Transform and Conquer

(Chapter 6)

Transform and Conquer:

This technique solves a problem by a *transformation* to

1. **Instance simplification**
   a more convenient instance of the same problem
2. **Representation change**
   a different representation of the same instance

Transform and Conquer:

1. **Instance simplification (Pre–sorting)**
   - *Checking element uniqueness in an array*
   - *Computing a mode*

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

Element uniqueness in an array

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

- Instance simplification (presorting)
  - Stage 1: sort by efficient sorting algorithm (e.g. mergesort)
  - Stage 2: scan array to check pairs of adjacent elements
  - Efficiency: O(nlog n) + O(n) = O(nlog n)

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

Transform and Conquer:

1. **Instance simplification (Pre-sorting)**
   - *Checking element uniqueness in an array*
   - *Computing a mode*

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

Computing a mode

- A *mode* is a 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
Frequencies	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1
Frequencies	1	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6
Frequencies	1	1	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequencies	1	1	1	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequencies	1	1	2	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequencies	2	1	2	1

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

i

Data	5	1	6	7
Frequencies	2	1	2	2

Computing a mode

- Brute Force:

5	1	6	7	6	5	7	6

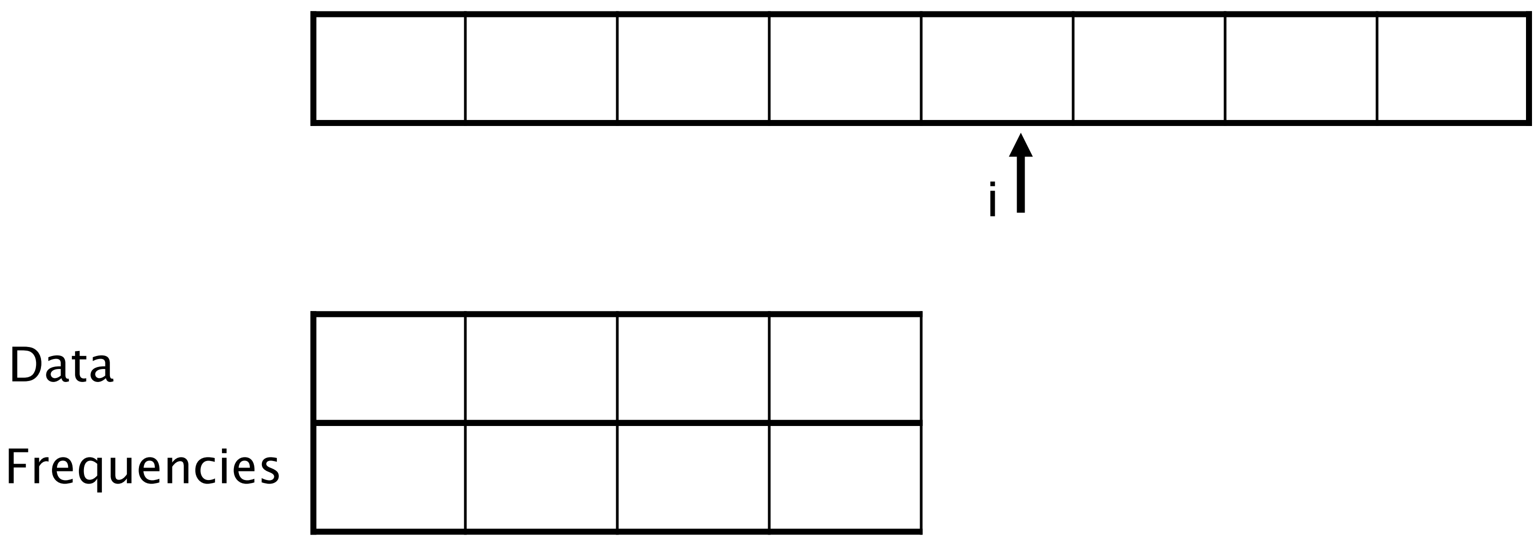
i

Data	5	1	6	7
Frequencies	2	1	3	2

Max

Computing a mode

- Efficiency (worst-case) :
  - A list with no equal elements
  - i^{th} element is compared with $i - 1$ elements



Computing a mode

- Efficiency (worst-case):
  - Creating auxiliary list: $0 + 1 + 2 + \cdots + n - 1 = \mathrm{O}(n^2)$
  - Finding max: O(n)

Efficiency (worst-case): $\mathrm{O}(n^2)$

Computing a mode(pre-sorting)

- Sort the input
- All equal values will be adjacent to each other
- 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 $i \leq n-1$ **do**
    $runlength \leftarrow 1$; $runvalue \leftarrow A[i]$
    while $i + runlength \leq n-1$ **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:
  - $T(n) = T_{sort}(n) + T_{search}(n) =$
    $(n \log n) + (n) = (n \log n)$

Searching with presorting

Problem: Search for a given K in A[0..n−1]

Presorting-based algorithm:

Stage 1 Sort the array by an efficient sorting algorithm

Stage 2 Apply binary search

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

Good or bad? (sequential search is O(n))

Why do we have our dictionaries, telephone directories, etc. sorted?

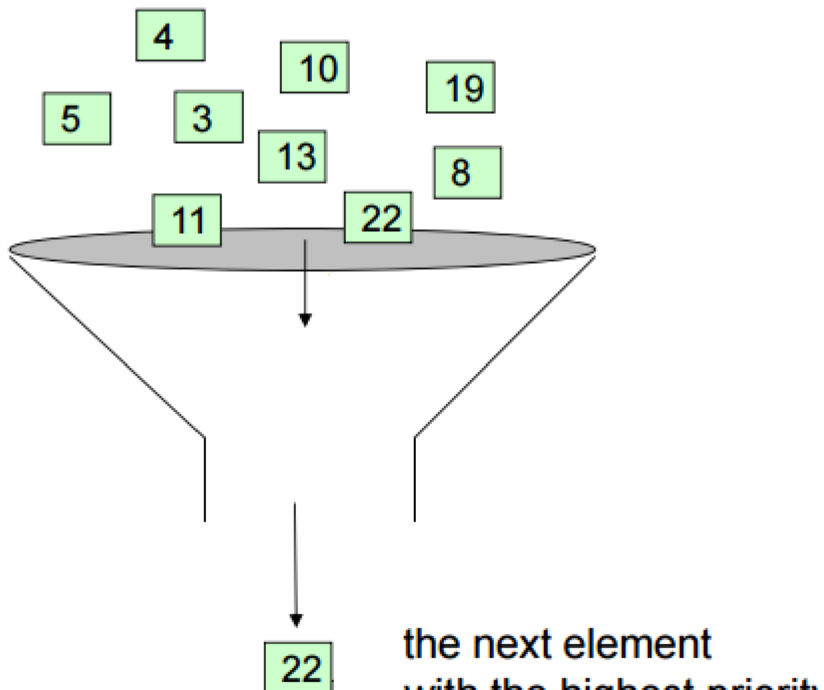
Transform and Conquer:

1. **Instance simplification (Pre-sorting)**
   - *Checking element uniqueness in an array*
   - *Computing a mode*

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

Sample problem

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



Simple Implementations

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

7	5	8	1	9

- SortedArraylist
  - Insert: O(logn + n)
  - deleteMax: O(n)

9	8	7	5	1

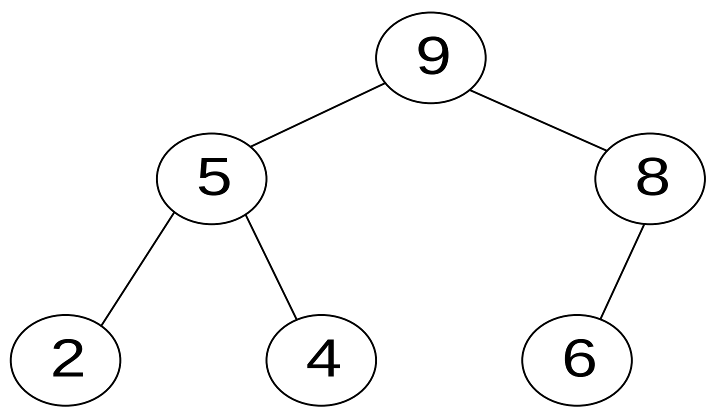
Representation change

- Idea:
  - Given an array
  - Transform to a new data structure
    (Make a “**heap**” out of it)

- Efficiency of heap:
  - Insert an item: O(logn)
  - Delete an item with max priority: O(logn)

Heap definition

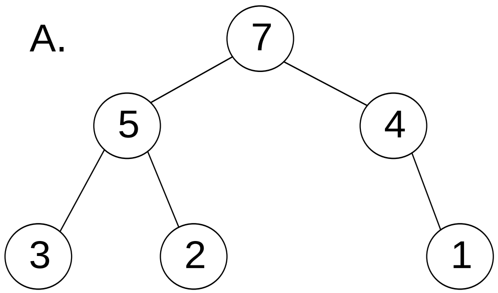
- Almost complete binary tree.
  - filled on all levels, except last, where filled from left to right
- Every parent is greater than (or equal to) child



Heap or No Heap?

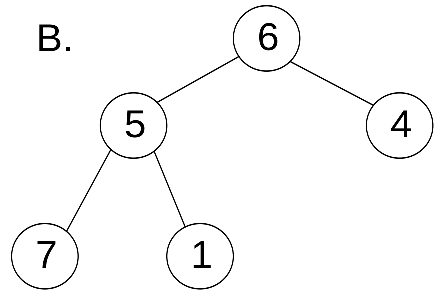
NO

A.



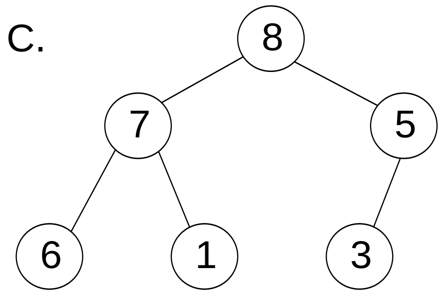
NO

B.



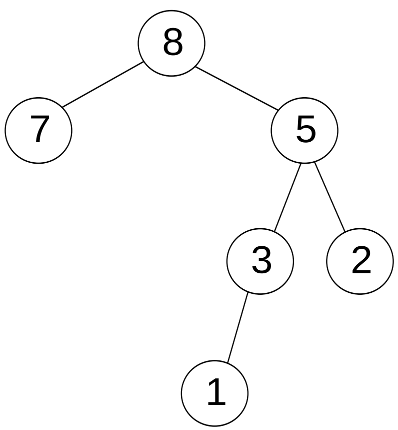
YES

C.



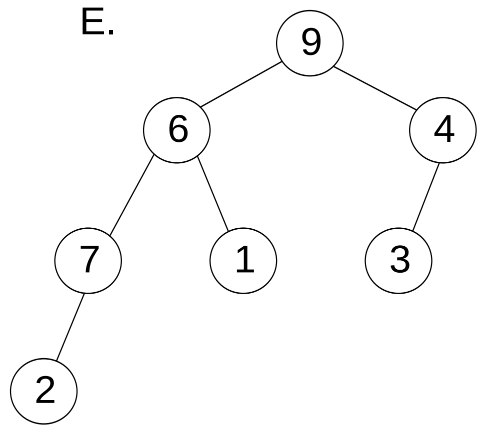
NO

D.



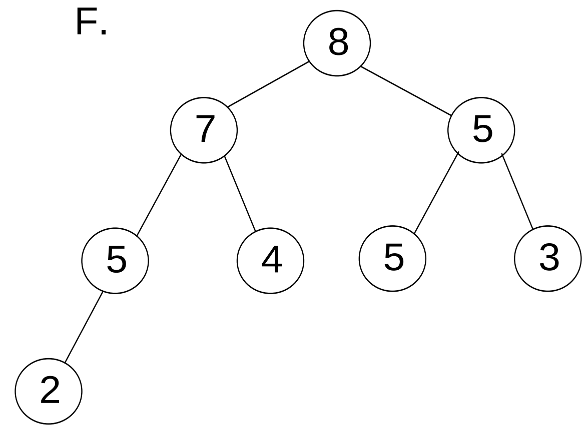
NO

E.



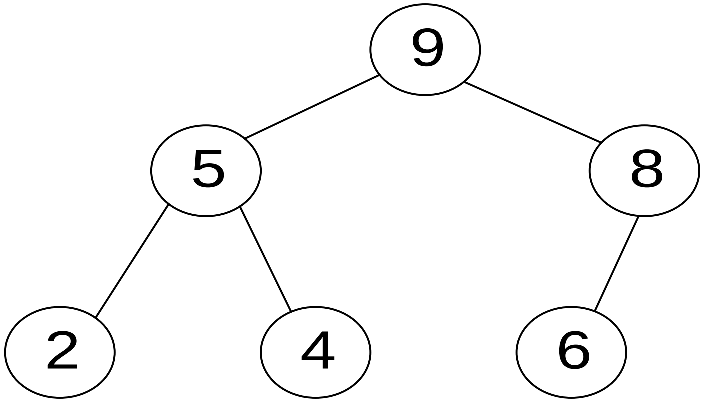
YES

F.



Heap properties

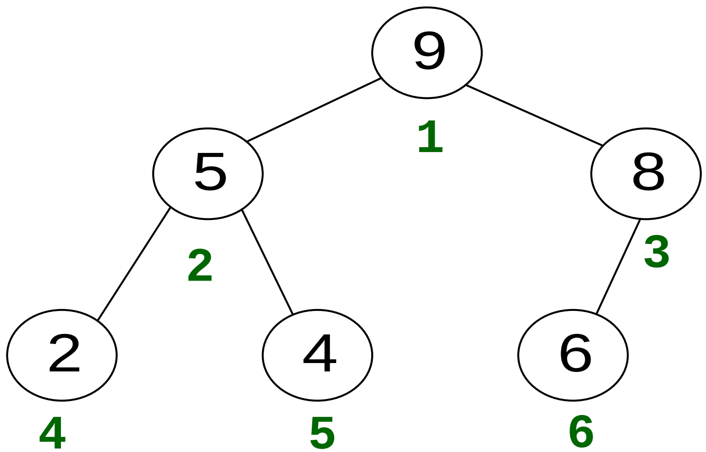
- Max element is in root.
- Heap with N elements has height = $\lfloor \log_2 N \rfloor$.



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`

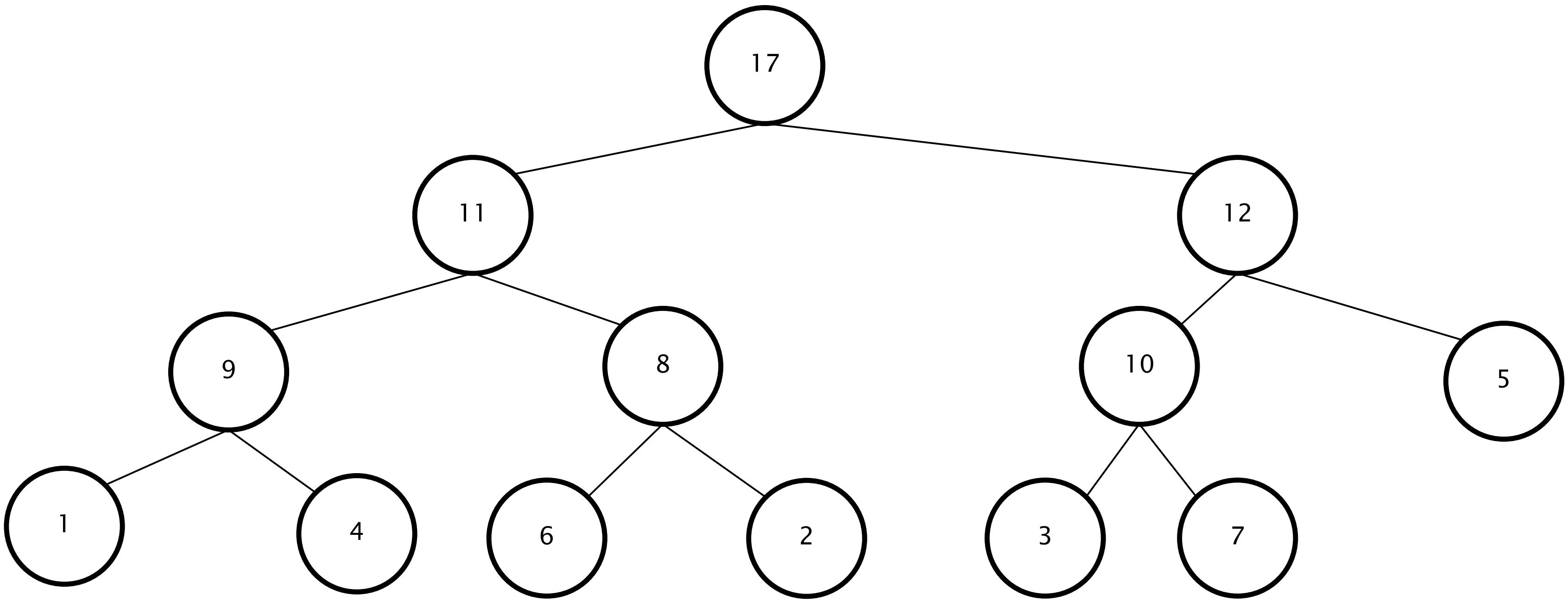


0	1	2	3	4	5	6
	9	5	8	2	4	6

Example1

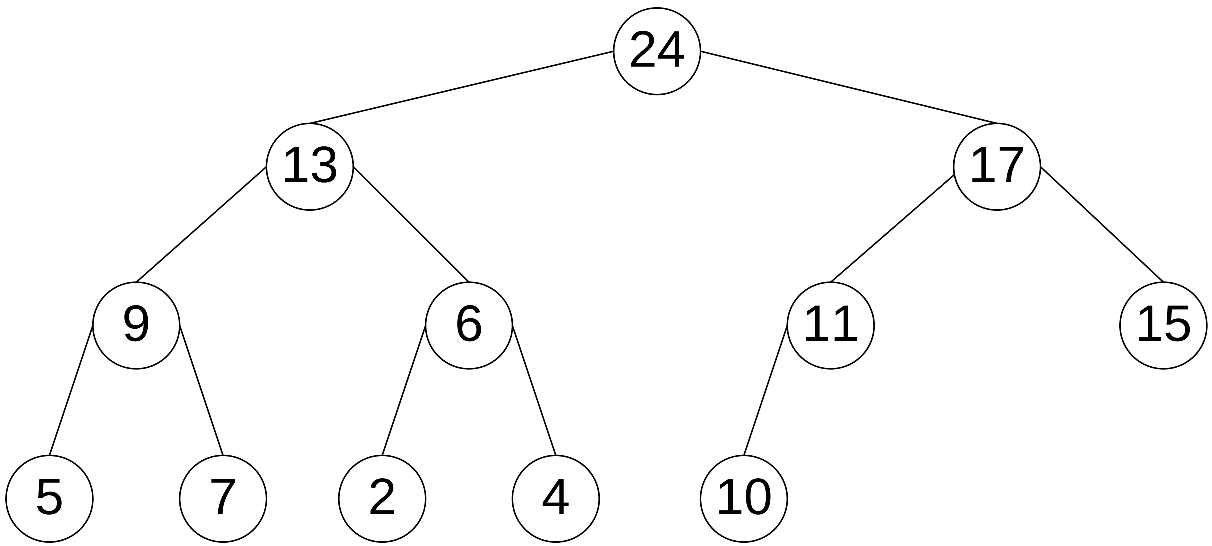
- draw the tree representation of this heap

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



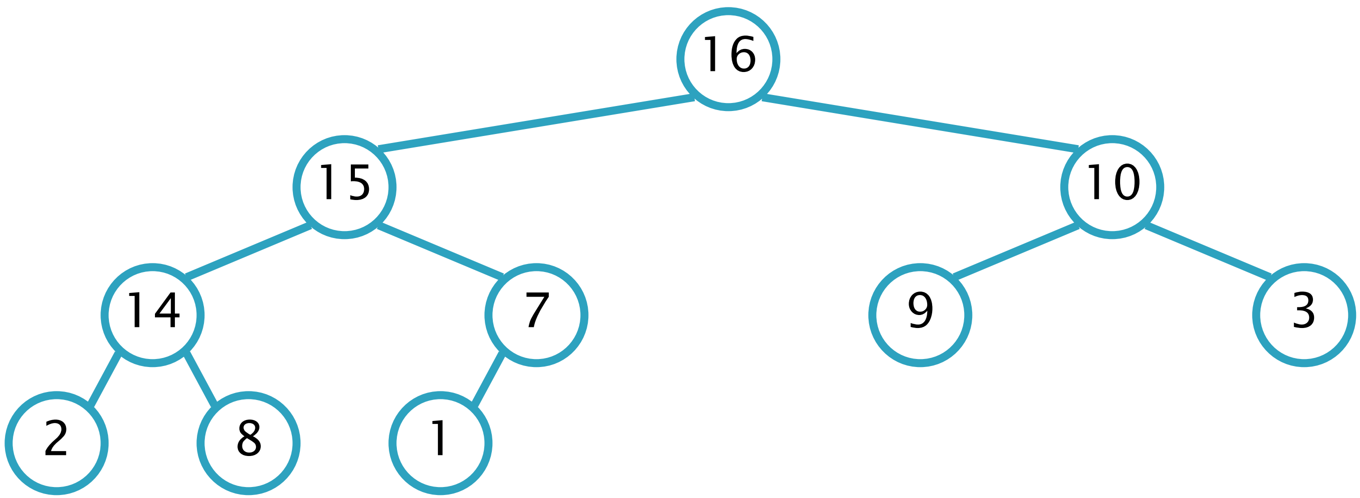
Index	1	2	3	4	5	6	7	8	9	10	11	12
value	24	13	17	9	6	11	15	5	7	2	4	10

Heap insertion

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

Insert to heap Example

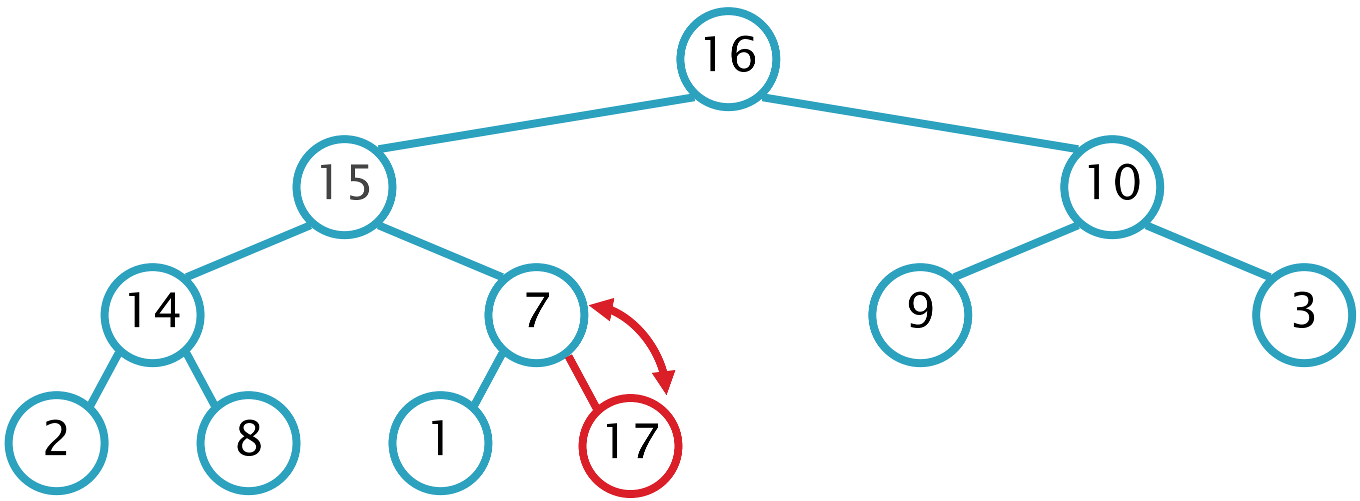
- Insert 17



A =

16	15	10	14	7	9	3	2	8	1

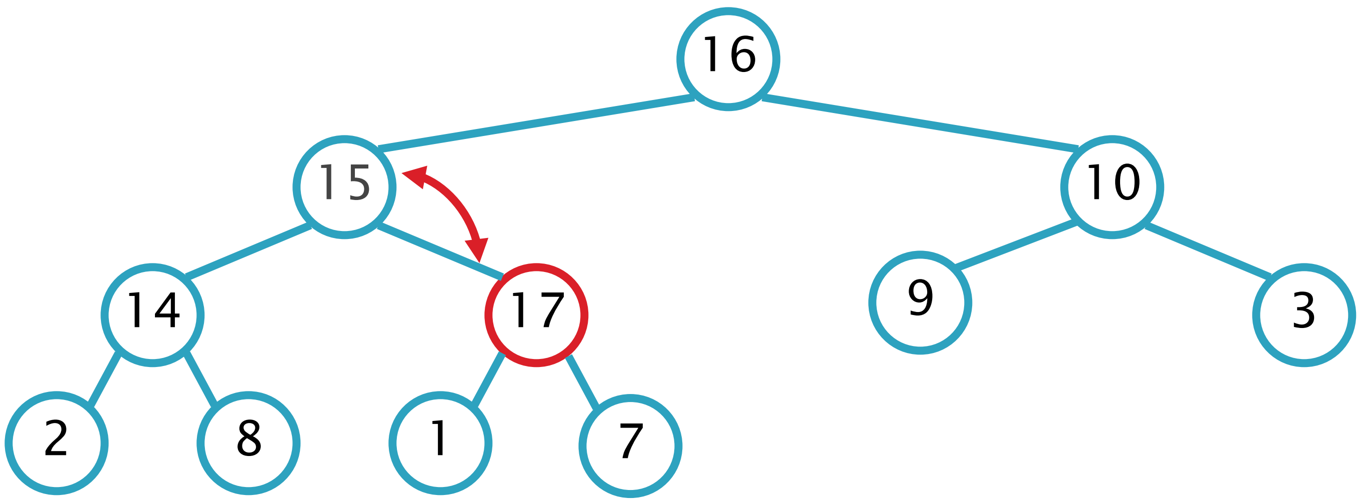
Insert to heap Example



A =

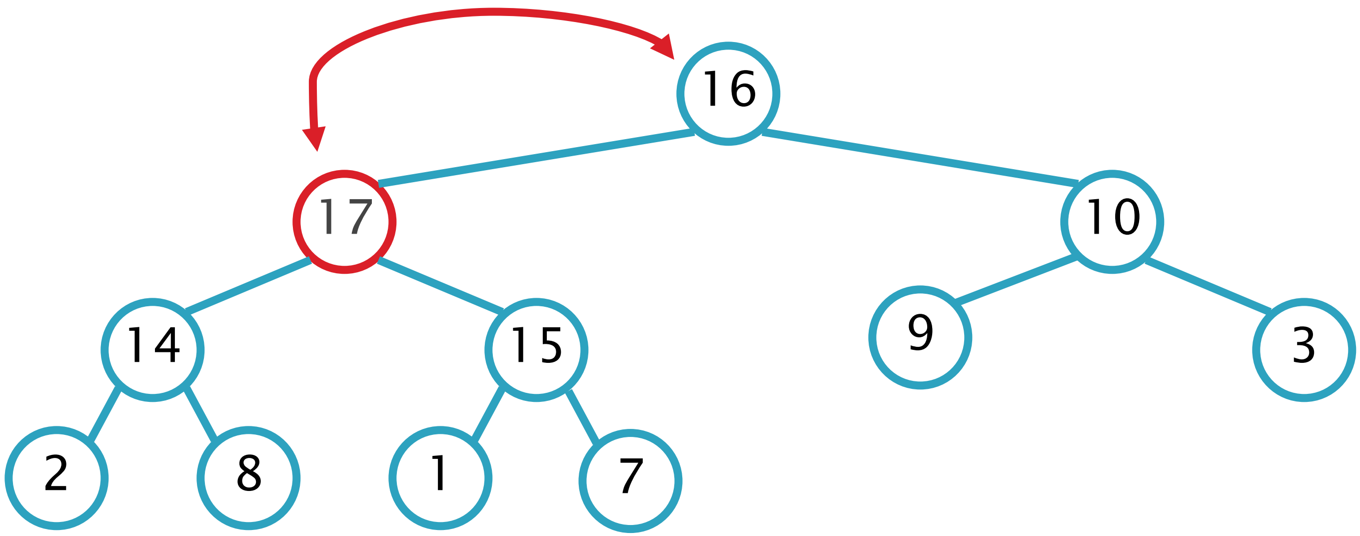
16	15	10	14	7	9	3	2	8	1	17

Insert to heap Example



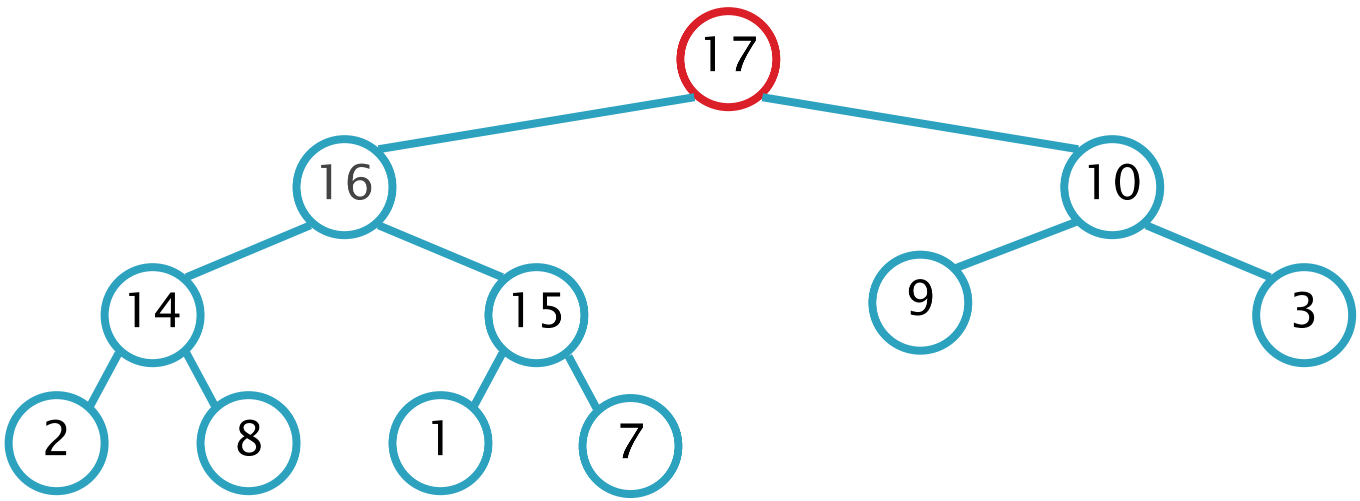
A =	16	15	10	14	17	9	3	2	8	1	7

Insert to heap Example



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

Insert to heap Example

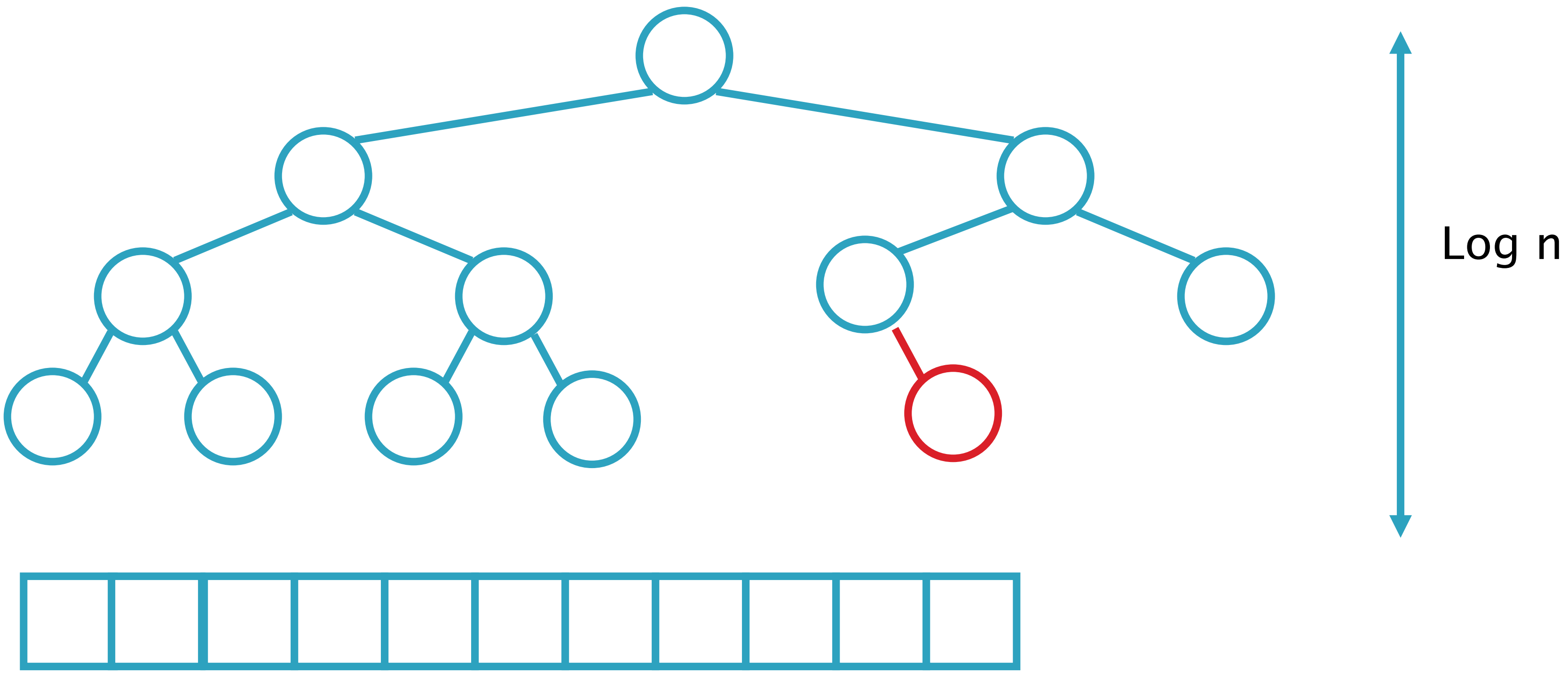


A =

17	18	10	14	15	9	3	2	8	1	7

Insert to heap Example

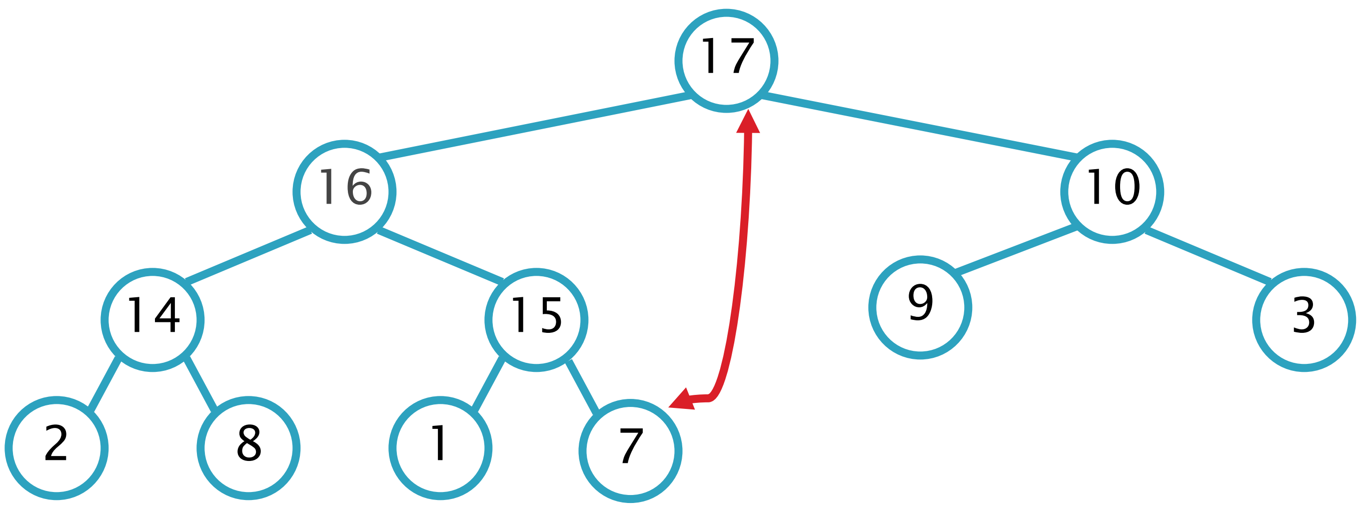
- Efficiency is O(log n)



Delete max from Heap

- Exchange root with rightmost leaf
- Delete element
- Bubble root down until it's heap ordered

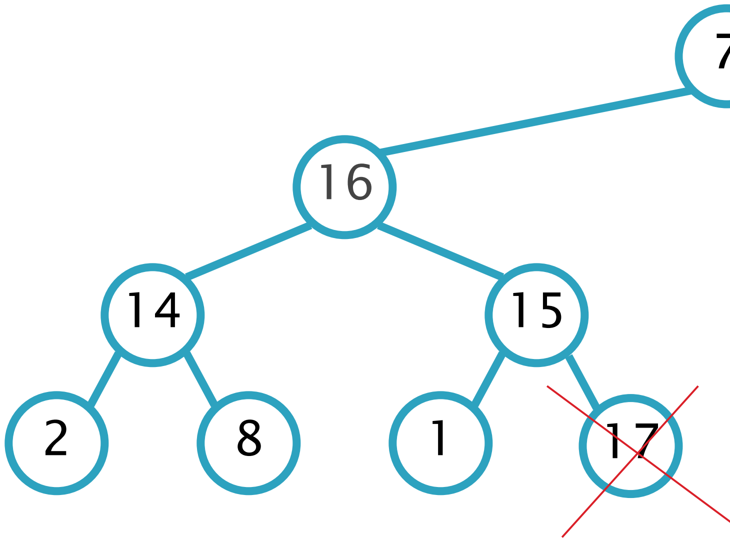
Delete max from Heap Example



A =

17	16	10	14	15	9	3	2	8	1	7

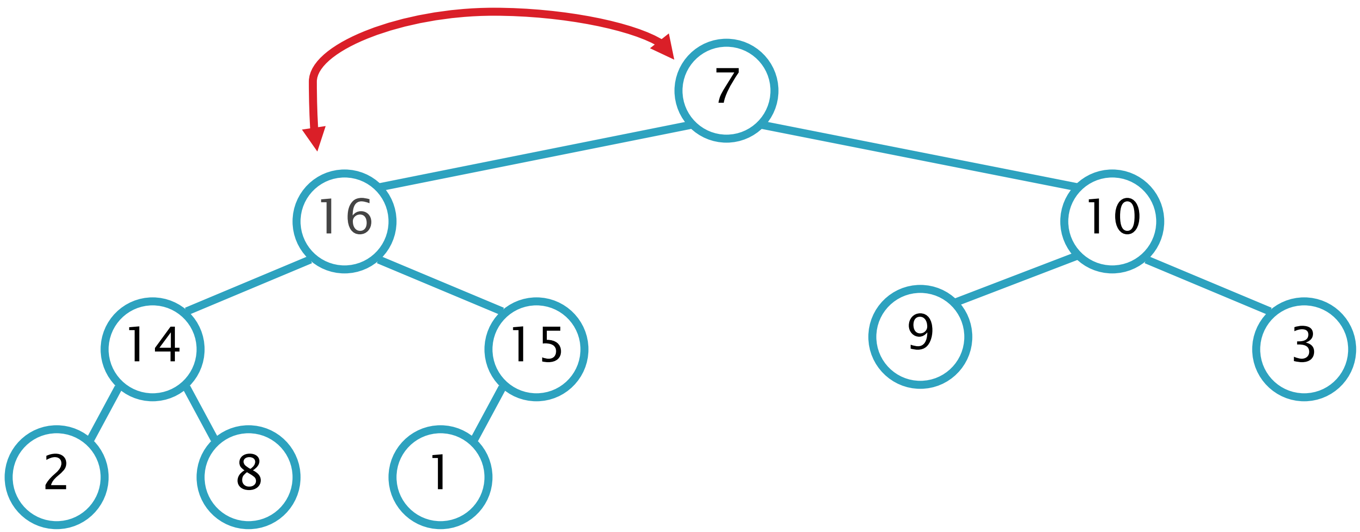
Delete max from Heap Example



A =

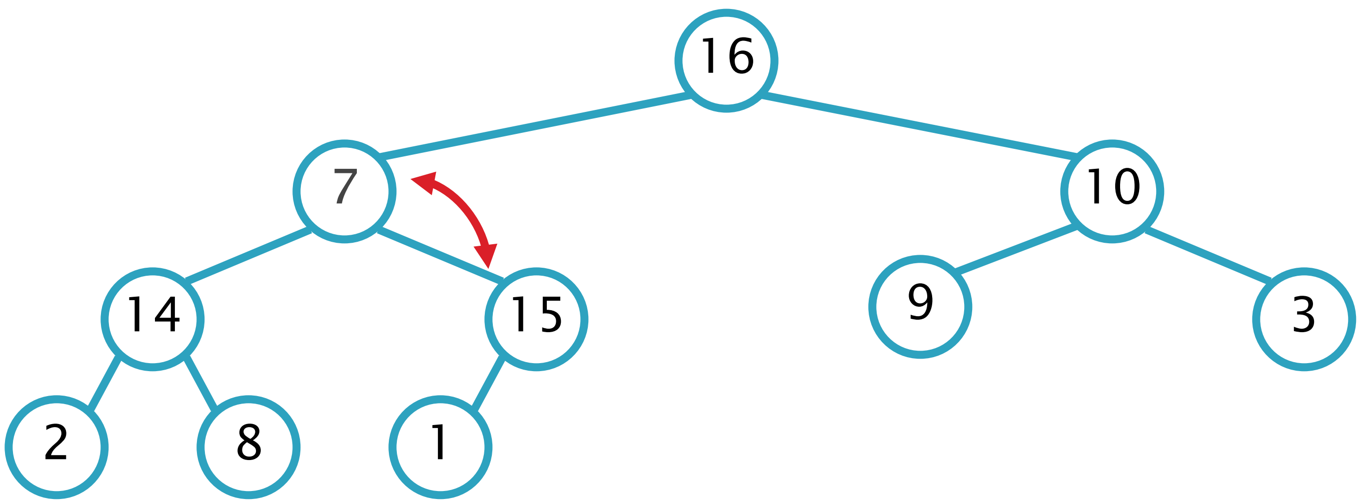
7	16	10	14	15	9	3	2	8	1	~~17~~

Delete max from Heap Example



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

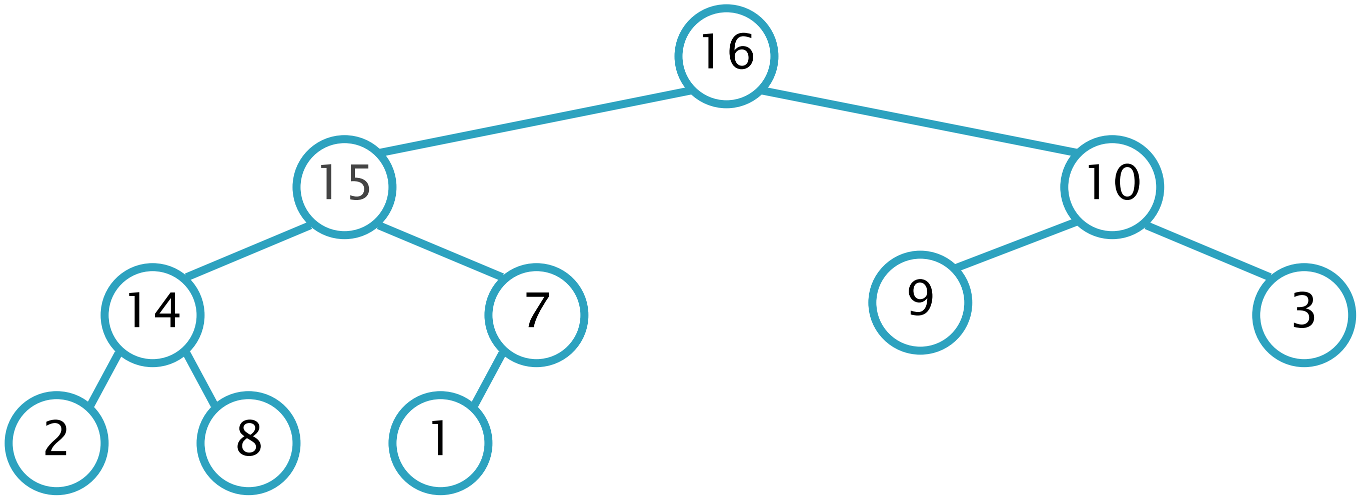
Delete max from Heap Example



A =

16	7	10	14	15	9	3	2	8	1

Delete max from Heap Example

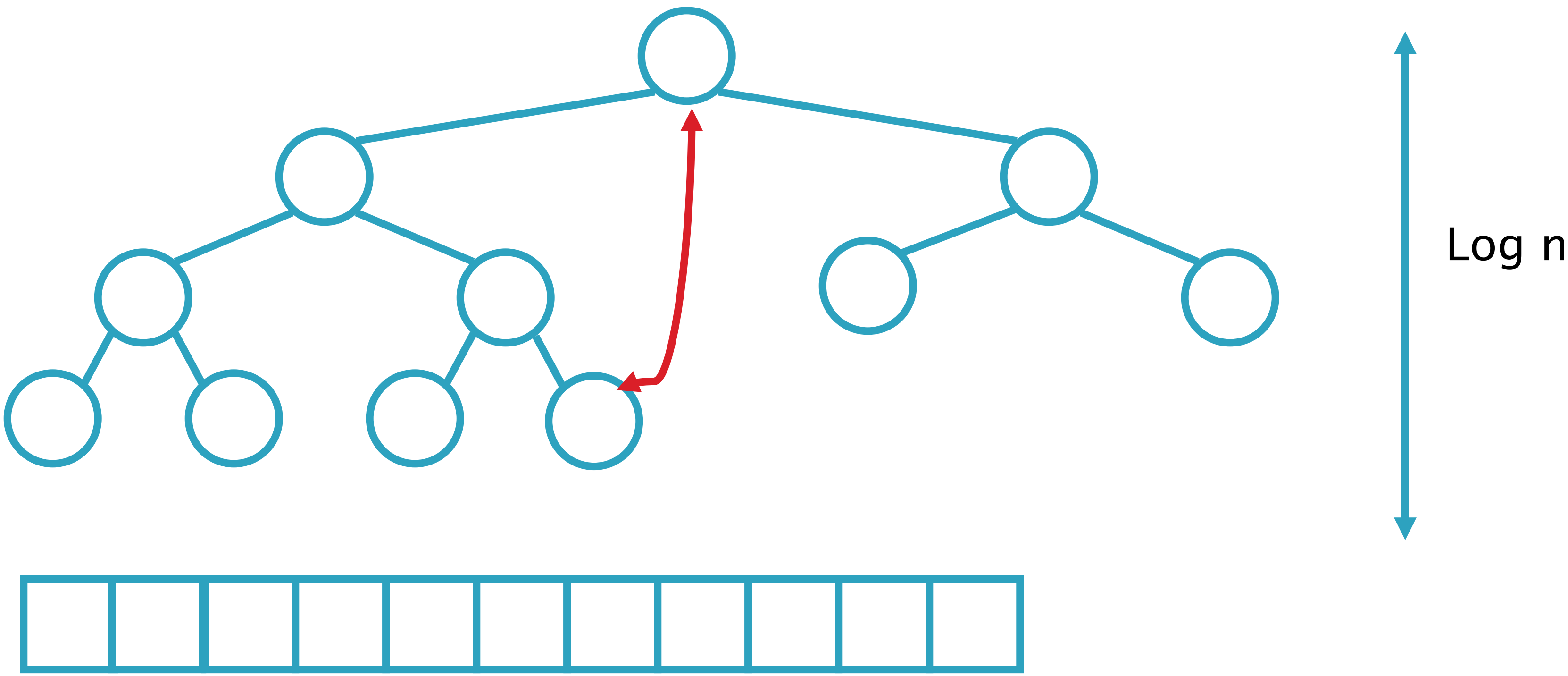


A =

16	15	10	14	7	9	3	2	8	1

Delete from heap Example

- Efficiency is O(log n)



Heap Construction

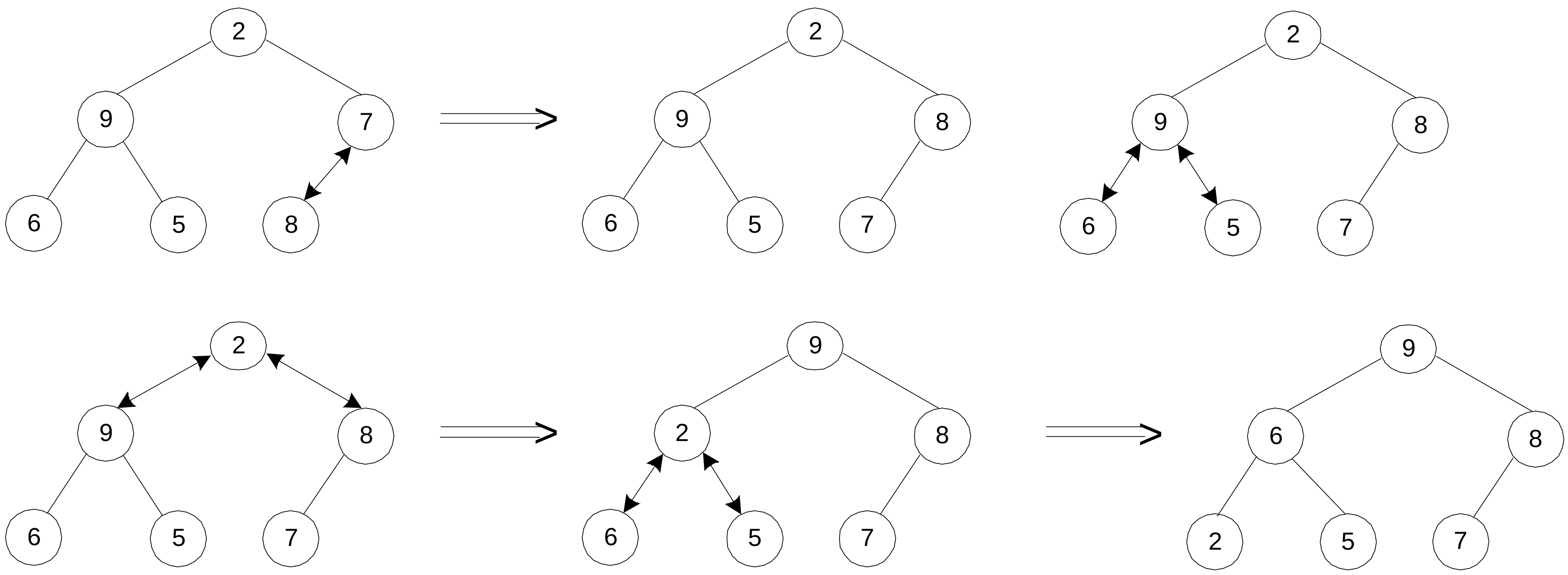
Step 0: Initialize the structure with keys in the order given

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



Transform and Conquer:

1. **Instance simplification (Pre-sorting)**
   - *Checking element uniqueness in an array*
   - *Computing a mode*

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

HeapSort

How can we use a Heap to sort an arbitrary array?

1. transform the array into a heap (Construct a heap)
2. call RemoveMax to get all array elements in sorted order

Example of Sorting by Heapsort

Sort the list 2, 9, 7, 6, 5, 8 by heapsort

Stage 1 (heap construction)

2	9	7	6	5	8
2	9	8	6	5	7
2	9	8	6	5	7
9	2	8	6	5	7
9	6	8	2	5	7

stage 2

9	6	8	2	5	7
7	6	8	2	5	
8	6	7	2	5	
5	6	7	2		
7	6	5	2		
2	6	5			
6	2	5			
5	2				
5	2				
2					

Analysis of Heapsort

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

Stage 2: Repeat operation of root removal *n*−1 times (fix heap)
O(nlogn)

Try it/ homework

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