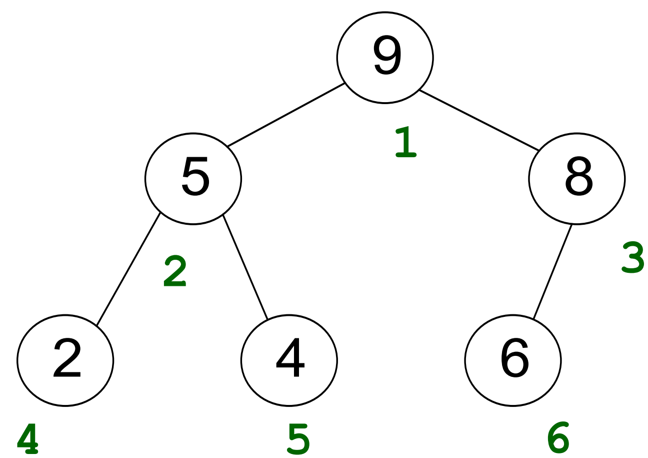
- Max value is in the root
- Heap with N elements has height = $\lfloor \log_2 N \rfloor$



N = 6
Height = 2

Heap implementation

- Use an array: no need for explicit parent or child pointers.
  - `Parent(i) = ⌊i/2⌋`
  - `Left(i)   = 2i`
  - `Right(i)  = 2i + 1`

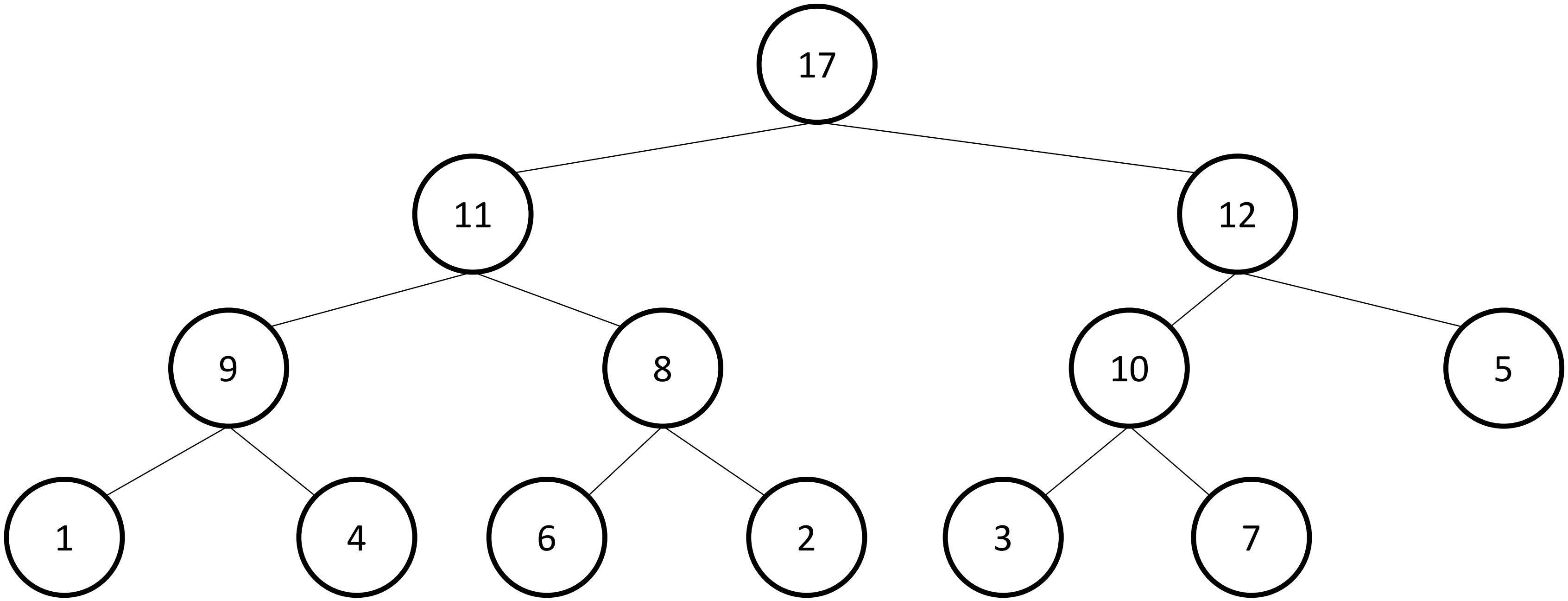


0	1	2	3	4	5	6
	9	5	8	2	4	6

Example 1

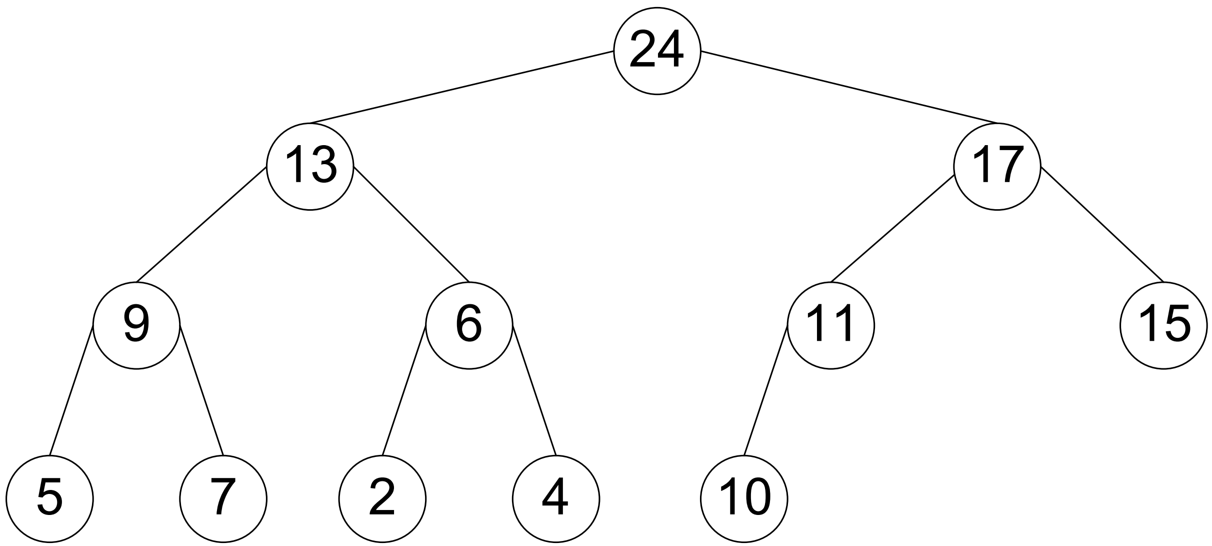
- Draw the tree representation of this heap

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
value	17	11	12	9	8	10	5	1	4	6	2	3	7



Example 2

- Draw the array representation of this heap



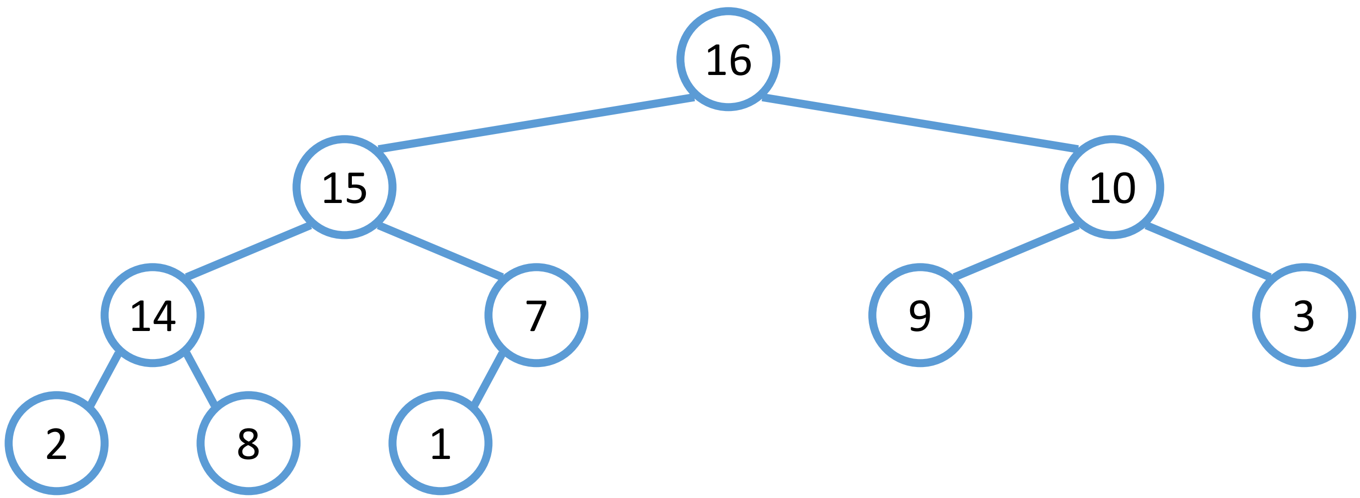
Index	1	2	3	4	5	6	7	8	9	10	11	12
value	24	13	17	9	6	11	15	5	7	2	4	10

Heap insertion

- Insert into next available slot
- Bubble up until it's heap ordered ("heapify")

Insert to heap example

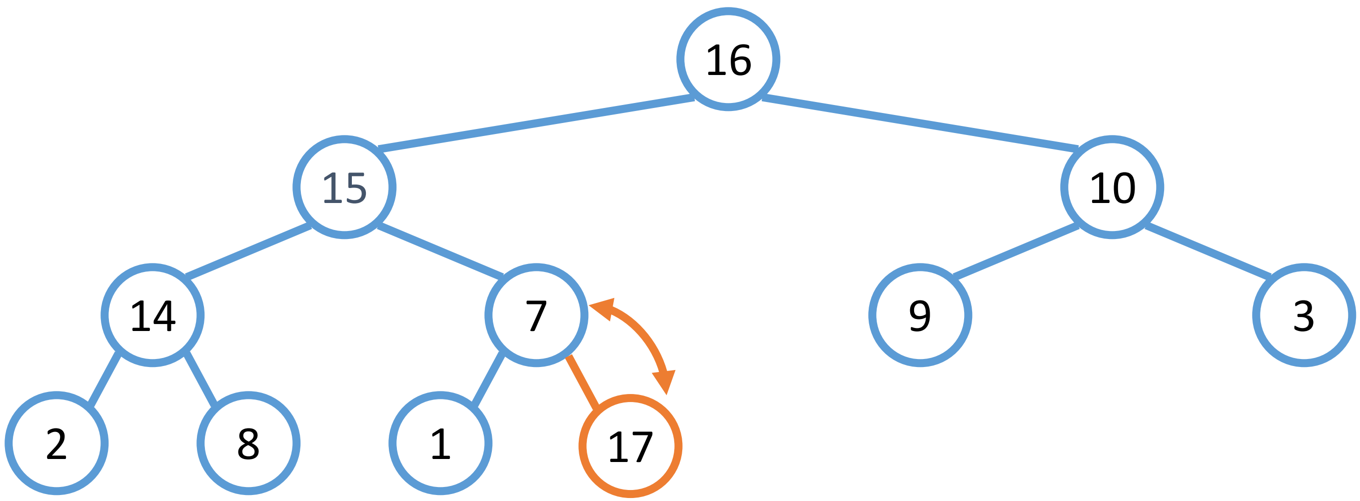
- Insert 17



A =

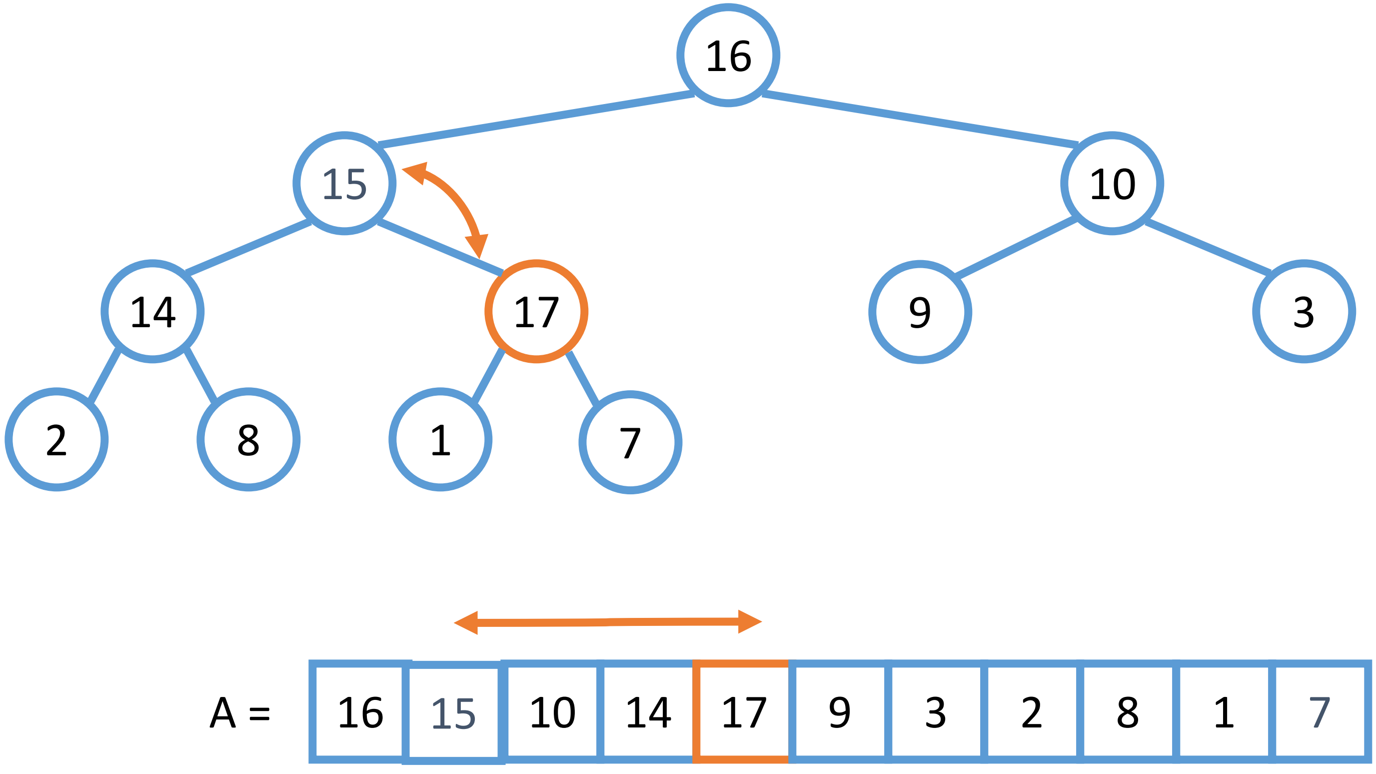
16	15	10	14	7	9	3	2	8	1

Insert to heap example

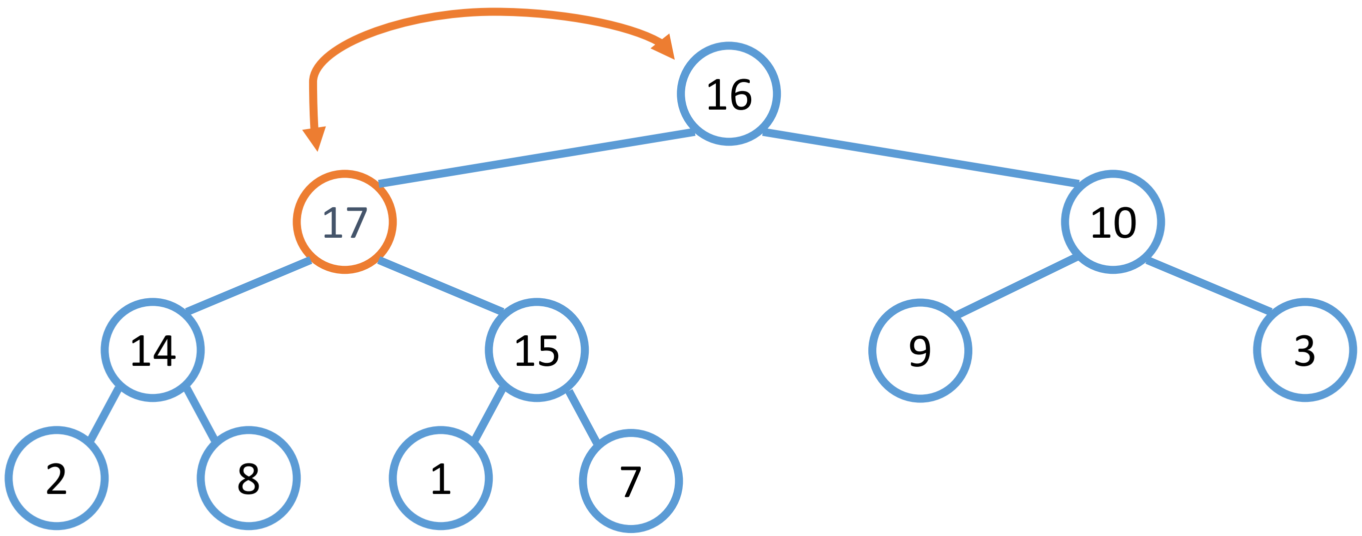


A =										
16	15	10	14	7	9	3	2	8	1	17

Insert to heap example



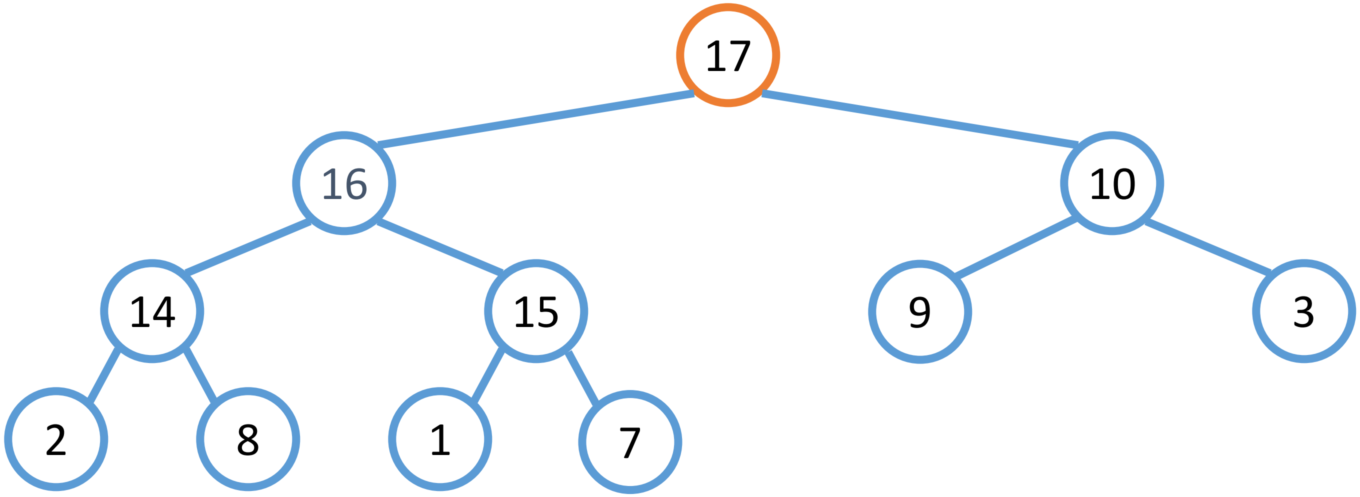
Insert to heap example



A =

16	17	10	14	15	9	3	2	8	1	7

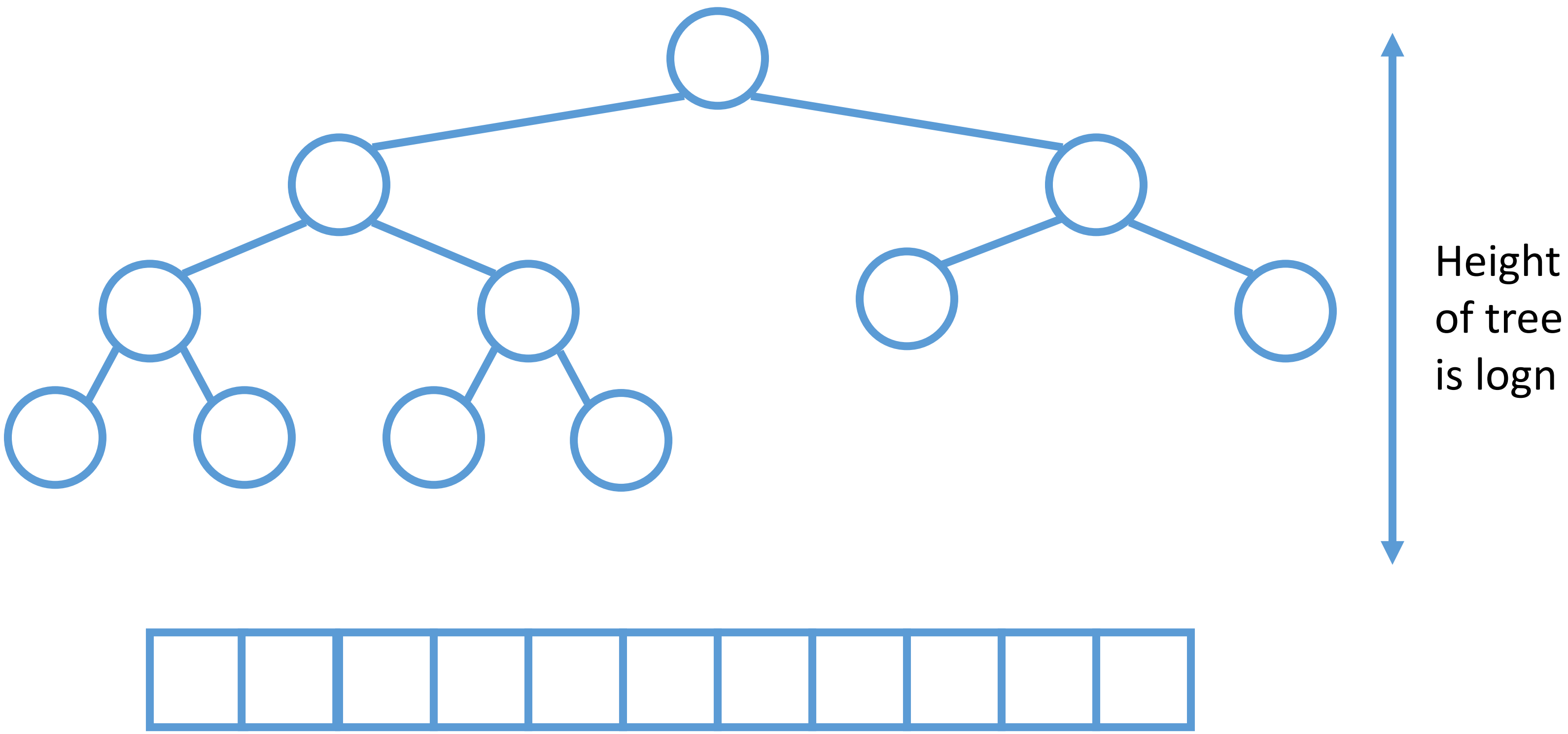
Insert to heap example



A =	17	16	10	14	15	9	3	2	8	1	7

Insert to heap example

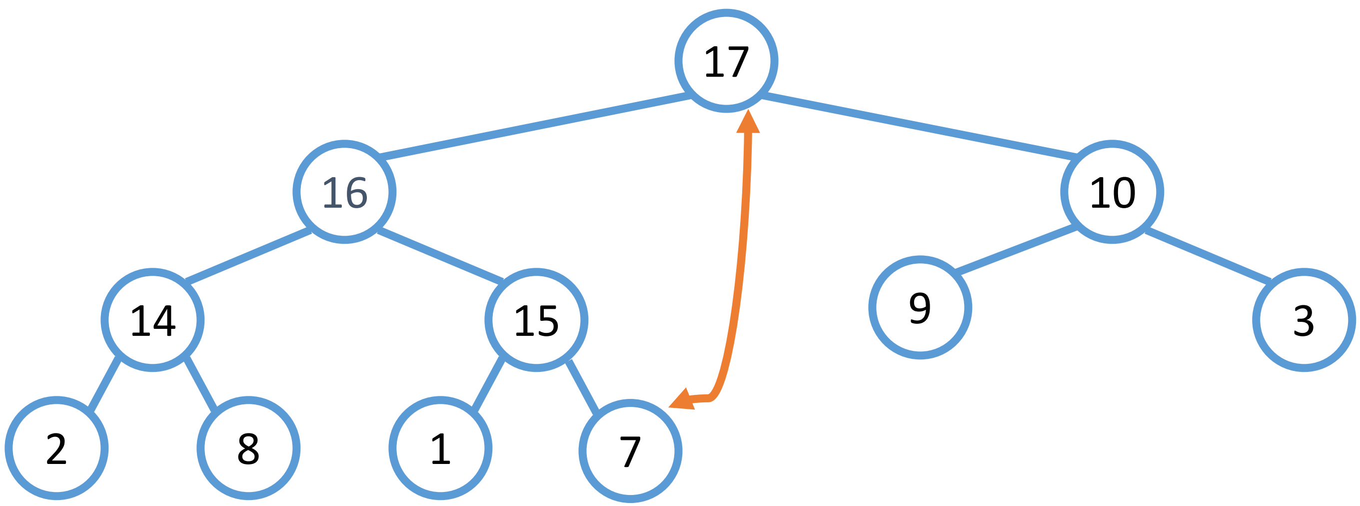
- Efficiency is O(log n)



Delete max from heap

- Exchange root with "last" leaf (bottom-most, right-most)
- Delete element
- Bubble root down until it's heap ordered

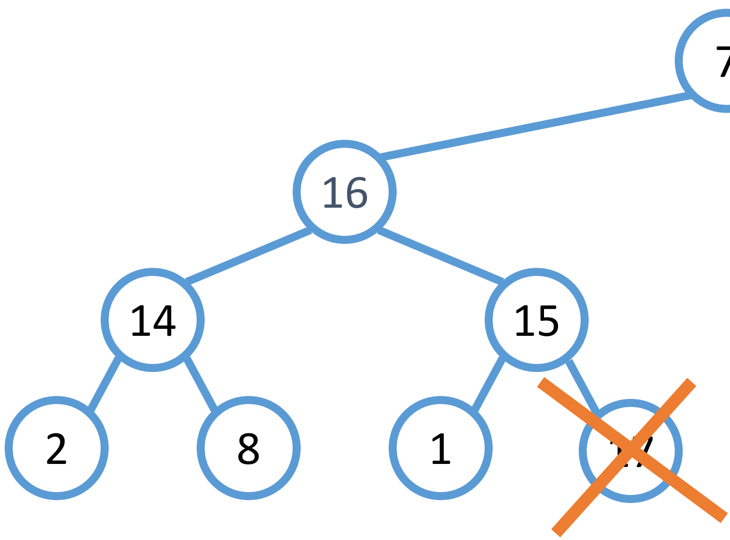
Delete max from heap Example



A =

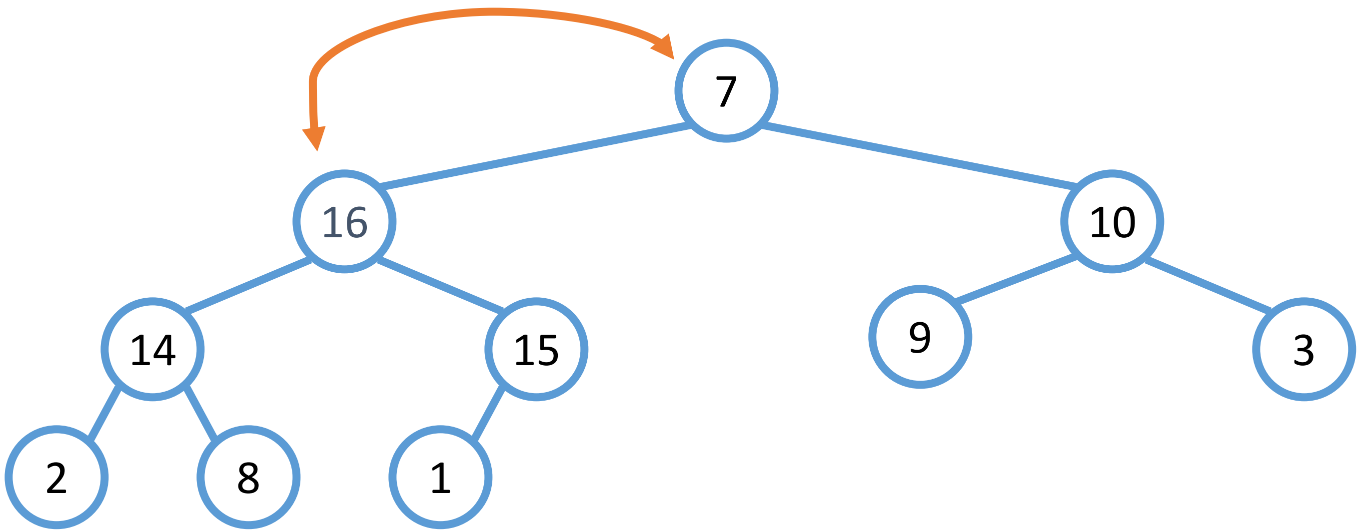
17	16	10	14	15	9	3	2	8	1	7

Delete max from heap Example



A =	7	16	10	14	15	9	3	2	8	1	~~17~~

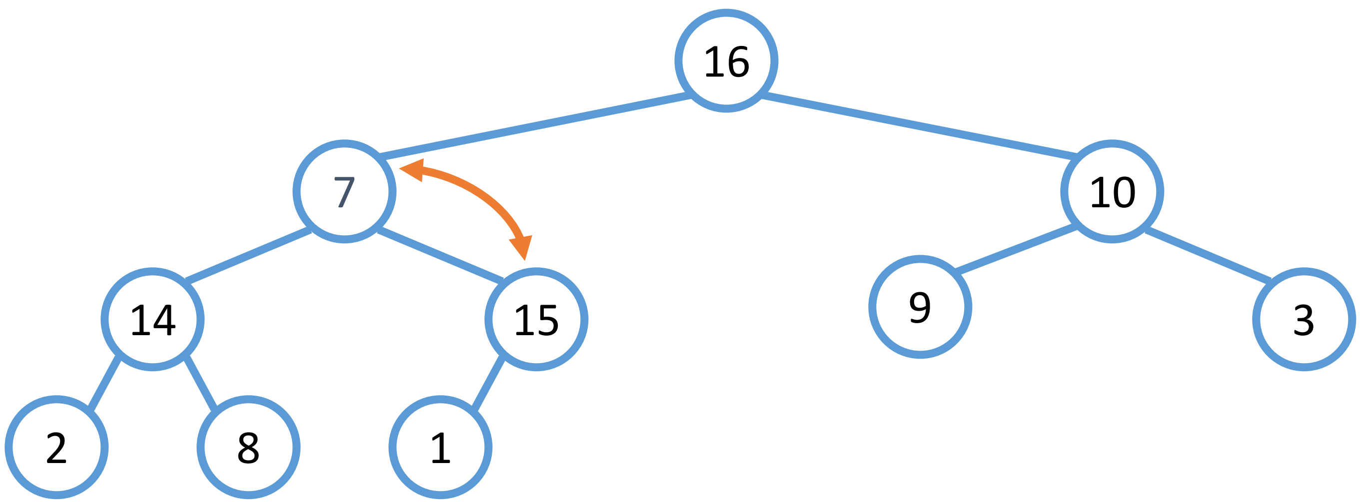
Delete max from heap Example



A =

7	16	10	14	15	9	3	2	8	1

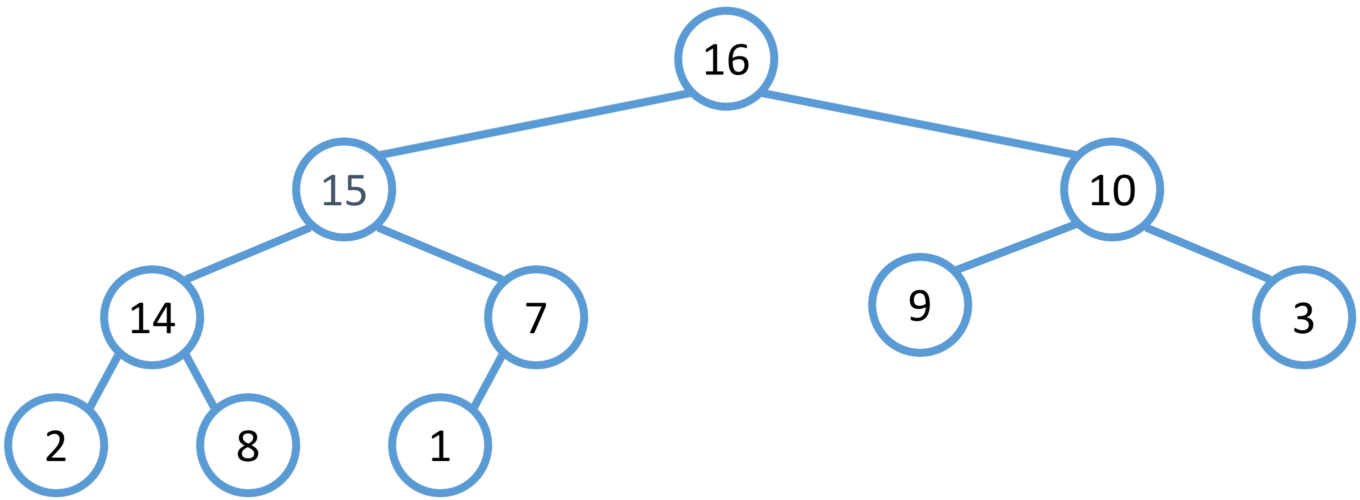
Delete max from heap Example



A =

16	7	10	14	15	9	3	2	8	1

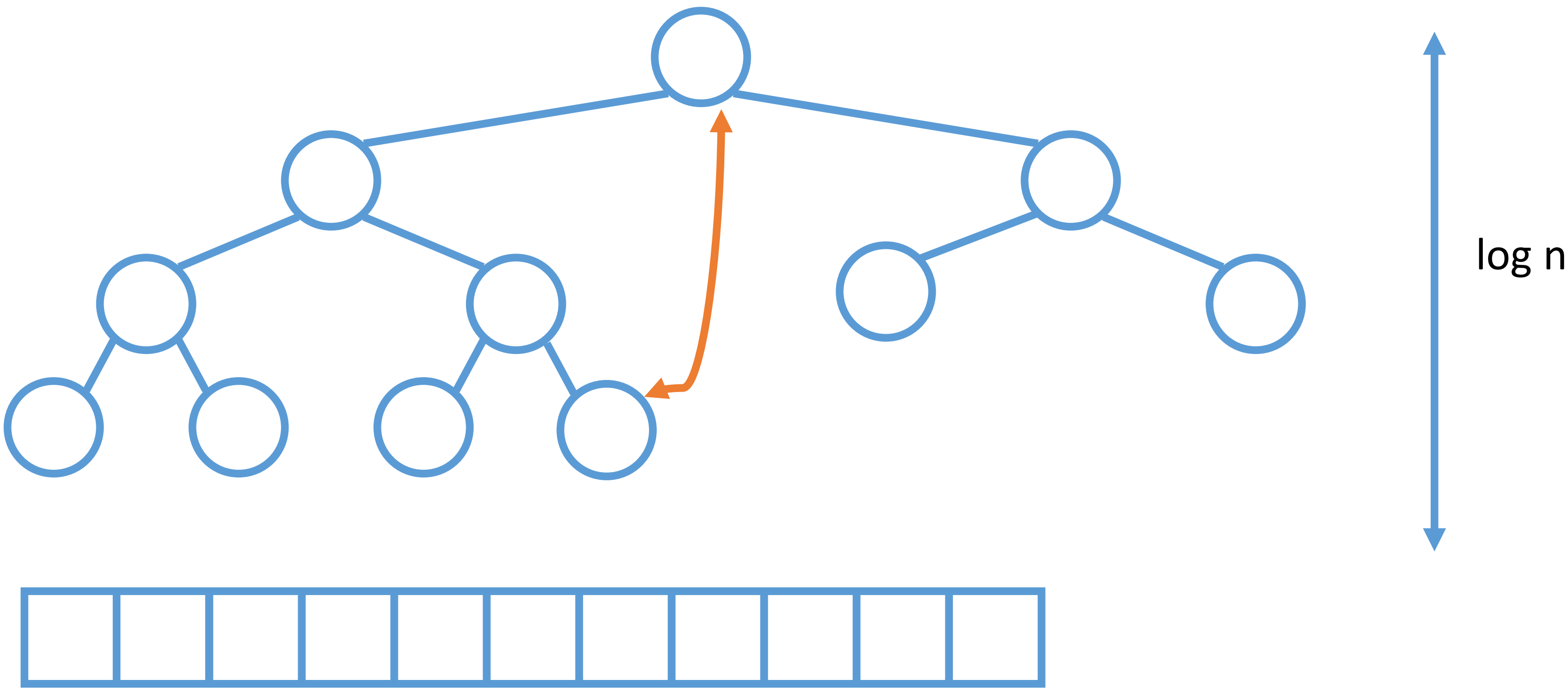
Delete max from heap Example



A =	16	15	10	14	7	9	3	2	8	1

Delete from heap Example

- Efficiency is O(log n)



Heap construction

- What if we are given an entire array?
- How can we transform it into a heap?

Heap construction

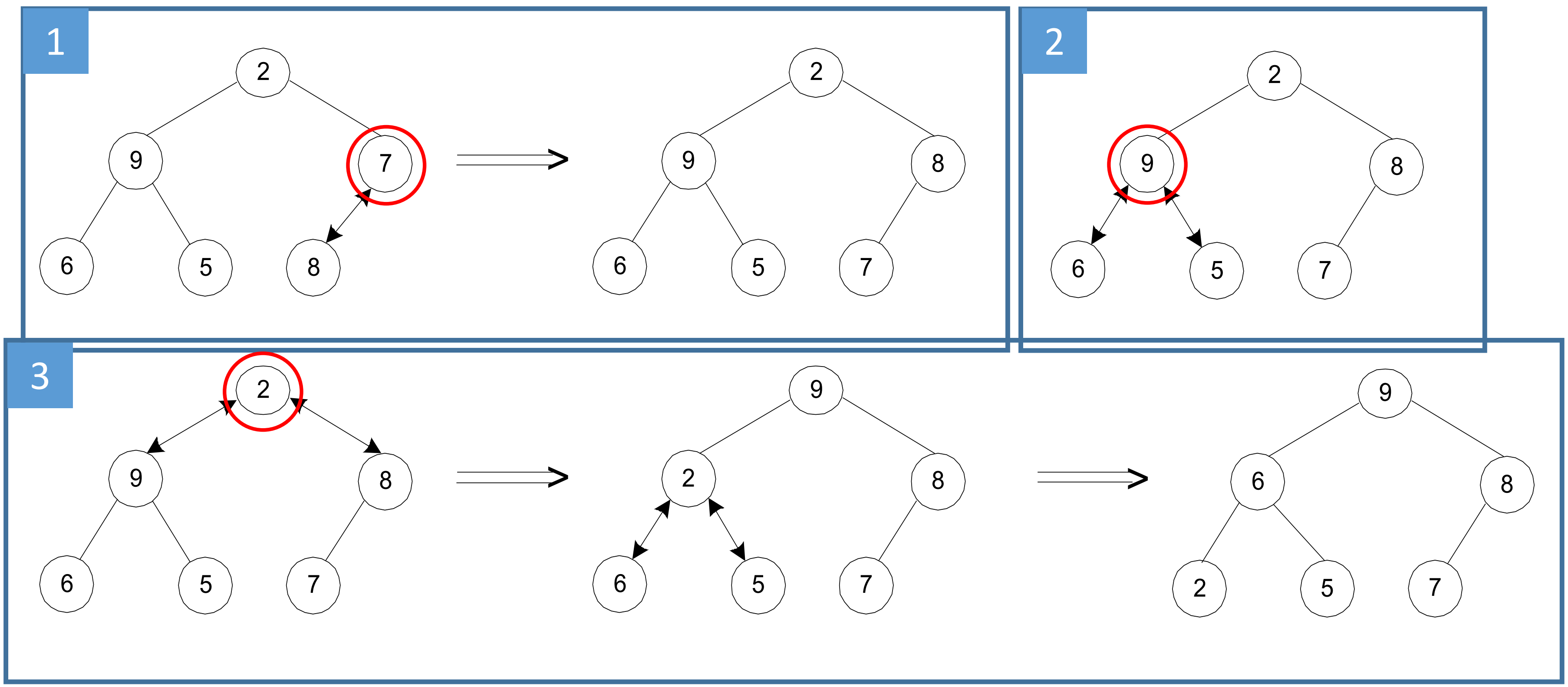
- Step 0: Initialize the structure with keys in the order given
  - (Done already)

- Step 1: Starting with the last (rightmost) parental node, fix the heap rooted at it, if it doesn't satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds

- Step 2: Repeat Step 1 for the preceding parental node

Example of heap construction

Construct a heap for the list 2, 9, 7, 6, 5, 8



Complexity of heap construction

- ~ n/2 “parental” modes → O(n)
- logn steps to fix each node → O(logn)
- Overall → O(nlogn)

Heapsort

Heapsort

- How can we use a Heap to sort an arbitrary array?
  - Stage 1: Transform the array into a heap (Construct a heap)
  - Stage 2: Call deleteMax N times to get all array elements in sorted order

Analysis of Heapsort

- Stage 1: Build heap for a given list of n keys
    - O(nlogn)

- Stage 2: Repeat operation of root removal n times (fix heap each time)
    - O(nlogn)

Practice problems

- Chapter 6.1, page 205, questions 2, 3, 7
- Chapter 6.4, page 233, question 1, 2, 7