Lesson 5: Heaps

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS | SPRING 2022

DR. ANDREY TIMOFEYEV

OUTLINE

- Introduction.
- Heap structure.
- Building a heap.
- Heapsort.

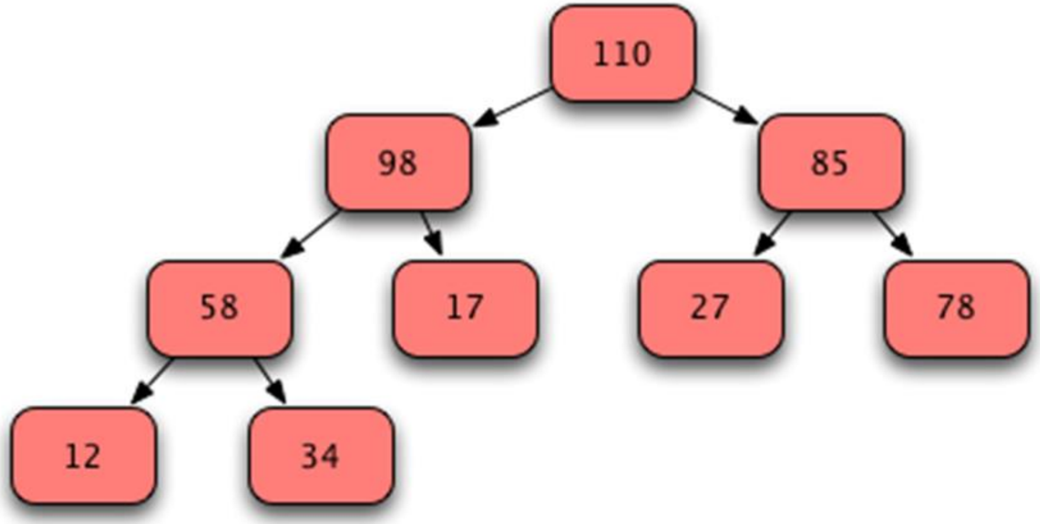
INTRODUCTION

- Heap – **complete** (**semi-**)**ordered tree** that satisfies **heap property**:
  - **Max** (**largest-on-top**) **heap** – every node is >= all its children (if any).
    - For any given node C, if P is a parent node of C, then the value of P is >= to the value of C.
  - **Min** (**smallest-on-top**) **heap** – every node is <= all its children (if any).
    - For any given node C, if P is a parent node of C, then the value of P is <= to the value of C.
- In heap **highest**/**lowest** element is always at the **top** (**root**).
- **<u>Conceptually</u>**:
  - **Heap** - tree that is **full on all levels** (except possibly the lowest level) and filled in from **left to right.**
- **Heap use cases:**
  - Repeatedly **removing** object with the **highest**/**lowest** value (priority).
  - **Insertions** are combined with **removals** of the root node.

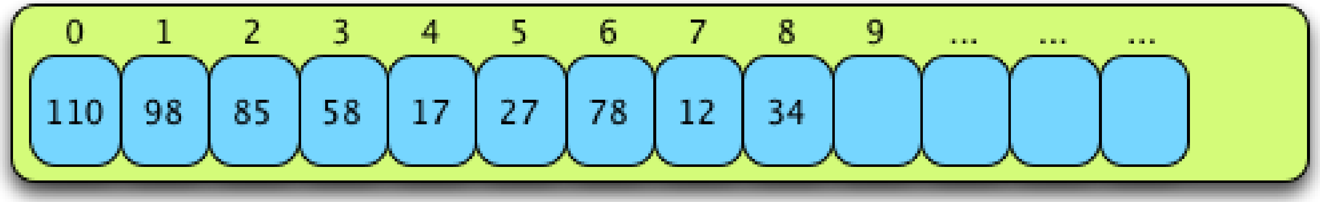
Complete tree

HEAP STRUCTURE

- **Heap** - **complete tree** -> convenient to store in an **array**.
- When heap stored in an array, **indexes** of **parent** & **children** nodes are computed as follows:
  - leftChildIndex = 2 x parentIndex + 1
  - rightChildIndex = 2 x parentIndex + 2
  - parentIndex = (childIndex – 1) // 2
- **Not every** node has two or one child.
  - If left/rightChildIndex >= heap size -> **leaf node**.
- All nodes **except root** node have parents.
- **Height** of heap storing *n* nodes: $\mathbf{h} = \lfloor \mathbf{log n} \rfloor$



Sample heap



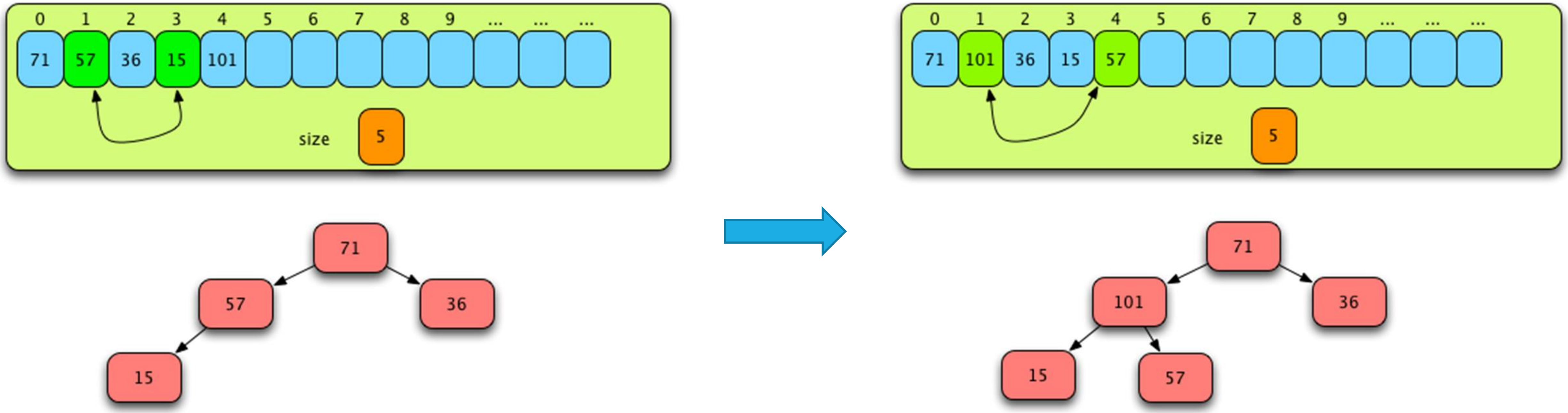
Heap represented as array

BUILDING A HEAP (1)

- Heaps are built **based** on their **type**:
  - **Max** heap - **largest** on top, by **sift-up** process.
  - **Min** heap - **smallest** on top, by **sift-down** process.
- **Building largest-on-top heap:**
  - Sequence of values is added into the heap in provided order.
  - Each subsequent value after root is sifted up into its final location.
  - Sift-up process:
    - Compute index of a parent node and compare values.
    - If value is greater than value at parent node -> swap values.
    - Repeat until root is reached (index 0) or node is in proper location (no swap needed).

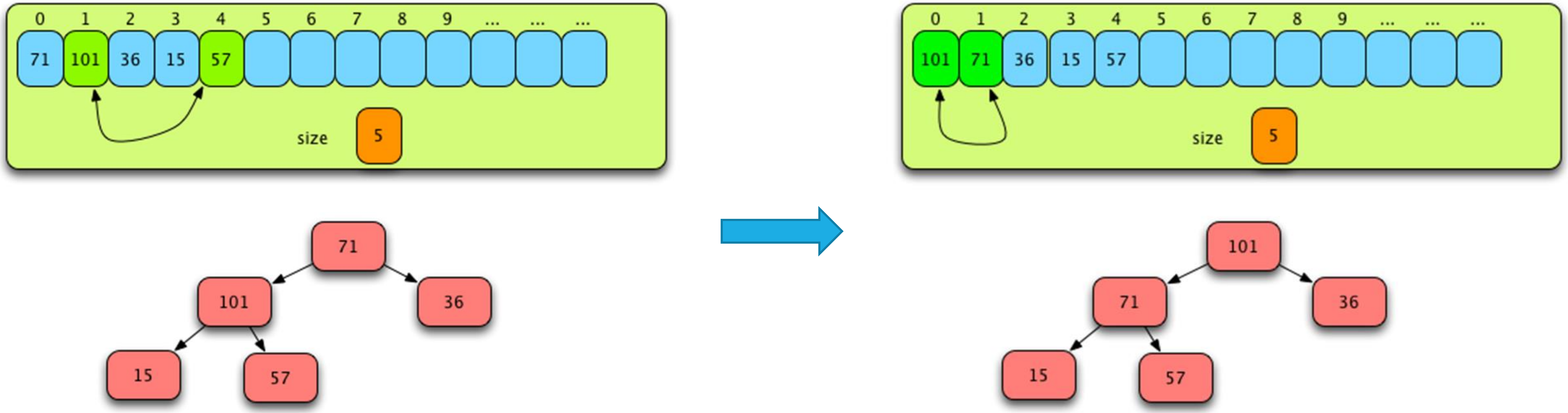
BUILDING A HEAP (2)

- Example of building a heap: [71, 15, 36, 57, 101].



BUILDING A HEAP (3)

- Example of building a heap: [71, 15, 36, 57, 101].



HEAPSORT

- **Heapsort algorithm consists of two phases:**
  - **Phase I.**
    - **Adding** values to the heap & “**heapifying**” them.
  - **Phase II.**
    - **Removing** values from the top (root) of the heap.
    - **“Re-heapifying”** the rest of the values in the heap.

- **Phase II applied on the max (largest on top) heap:**
  - Place **largest** value at the **end** of the heap data list (correct position).
    - **Swap root** with **last value** in the heap.
  - **Decrease size** of the heap by 1.
  - **Root** now **does not comply** with **heap properties** -> **sift** it **down** to the correct position.
    - Keep **swapping** with the **largest child** until it is in the **correct position**, or **no children** left (becomes leaf).

HEAPSORT: EXAMPLE

- **Heapsort example: [10, 30, -100, 50, 20, 30, -40, 70, 5, 50].**
  - After adding values to the heap & “heapifying” in Phase I:
    - Data = [70, 50, 30, 30, 50, -100, -40, 10, 5, 20]
  - Phase II applied on [70, 50, 30, 30, 50, -100, -40, 10, 5, 20]

HEAPSORT COMPLEXITY

- **Heapsort complexity.**
  - **Complexity of Phase I.**
    - Adding items to the heap takes $\mathrm{O(logn)}$ and performed for each element *n*, thus $\mathrm{O(nlogn)}$ time.
  - **Complexity of Phase II.**
    - Sifting root down into correct position after swapping performed $n-1$ times, thus $\mathrm{O(nlogn)}$ time.
  - **Overall** heapsort complexity is $\mathrm{O(nlogn)}$**.**

- **Comparisons to quicksort.**
  - Both **heapsort** & **quicksort** operate in $\mathrm{O(nlogn)}$ time.
    - **Quicksort** – values are always moved toward their final location.
    - **Heapsort** – values are moved to a heap, then moved again to arrive at their final location.
  - **Quicksort >> heapsort** even though they have the same computational complexity.