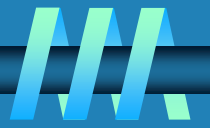
CS 395 – Analysis of Algorithms

Chapter 6 – Transform-and-Conquer

Read 6.4 – 6.6

- Heaps and Heapsort

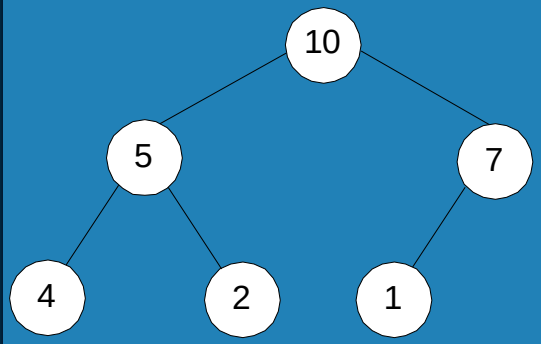
Heaps



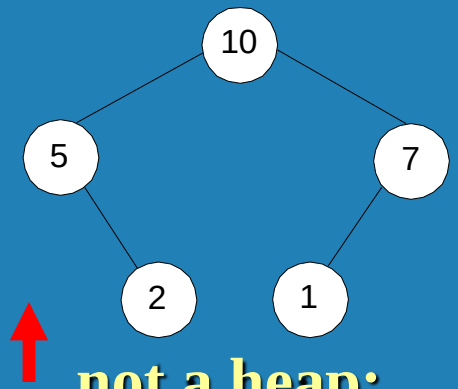
Definition A *heap* is a binary tree with keys at its nodes (one key per node) such that:

- Shape property – the binary tree is *essentially complete*, i.e., all its levels are full except possibly the last level, where only some rightmost keys may be missing.
- *parental dominance* or *heap property* — the key in each node is greater than or equal to the keys in its children. (This condition is considered automatically satisfied for all leaves.)

Illustration of the heap's definition

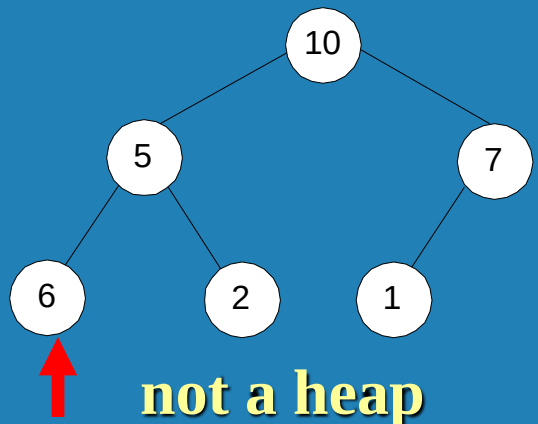


a heap



not a heap:

Left child missing

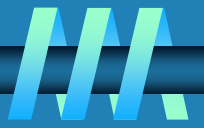


not a heap

6>5

Note: Heap's elements are ordered top down (along any path down from its root), but they are not ordered left to right

Some Important Properties of a Heap

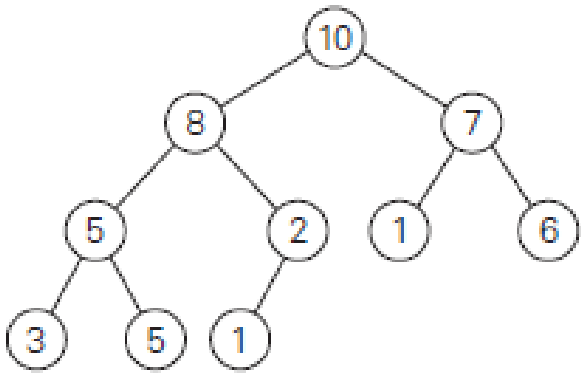


- Given n, there exists a **unique** binary tree with n nodes that is essentially complete, with $h = \lfloor \log_2 n \rfloor$
- The root contains the largest key
- The subtree rooted at any node of a heap is also a heap
- A heap can be represented as an array

Heap's Array Representation

Store heap's elements in an array (whose elements indexed, for convenience, 1 to *n*) in top-down left-to-right order

Example:

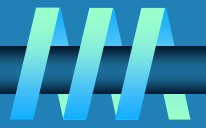


the array representation

index	0	1	2	3	4	5	6	7	8	9	10
value		10	8	7	5	2	1	6	3	5	1
		parents					leaves				

- **Left child of node *i* is at 2*i***
- **Right child of node *i* is at 2*i* +1**
- **Parent of node *i* is at $\lfloor i/2 \rfloor$**
- **Parental nodes are represented in the first $\lfloor n/2 \rfloor$ locations**

Heap Construction (bottom-up)



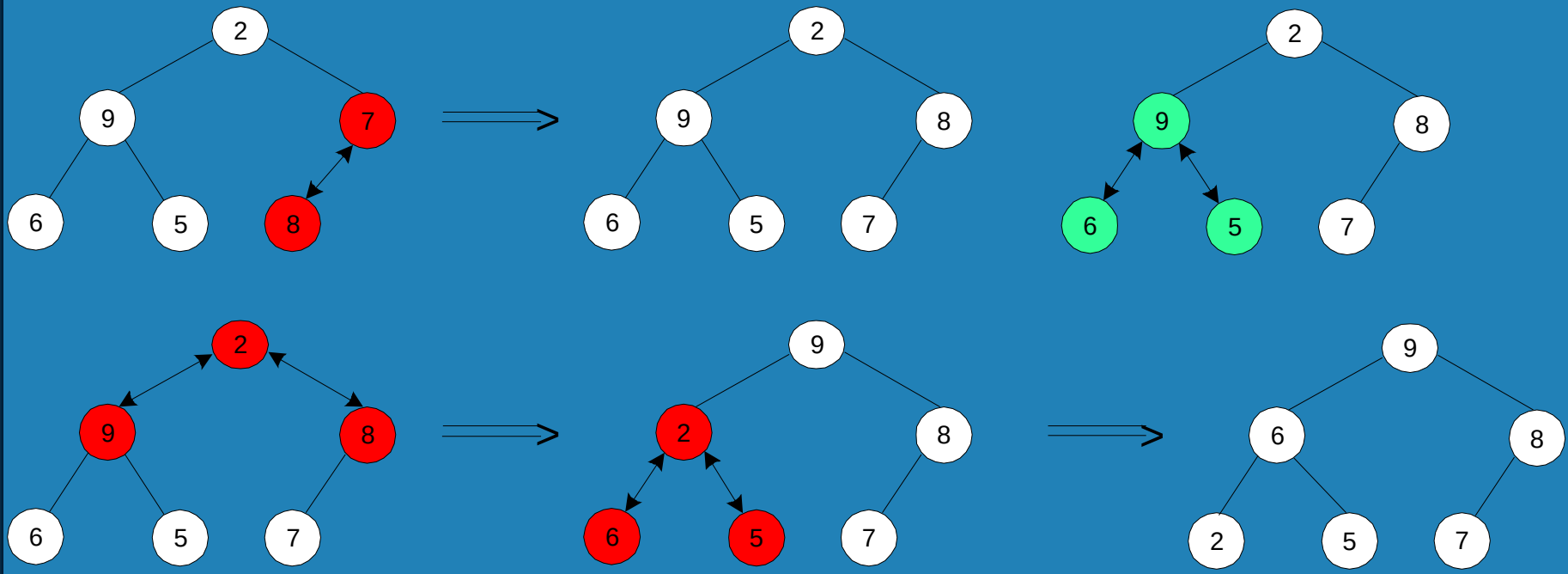
Step 0: Initialize the structure with keys in the order given

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



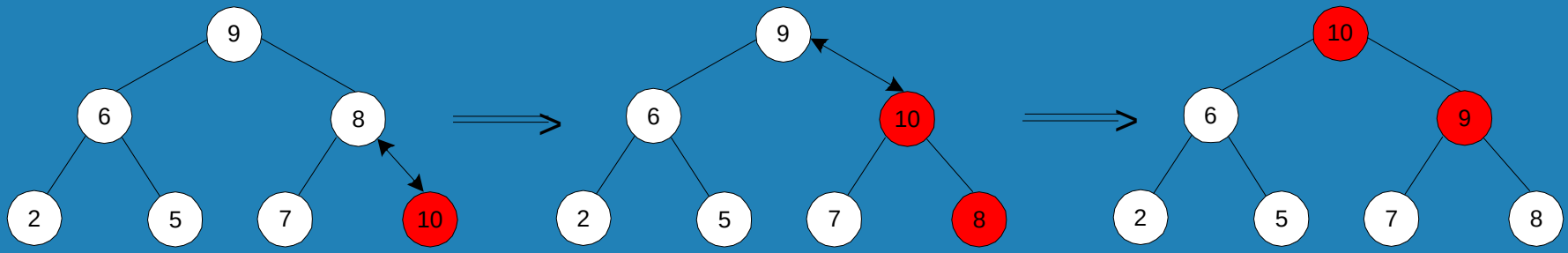
Pseudopodia of bottom-up heap construction

Algorithm *HeapBottomUp*$(H[1..n])$
//Constructs a heap from the elements of a given array
// by the bottom-up algorithm
//Input: An array $H[1..n]$ of orderable items
//Output: A heap $H[1..n]$
for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do**
    $k \leftarrow i; \quad v \leftarrow H[k]$
    $heap \leftarrow$ **false**
    while not $heap$ **and** $2 * k \leq n$ **do**
        $j \leftarrow 2 * k$
        if $j < n$ //there are two children
            if $H[j] < H[j+1] \quad j \leftarrow j+1$
        if $v \geq H[j]$
            $heap \leftarrow$ **true**
        else $H[k] \leftarrow H[j]; \quad k \leftarrow j$
    $H[k] \leftarrow v$

Insertion of a New Element into a Heap

- Insert the new element at last position in heap.
- Compare it with its parent and, if it violates heap condition, exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: Insert key 10



Efficiency: O(log *n*)

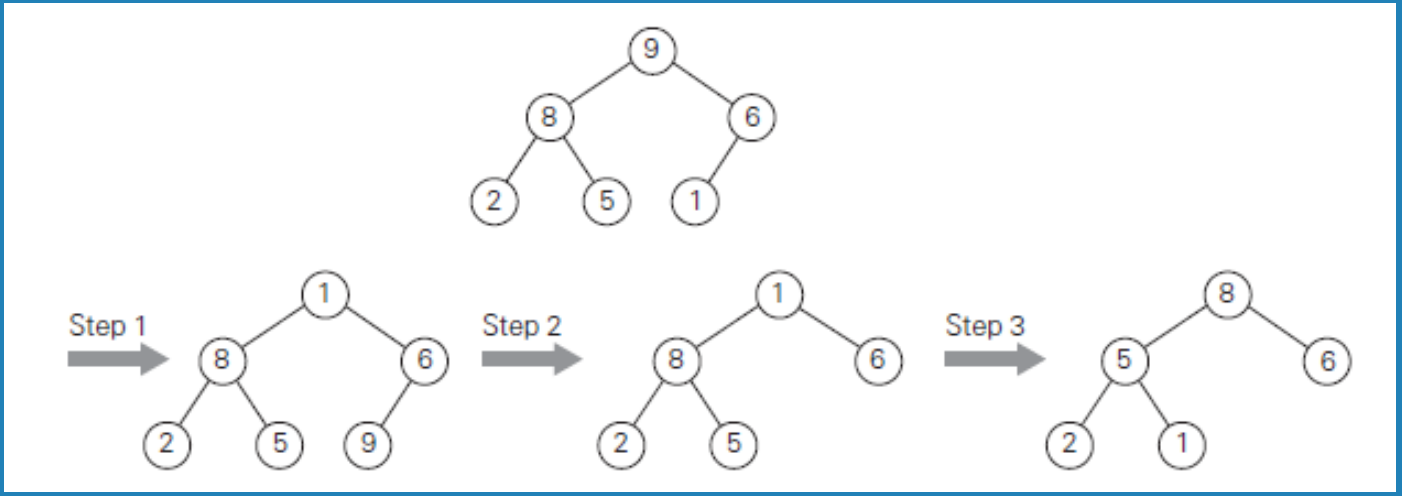
Maximum Key Deletion from a heap

Step 1 Exchange the root's key with the last key K of the heap.

Step 2 Decrease the heap's size by 1.

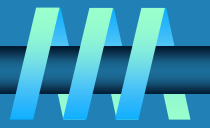
Step 3 "Heapify" the smaller tree by sifting K down the tree using the bottom-up heap construction algorithm.
That is, verify the parental dominance for K: if it holds, we are done; if not, swap K with the larger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Example:



Efficiency: O(log *n*)

Heapsort

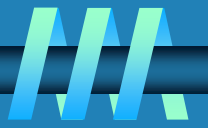


Stage 1: Construct a heap for a given list of *n* keys

Stage 2: Repeat operation of root removal *n*-1 times:

- **Exchange keys in the root and in the last (rightmost) leaf**
- **Decrease heap size by 1**
- **If necessary, swap new root with larger child until the heap condition holds**

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 (root/max removal)

9 6 8 2 5 7

7 6 8 2 5 | 9

8 6 7 2 5 | 9

5 6 7 2 | 8 9

7 6 5 2 | 8 9

2 6 5 | 7 8 9

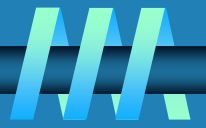
6 2 5 | 7 8 9

5 2 | 6 7 8 9

5 2 | 6 7 8 9

2 | 5 6 7 8 9

Analysis of Heapsort



Stage 1: Build heap for a given list of *n* keys

worst-case

$$C(n) = \sum_{i=0}^{h-1} 2(h-i)\, 2^i = 2\,(n - \log_2(n+1)) \in \Theta(n)$$

nodes at level *i*

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

worst-case

$$C(n) = \sum_{i=1}^{n-1} 2\log_2 i \in \Theta(n \log n)$$

Both worst-case and average-case efficiency: $\Theta(n \log n)$
In-place : yes, meaning space efficiency is $\Theta(1)$
Stability: no