CMP418: Algorithm and Complexity Analysis (3 units)

Lecture 6: Transform and Conquer

MR. M. YUSUF

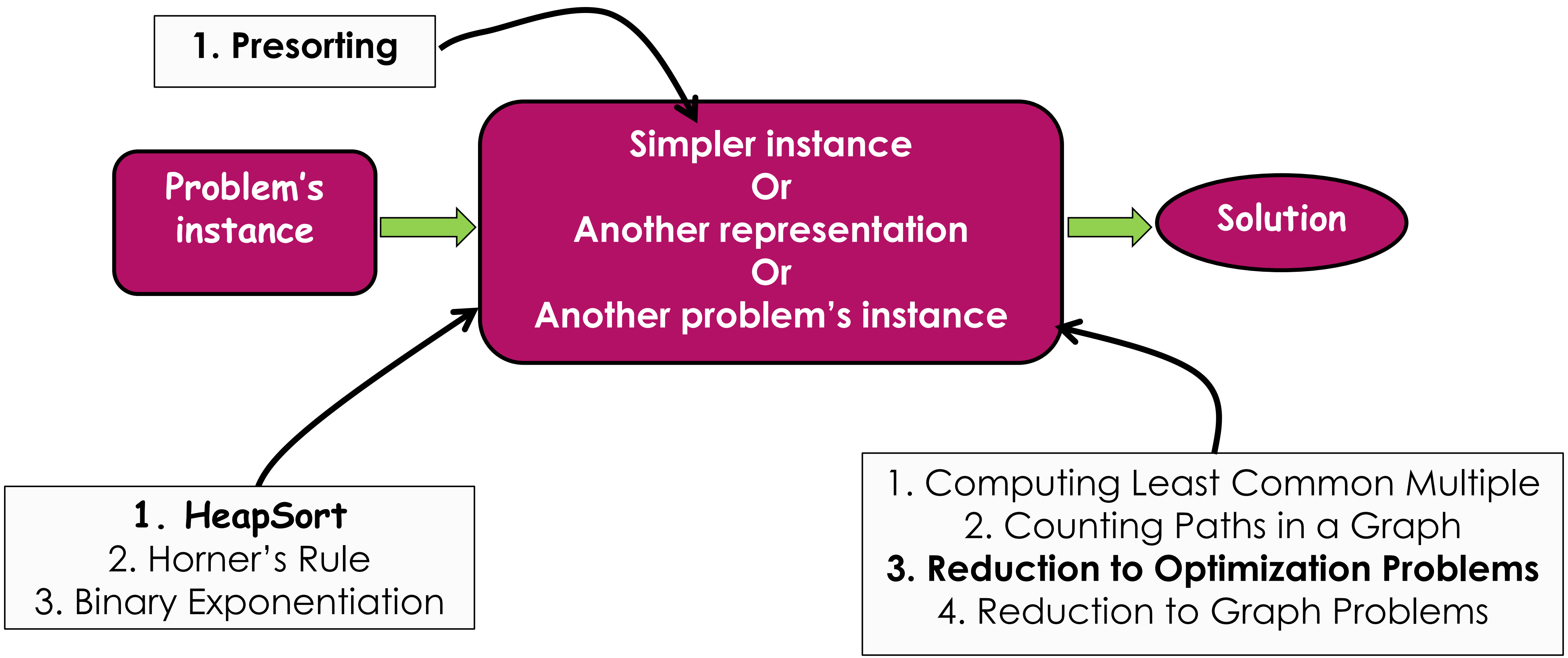
Outline

- What is Transform and Conquer
- Instance Simplification: Presorting
- Representation Change: Heap and Heapsort
- Problem Reduction: Linear Programming

What is Transform and Conquer (TnC)

- There are three major variations of TnC problems
  - **Instance Simplification:** Transform to a ***simpler*** or ***more convenient*** instance of the **<u>same problem</u>**
  - **Representation Change:** Transform to a **different** representation of the **<u>same instance</u>**
  - **Problem Reduction:** Transform to an ***instance*** of a **<u>different problem</u>** for which you know an efficient algorithm
- You can perform TnC in just two-step process:
  - **Step 1:** Modify problem instance to something easier to solve
  - **Step 2:** Conquer!

What is Transform and Conquer (TnC)

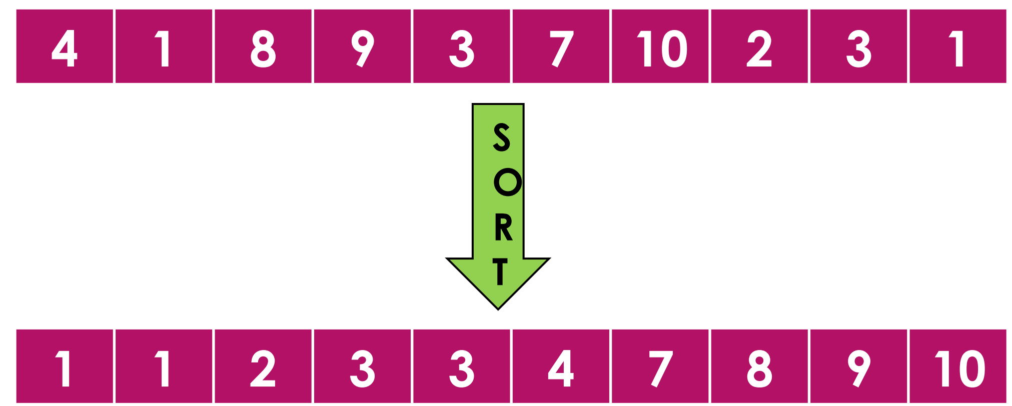


Outline

- What is Transform and Conquer
- Instance Simplification: Presorting
- Representation Change: Heap and Heapsort
- Problem Reduction: Linear Programming

Transform and Conquer: Presorting

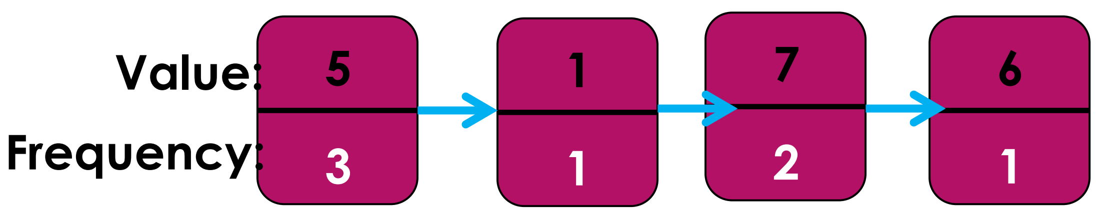
ALGORITHM PresortElementUniqueness(A[0..n-1])
//Input: sort the array A
//Output: true if area is sorted, false if not
for i <- 0 **to** n-2 **do**
    if A[i] = A[i+1]
        return false
return true



$$T(n) = \max(T_{sort}(n) \quad + \; T_{scan}(n))$$
$$\in \max(\Theta(n\log_2 n) + \Theta(n))$$
$$= \Theta(n\log_2 n)$$

Presorting: Computing Mode with Brute Force

- **Mode** the value with the highest frequency of occurrence in a given list
- The mode of the list | 5 | 1 | 5 | 7 | 6 | 5 | 7 | is 5
- **Brute-force: *scan*** the list and compute the ***frequencies of all distinct*** values, then ***find*** the ***value*** with the ***highest*** frequency
  - Store the values already encountered along with their frequencies, in a separate list.



$$T(n) = 0 + 1 + \ldots + (n-1)$$

$$= \frac{(n-1)n}{2}$$

$$\in \Theta(n^2)$$

Presorting: Computing Mode with TnC

- If we sort the array first, ***all equal values*** will be ***adjacent*** to each other.
- To compute the mode, all we need to know is to find the ***longest run of adjacent equal values*** in the sorted array

```
ALGORITHM PresortMode( A[0..n-1] )
//sort the array A
//input array A
//Output: returns the Mode value
i <- 0
modefrequency <- 0
while i ≤ n-1 do
    runlength <- 1;
    runvalue <- A[i]
    while i+runlength ≤ n-1 and A[i+runlength] = runvalue
        runlength <- runlength+1
    if runlength > modefrequency
        modefrequency <- runlength;
        modevalue <- runvalue
    i <- i+runlength
return modevalue
```

1	1	2	3	3	3	7	8	9	10

Outline

- What is Transform and Conquer
- Instance Simplification: Presorting
- Representation Change: Heap and Heapsort
- Problem Reduction: Linear Programming

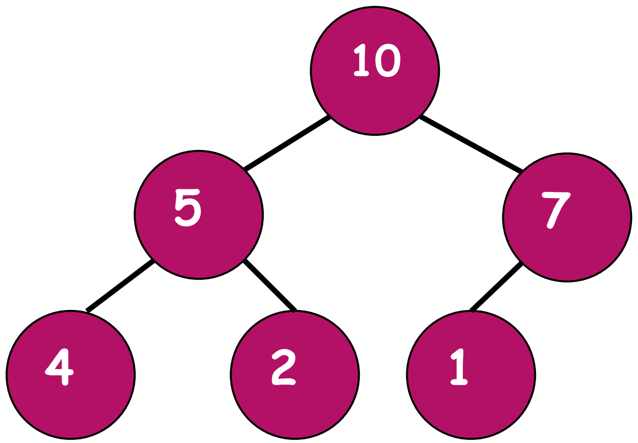
Transform and Conquer: Heap & Heap Sort

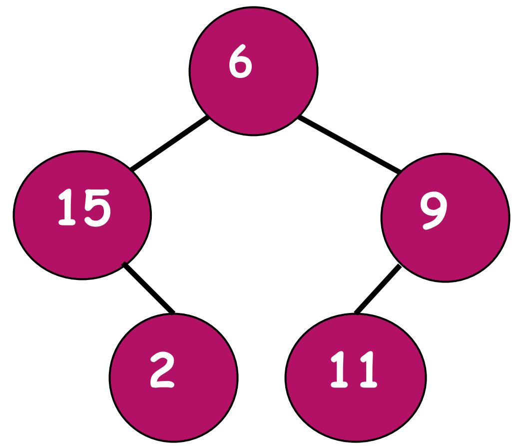
- Uses a clever data structure called “Heap”
- It transforms an array into a Heap and then sorting becomes very easy
- Heap has other important uses, like in implementing “priority queue”
- Recall, priority queue is a multiset of items with an orderable characteristic called “priority”, with the following operations:
  - **Find** an item with the **highest** priority
  - **Deleting** an item with the **highest** priority
  - **Adding** a new item to the **multiset**

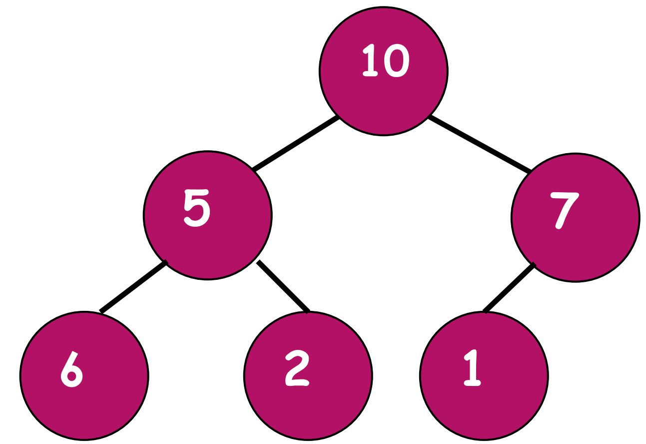
Transform and Conquer: Heap

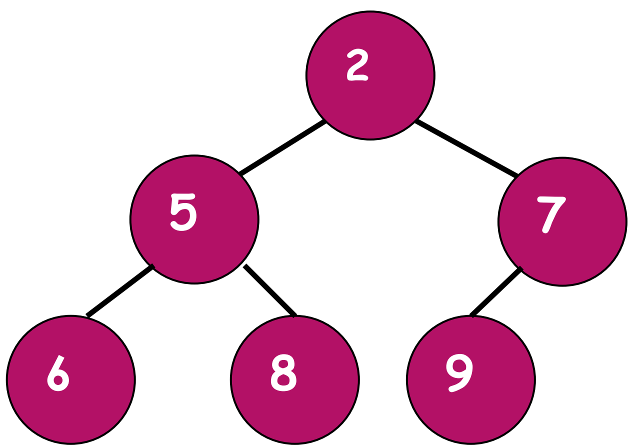
- A "heap" is a tree with **one** key per node
- Once a Heap met the following **two conditions**, it can be considered as **binary tree**.
  1. Shape property: Binary tree is ***essentially complete*** all levels are full except possibly the last level, where only some ***right most leaves*** may be missing
  2. Parental dominance: This condition also applies to the leaves
     - When key in each node ≥ the keys in its children, Its referred to as "**max heap**"
     - When key in each node ≤ the keys in its children, its referred to as "**min heap**"

Heap: Examples



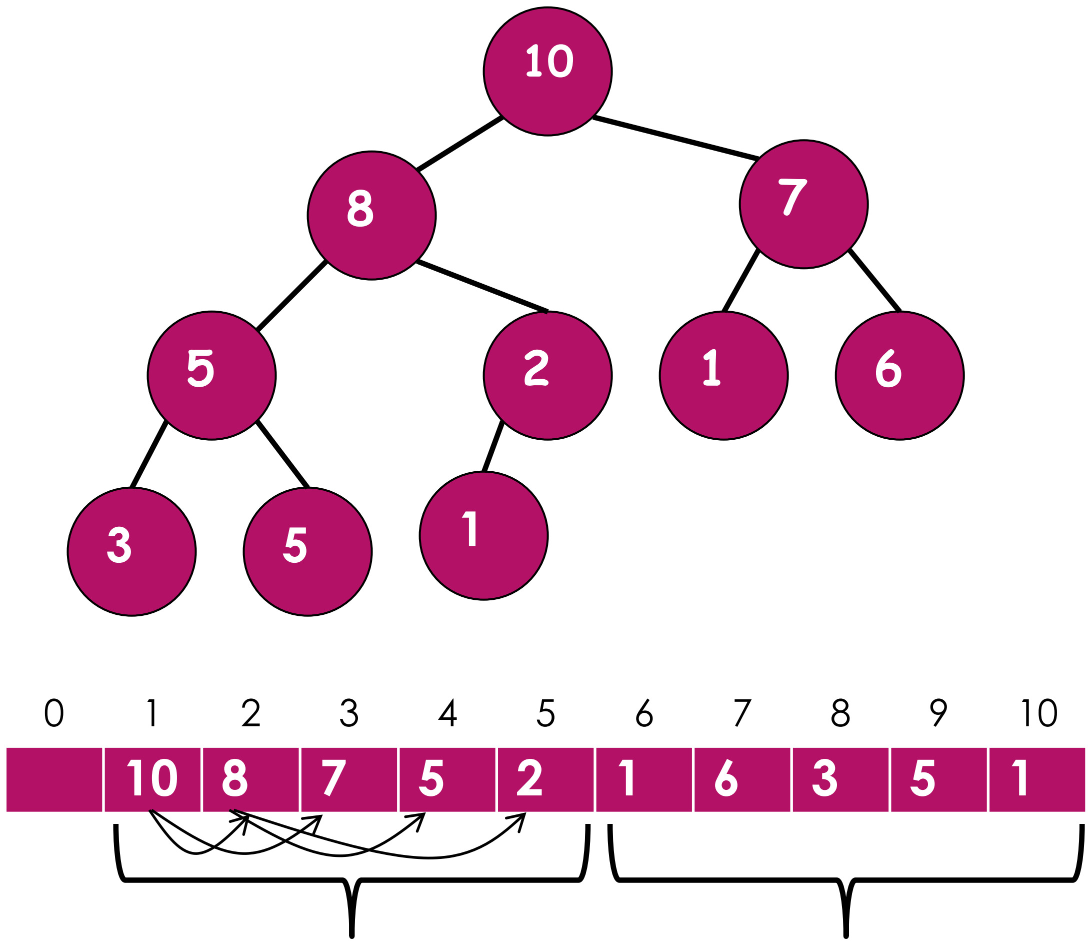






Which of the following is a heap?

Heap: Examples



Note:
Sequence of values on a path from the root to a leaf is nonincreasing

There is no left-to-right order in key values, though

ALGORITHM Parent(i)
return $\lfloor i/2 \rfloor$

ALGORITHM Left(i)
return 2i

ALGORITHM Right(i)
return $2i + 1$

Important Properties of Heaps:

1. There exists one essentially **complete binary tree** with n nodes, its height is $\lfloor log_2 n \rfloor$

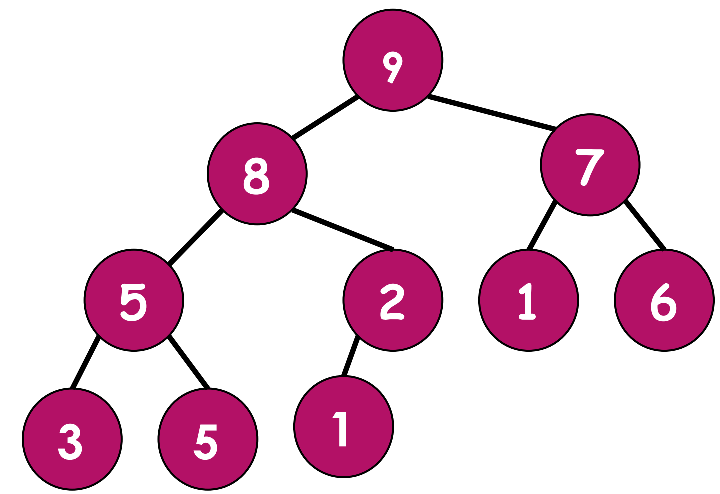
2. The **root** of a heap contains the **largest element**

3. A **node** with all its **descendants** is also a heap

4. Heap can be implemented as an array by recording its elements in the **top-down**, **left-right fashion**. **H[0]** is **unused** or contains a number bigger than all the rest.

   a. Parental node keys will be in first $\lfloor n/2 \rfloor$ positions, leafs will Be in the last $\lceil n/2 \rceil$ positions

   b. Children of key at index i (1 ≤ i ≤ $\lfloor n/2 \rfloor$) will be in positions 2i and 2i+1. The parent of a key at index i (2 ≤ i ≤ n) will be in Position $\lfloor i/2 \rfloor$.

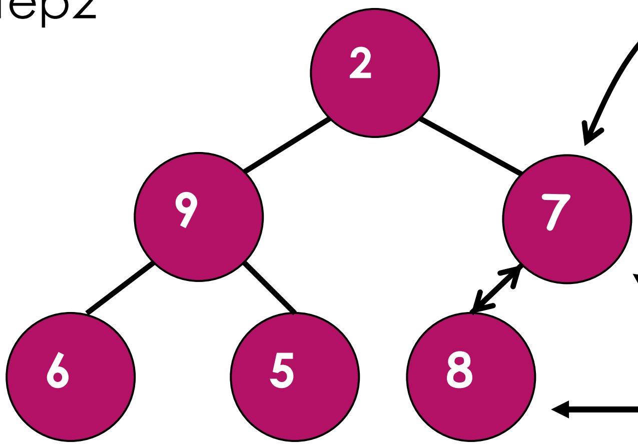


	0	1	2	3	4	5	6	7	8	9	10
H:		9	8	7	5	2	1	6	3	5	1

H[i] ≥ max{ H[2i], H[2i+1] } for i = 1, … , $\lfloor n/2 \rfloor$

Heapifying: Bottom-Up Heap Construction

Step2

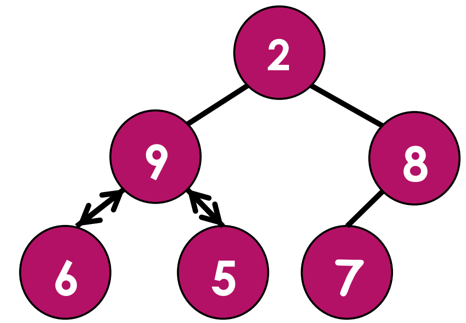


1. **Start at the last parent at index $\lfloor n/2 \rfloor$**

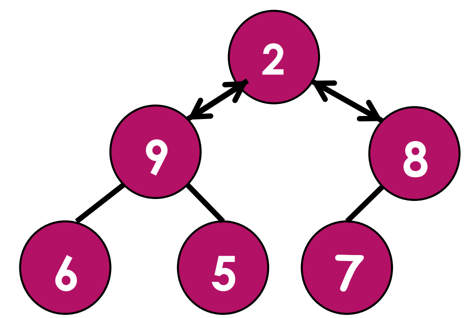
2. **Check if parental dominance holds for this node's key**

3. **If it does not, Exchange the node's key K with the larger key of its children**

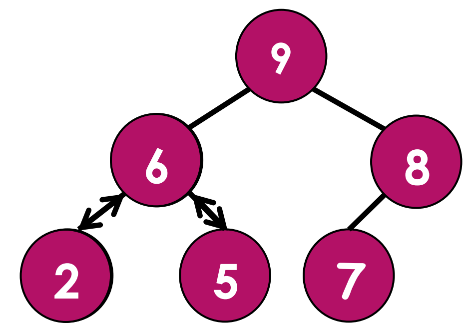
4. **Check if parental dominance holds for K in its new position and so on…**
Stop after doing for root.



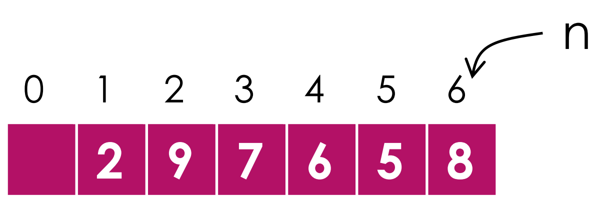
Step3



Step4



Step5



Step1

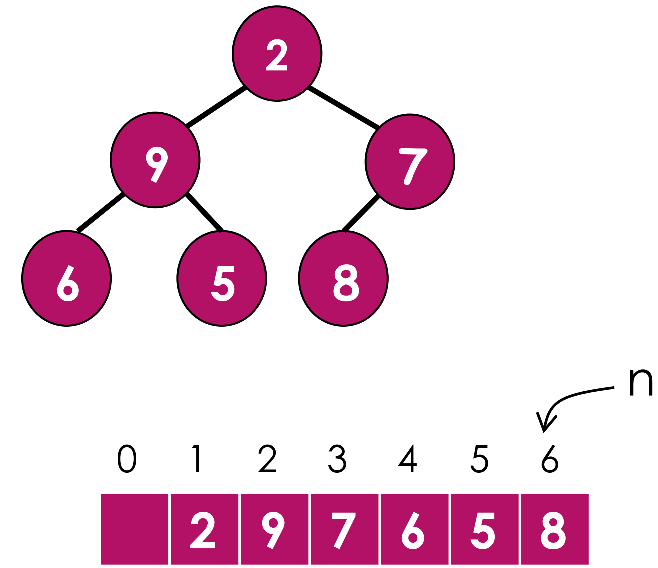
This process is called "Heapifying"

This method is called "bottom-up heap construction"

Heap Transformation Algorithm

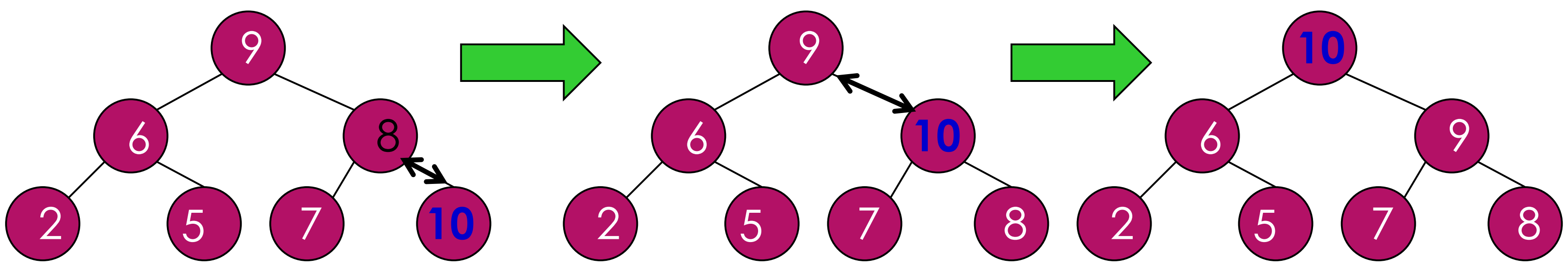

```
ALGORITHM HeapBottomUp(H[1..n])
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i <- ⌊n/2⌋ downto 1 do
    k <- i;
    v <- H[k]
    heap <- false
    while not heap and 2*k ≤ n do
        j <- 2*k
        if j < n          // there are 2 children
            if H[j] < H[j+1]
                j <- j+1
        if v ≥ H[j]       // only left child
            heap <- true
        else
            H[k] <- H[j]
            k <- j
    H[k] <- v
```

Heapify one parent



How to Transform Array into A Heap 2

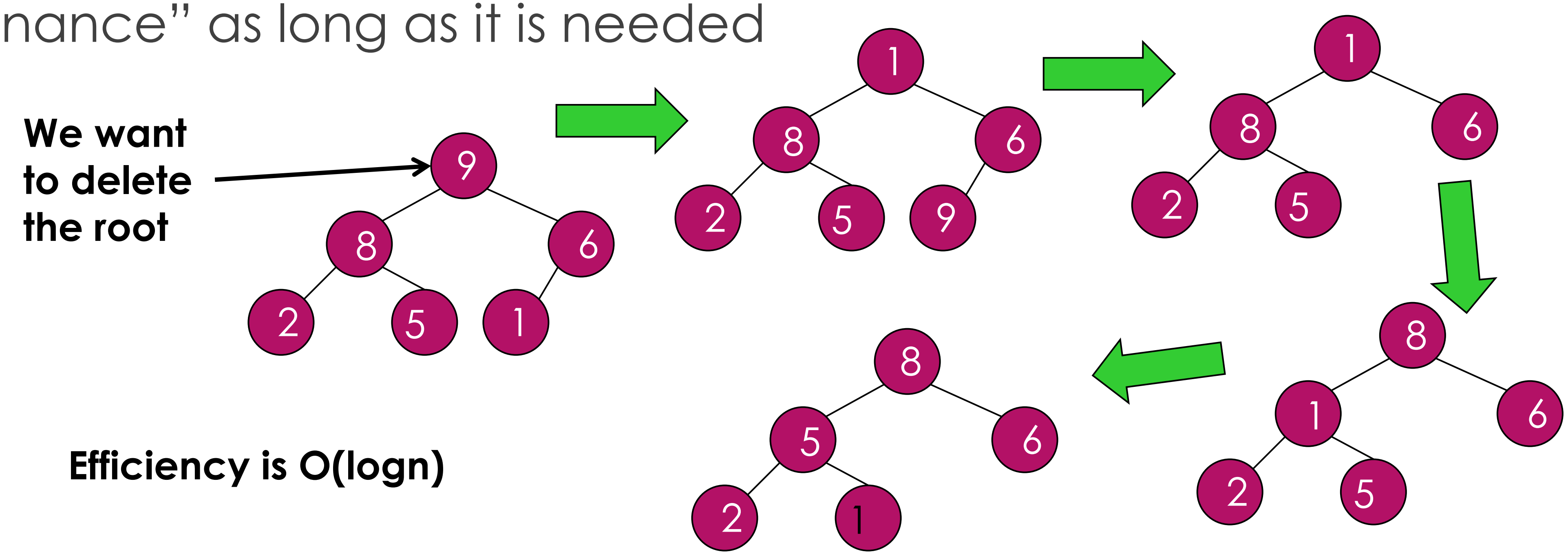
- It's called "top-down heap construction"
- Idea: Successively insert a new key into a previously constructed heap
- Attach a new node with key K after the last leaf of the existing heap
- Sift K up until the parental dominance is established



Insertion operation cannot take more than twice the height of the heap's tree, so it is in O(logn)

How to deletion Maximum key from a Heap

- Exchange root’s key with the last key K in the heap
- Decrease the heap’s size by 1
- “Heapify” the smaller tree by sifting K down establishing “parental dominance” as long as it is needed



Efficiency is O(logn)

Outline

- What is Transform and Conquer
- Instance Simplification: Presorting
- Representation change: Heap and Heapsort
- Problem Reduction: Linear Programming

Transfer and Conquer: Heapsort

- J. W. J. Williams discovered an interesting algorithm in 1964
- It has 2 stages
  - Stage 1: Construct a heap from a given array
  - Stage 2: Apply the root-deletion operation n-1 times to the remaining heap

J. W. J. Williams Algorithm for Heap Construction

Stage 1: Heap construction

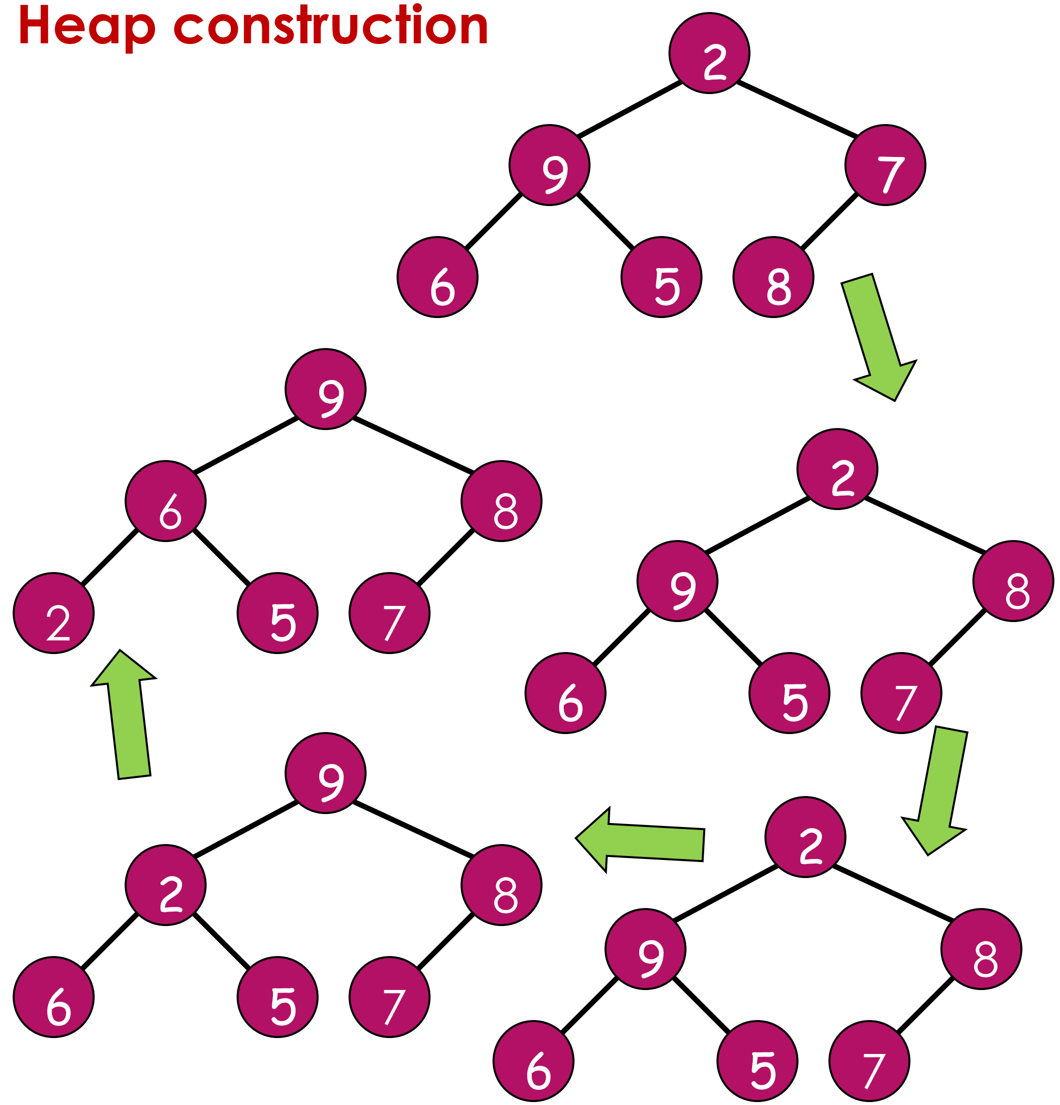
0	1	2	3	4	5	6
	2	9	7	6	5	8

0	1	2	3	4	5	6
	2	9	8	6	5	7

0	1	2	3	4	5	6
	2	9	8	6	5	7

0	1	2	3	4	5	6
	9	2	8	6	5	7

0	1	2	3	4	5	6
	9	6	8	2	5	7



J. W. J. Williams Algorithm for Heap Construction

Stage 2: Maximum deletions

0	1	2	3	4	5	6
	9	6	8	2	5	7

0	1	2	3	4	5	6
	7	6	8	2	5	9

0	1	2	3	4	5	6
	8	6	7	2	5	9

0	1	2	3	4	5	6
	5	6	7	2	8	9

0	1	2	3	4	5	6
	7	6	5	2	8	9

0	1	2	3	4	5	6
	2	6	5	7	8	9

0	1	2	3	4	5	6
	6	2	5	7	8	9

0	1	2	3	4	5	6
	5	2	6	7	8	9

0	1	2	3	4	5	6
	2	5	6	7	8	9

Complexity Analysis of Heapsort

- What is Heapsort's worst-case complexity ?
- **Stage 1: Heap construction is in? O(n)**
- **Stage 2: Maximum deletions is in?**
  - **Let C(n) be the # of comparisons needed in Stage 2**
    - **Recall, height of heap's tree is $\lfloor logn \rfloor$**
    - **C(n) ≤ 2$\lfloor \log(n-1) \rfloor$+ 2$\lfloor log(n-2) \rfloor$ + … … + 2$\lfloor log(1) \rfloor$ ≤ 2$\sum_{i=1}^{n-1} \log(i)$**
    - **C(n) ≤ 2 $\sum_{i=1}^{n-1} log(n-1)$ = 2(n-1)log(n-1) ≤ 2nlogn**
    - **C(n) ϵ O(nlogn)**
    - **for two stages, O(n) + O(nlgn) = O(nlgn)**

Complexity Analysis of Heapsort

- More careful analysis shows that Heapsort's worst and best cases are in $\Theta(n\log n)$ like Mergesort
- Additionally Heapsort is in-place
- Timing experiments on random files show that Heapsort is slower than Quicksort, but can be competitive with Mergesort

Outline

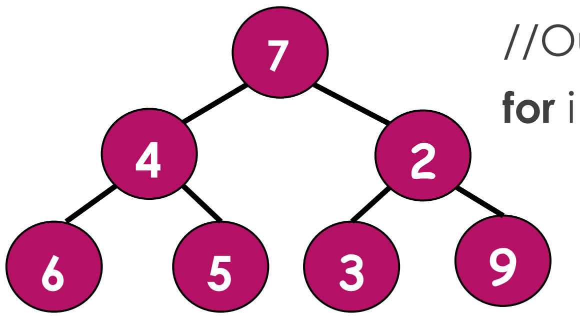
- What is Transform and Conquer
- Instance Simplification: Presorting
- Representation Change: Heap and Heapsort

Summary

- There are three principal varieties of the transform-and-conquer strategy: *instance* ***simplification****, representation* ***change****, and problem* ***reduction***
- *Instance* ***simplification*** *is transforming an instance of a problem to an* ***instance of the same problem*** with **some special property** that makes the problem easier to solve. Example list presorting
- *Representation* ***change*** *implies changing one representation of a problem's* instance to **another** representation of the same instance. Example Heap and Heapsort
- *Problem* ***reduction*** *calls for* ***transforming a given problem to another*** *problem* that can be solved by a **known algorithm**. Example linear programming

Thank You

Heap Transformation Algorithm




```
ALGORITHM HeapBottomUp(H[1..n])
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i <- ⌊n/2⌋ downto 1 do
    k <- i;
    v <- H[k]
    heap <- false
    while not heap and 2*k ≤ n do
        j <- 2*k
        if j < n          // there are 2 children
            if H[j] < H[j+1]
                j <- j+1
        if v ≥ H[j]       // only left child
            heap <- true
        else
            H[k] <- H[j]
            k <- j
    H[k] <- v
```

Heapify one parent

$$\sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2$$

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{j=1}^{2^i} 2(h-i)$$

$$C_{worst}(n) = \sum_{i=0}^{h-1} 2(h-i)2^i$$

h = $\lfloor lg\, n \rfloor$
= $\lceil \lg(n+1) \rceil$-1
= m-1

$$C_{worst}(n) = 2h\sum_{i=0}^{h-1} 2^i - 2i\sum_{i=0}^{h-1} i2^i$$

$= 2h(2^h - 1) - 2(h - 2)2^h + 4$

$= h2^{h+1} - 2h - h2^{h+1} + 2*2^{h+1} + 4$

$= 2\ (\ n - \log(n+1) + 4\ )$