

# Lecture 5: Heap Sort

**Dr. Khalil Yousef**

**Adopted from the Slides of the ECE 608 Computational Models and Methods Course at Purdue University**

Read Chapter 6 of *Introduction to Algorithms*

# The Sorting Problem

---

## The Sorting Problem (non-decreasing):

**Input:** a sequence of  $n$  numbers  $\langle a_1, a_2, \dots, a_n \rangle$

**Output:** a permutation of the input sequence  $\langle a'_1, a'_2, \dots, a'_n \rangle$  such that  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

## Sorting algorithms:

**Insertion Sort:**  $\Theta(n^2)$

**Merge Sort:**  $\Theta(n \lg n)$

**Heap Sort:**  $\Theta(n \lg n)$  (large constant factor)

**Quick Sort:** It has a worst case running time of  $\Theta(n^2)$ , but an average case running time of  $\Theta(n \lg n)$  (small constant factor).

Linear sort algorithms will also be discussed which are not comparison sorts.

## Some Sorting Terminology

---

A list or array of elements that is to be sorted often consists of elements that are records. Each record is sorted with respect to its **key**, which is some ordinal type. The record will also typically contain **satellite data**. The book does not focus on the satellite data.

**Comparison Sort:** Uses comparisons between elements in the input array (or list) of elements to produce a sorted output.

**In-place Sort:** Uses only a constant amount of storage in addition to the input data structure.

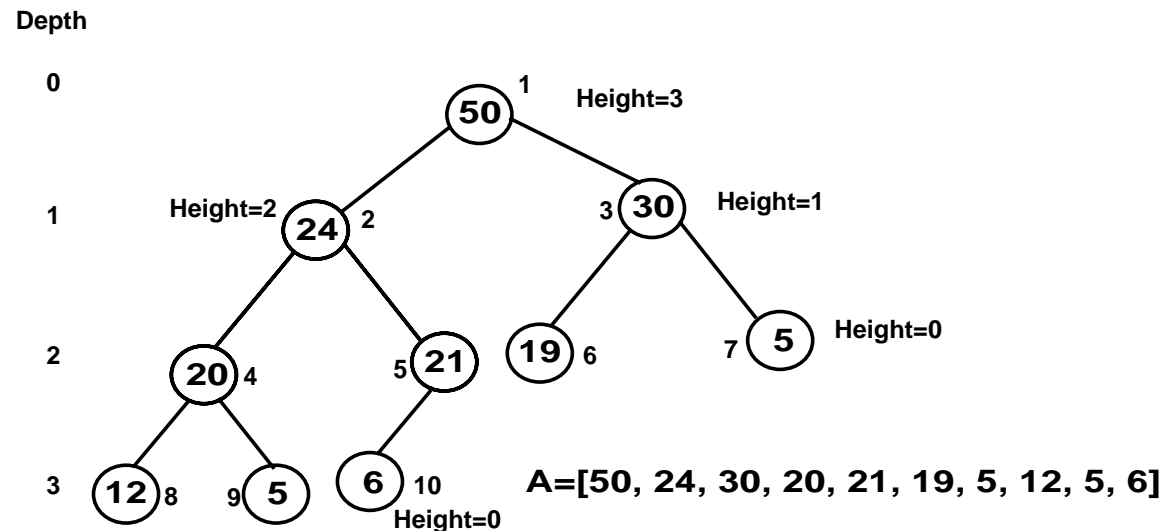
**Stable Sort:** Maintains the same relative ordering of elements with the same key in the sorted output as in the input.

# The Heap Data Structure

---

We first define heaps and the various operations on them.

**Definition:** A (binary) **heap** is a nearly complete binary tree that satisfies the heap property. For example,



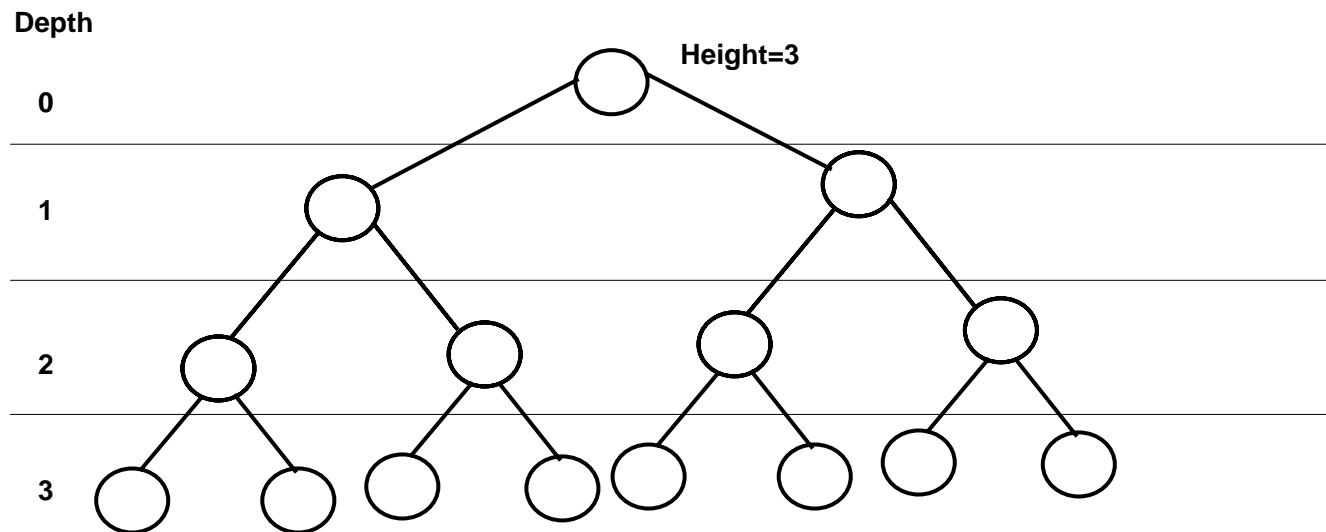
**Definition:** The **heap property** is the key of the parent must be greater than or equal to the key of the child, i.e.,  $A[\text{PARENT}(i)] \geq A[i]$ .  $A[1]$  is the root of the heap.

## Binary Tree Terminology

---

A **binary tree**  $T$  either contains no nodes or is comprised of three disjoint sets:

- the root node,
- the left subtree, which is a binary tree,
- the right subtree, which is a binary tree.



## Binary Tree Terminology *continued*

---

The number of children that a node  $x$  can have in a tree  $T$  is called its **degree**. The nodes in a binary tree can have at most a degree of 2.

The length of the longest path from root  $r$  to a node  $x$  in  $T$  defines the **depth** of  $x$ . Note that a root has depth 0. The **height** of a tree  $T$  is the largest depth of any node in  $T$ . The height of a node in a tree is the length of the longest simple path to a leaf node.

A **complete** binary tree contains only nodes that are leaf nodes or have a degree of 2, and all leaf nodes have the same depth. The number of nodes in a complete

binary tree of height  $h$  is  $\sum_{i=0}^h 2^i = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$ .

Nodes that are leaf nodes are often called **external nodes**. Nodes that have at least one child are called **internal nodes**. The number of internal nodes in a complete

binary tree of height  $h$  is  $\sum_{i=0}^{h-1} 2^i = \frac{2^h-1}{2-1} = 2^h - 1$ .

## The Heap Data Structure *continued*

---

Note that we number the nodes in the heap from top to bottom, left to right. Then we can store the heap in an array,  $A$ , where the number associated with a node is its array index. The array  $A$  has two attributes:

1.  $length[A]$  indicates the number of elements in the array; this stays constant throughout HEAPSORT.
2.  $heap-size[A]$  indicates the number of elements in the heap.  $heap-size[A] \leq length[A]$ . This value will decrease during the iterations in HEAPSORT.

We define three access functions:

1.  $PARENT(i)$  **return**  $\lfloor i/2 \rfloor$
2.  $LEFT(i)$  **return**  $2i$
3.  $RIGHT(i)$  **return**  $2i + 1$

## The Heap Data Structure *continued*

---

### Properties of a heap:

1. The root of a heap is the largest element.
2. Any path from leaf to root has values in ascending order.
3. The assignment of keys to a node in a heap structure is not unique, though the structure (i.e., shape) of the heap with a certain number of elements is unique.
4. All leaves are at depth  $d - 1$  or  $d$ , where  $d$  is the maximum depth.
5. All non-leaf vertices, except for possibly one, have two children. If a non-leaf vertex has only one child (which must be a left child), it is the right-most vertex at that depth with children.



## Heap Operations

---

There are several heap operations that we will discuss:

- HEAPIFY:  $\Theta(\lg n)$
- BUILD-HEAP:  $\Theta(n)$
- HEAPSORT:  $\Theta(n \lg n)$
- HEAP-EXTRACT-MAX:  $\Theta(\lg n)$
- HEAP-INSERT:  $\Theta(\lg n)$

A heap with  $n$  elements has height  $\Theta(\lg n)$ ; hence, many of the basic operations on a heap run in time proportional to  $\Theta(\lg n)$ .

## The HEAPIFY Algorithm

---

HEAPIFY takes two parameters, an array  $A$  and an index  $i$  into the array. It is assumed that the binary trees rooted at  $\text{LEFT}(i)$  and  $\text{RIGHT}(i)$  are heaps, but the element at position  $i$  may be out of place. This routine moves the  $i$ th element into its proper place, so that the subtree rooted at  $i$  becomes a heap.

```
HEAPIFY( $A, i$ )
1.  $l \leftarrow \text{LEFT}(i)$ 
2.  $r \leftarrow \text{RIGHT}(i)$ 
3. if  $l \leq \text{heap-size}[A]$  and  $A[l] > A[i]$ 
4.   then  $largest \leftarrow l$ 
5.   else  $largest \leftarrow i$ 
6. if  $r \leq \text{heap-size}[A]$  and  $A[r] > A[largest]$ 
7.   then  $largest \leftarrow r$ 
8. if  $largest \neq i$ 
9.   then exchange  $A[i] \leftrightarrow A[largest]$ 
10.    HEAPIFY( $A, largest$ )
```

## The HEAPIFY Algorithm

---

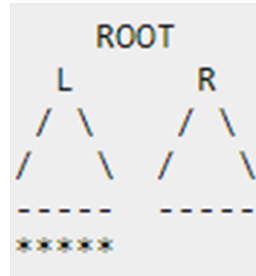
The way HEAPIFY works:

- Compare  $A[i]$ ,  $A[LEFT(i)]$ , and  $A[RIGHT(i)]$ .
- If necessary, swap  $A[i]$  with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at  $i$  is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

## Worst-case Analysis of HEAPIFY

---

**Assertion:** A child of node  $i$  can have a tree of size at most  $\frac{2n}{3}$ .



For a complete binary tree of height  $h$ , the number of nodes is  $f(h) = 2^{h+1} - 1$ . In above case we have nearly complete binary tree with the bottom half full. We can visualize this as collection of ROOT + LEFT complete tree (L) + RIGHT complete tree (R). If height of original tree is  $h$ , then the height of left is  $h - 1$  and right is  $h - 2$ . So the total number of nodes in the tree (without the \* nodes) can be expressed using the following equations:

$$n = 1 + f(h - 1) + f(h - 2) \dots (1)$$

We want to solve above for  $f(h - 1)$  expressed as in terms of  $n$

$$f(h - 2) = 2^{(h-1)} - 1 = (2^h - 1 + 1)/2 - 1 = (f(h - 1) - 1)/2 \dots (2)$$

Using above in (1) we have

$$n = 1 + f(h - 1) + (f(h - 1) - 1)/2 = 1/2 + 3 * f(h - 1)/2$$

$$\Rightarrow f(h - 1) = 2 * (n - 1/2)/3 \quad \Rightarrow \text{Hence } O(2n/3)$$

## Worst-case Analysis of HEAPIFY *continued*

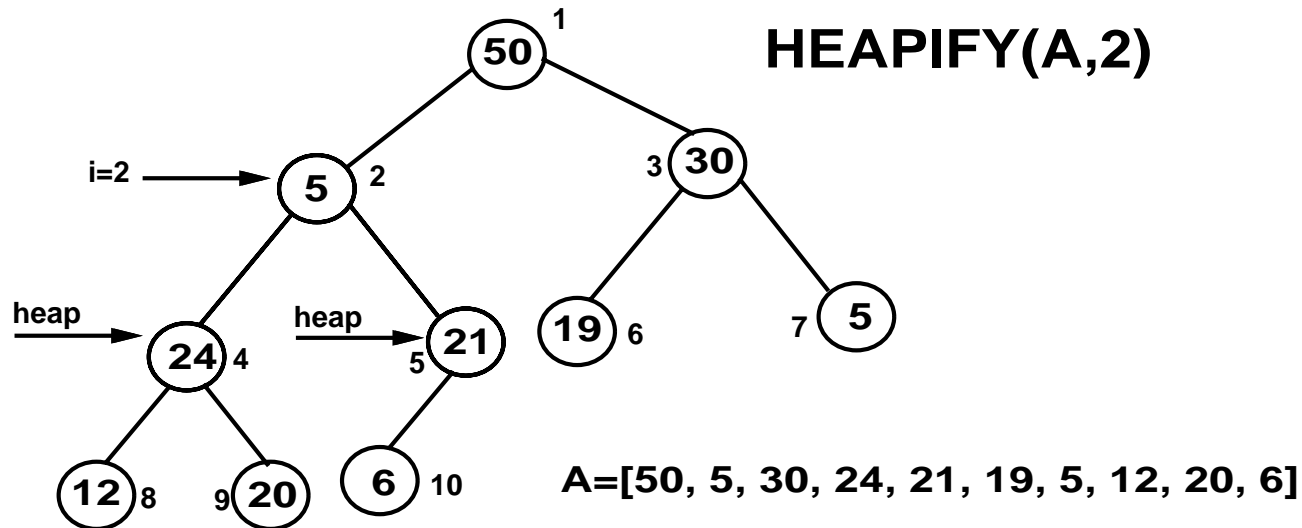
**Complexity:** Because the child of node  $i$  can have a tree of size at most  $\frac{2n}{3}$ ,

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

By case 2 of the Master Theorem,  $a = 1$ ,  $b = \frac{3}{2}$ , so  $n^{\log_b a} = \Theta(1)$ , and  $f(n) = \Theta(1)$ , so:

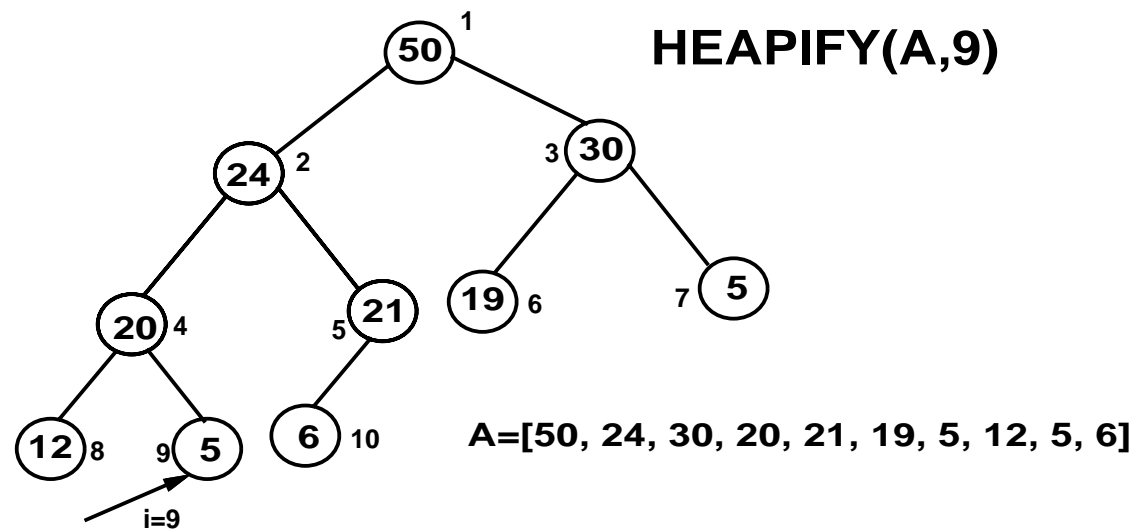
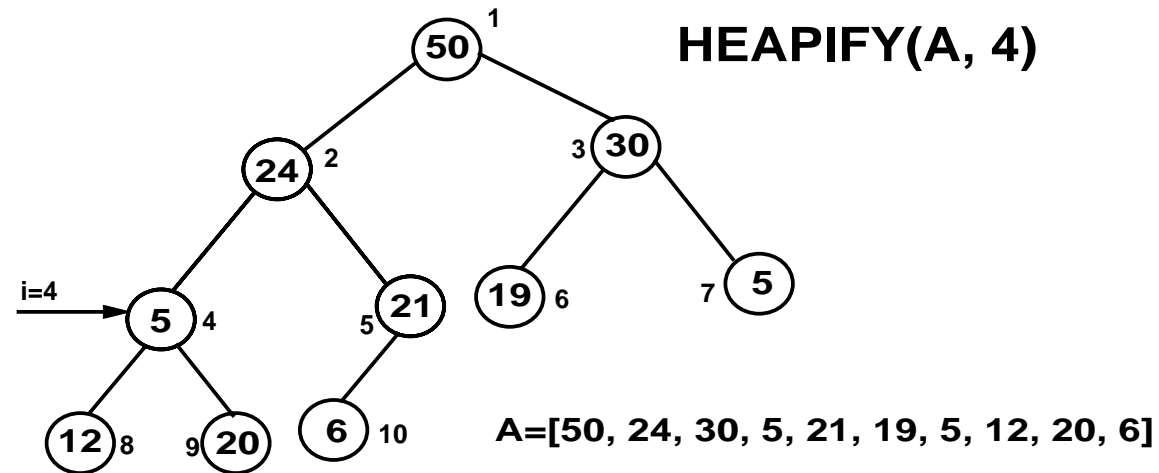
$T(n) = O(\lg n) = O(h)$ , where  $h$  is the height of the heap.

**Example:** Run HEAPIFY( $A, 2$ ) on the following:



## The HEAPIFY Algorithm *continued*

---



## Building a Heap

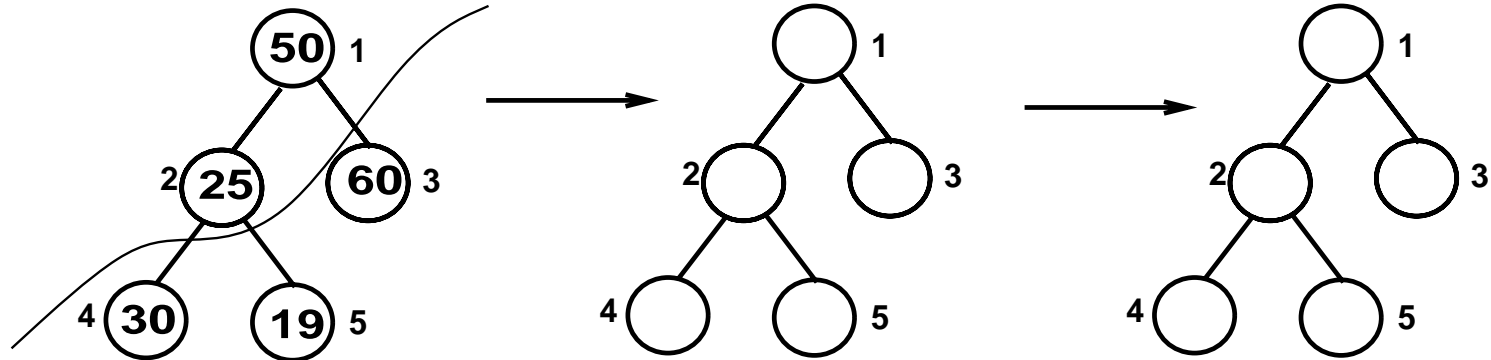
We can use HEAPIFY in a bottom-up way to convert  $A[1..n]$ ,  $n = \text{length}[A]$  into a heap. Since  $A[(\lfloor \frac{n}{2} \rfloor + 1)..n]$  are leaves, they are already heaps; hence, BUILD-HEAP works bottom-up with the remaining nodes of  $A$ .

BUILD-HEAP( $A$ )

1.  $\text{heap-size}[A] \leftarrow \text{length}[A]$
2. **for**  $i \leftarrow \lfloor \frac{\text{length}[A]}{2} \rfloor$  **downto** 1
3.     **do** HEAPIFY( $A, i$ )

**Example:**

**$A = [50, 25, 60, 30, 19]$**



## Building a Heap *continued*

---

### Complexity:

A simple upper bound for BUILD-HEAP can be determined: each call to HEAPIFY is  $O(\lg n)$  and there can be  $n$  such calls, giving an  $O(n \lg n)$  worst-case running time. However, this is not asymptotically tight.

The time to HEAPIFY a node at height  $h$  is  $O(h)$ , so we can express the cost of BUILD-HEAP with the following summation (since the number of nodes at height  $h$  in an  $n$  node heap is at most  $\lceil \frac{n}{2^{h+1}} \rceil$ ):

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})$$

Because  $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ ,  $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$ . Hence,

$$T(n) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(n \sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$



# Heapsort

---

HEAPSORT( $A$ )

1. BUILD-HEAP( $A$ )

2. **for**  $i \leftarrow \text{length}[A]$  **downto** 2

3.     **do** exchange  $A[1] \leftrightarrow A[i]$

4.          $\text{heap-size}[A] = \text{heap-size}[A] - 1$

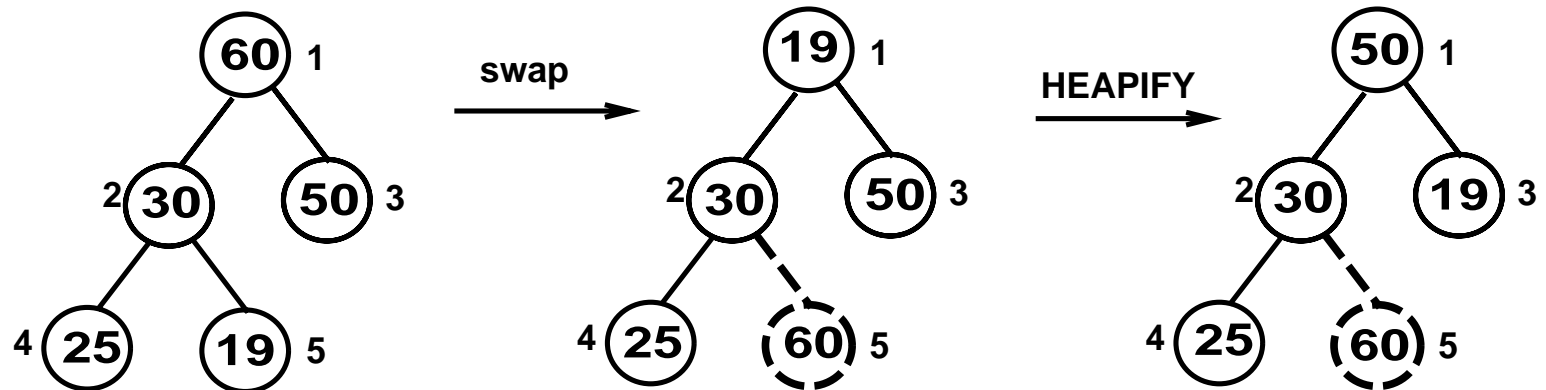
5.         HEAPIFY( $A, 1$ )

## Complexity:

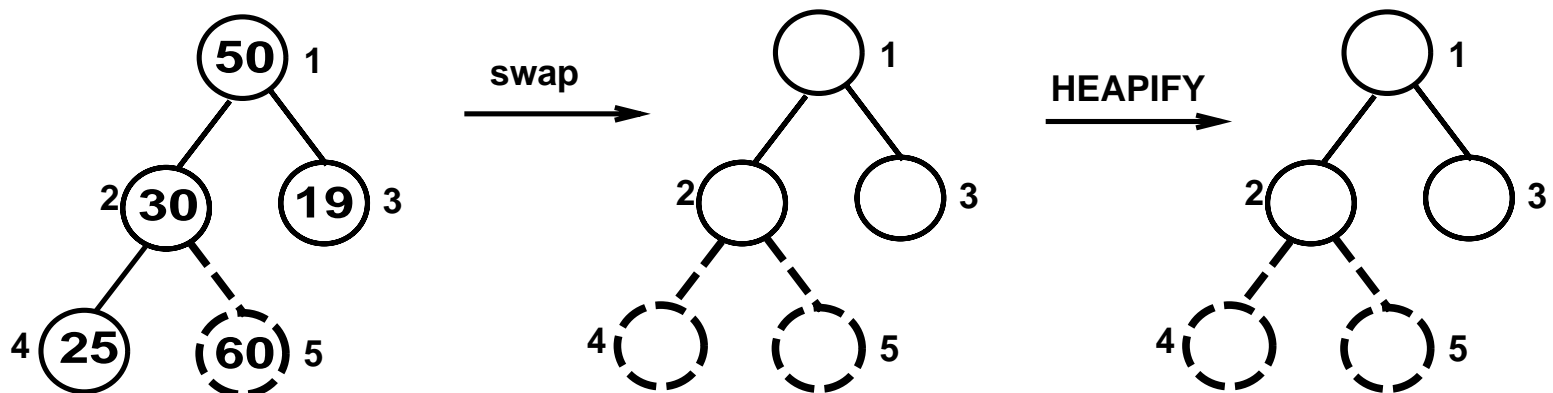
The  $O(n)$  time to do BUILD-HEAP together with  $n$  calls to HEAPIFY gives HEAPSORT a worst-case running time of  $O(n \lg n)$ .

## Heapsort Example

**A=[60, 30, 50, 25, 19]**



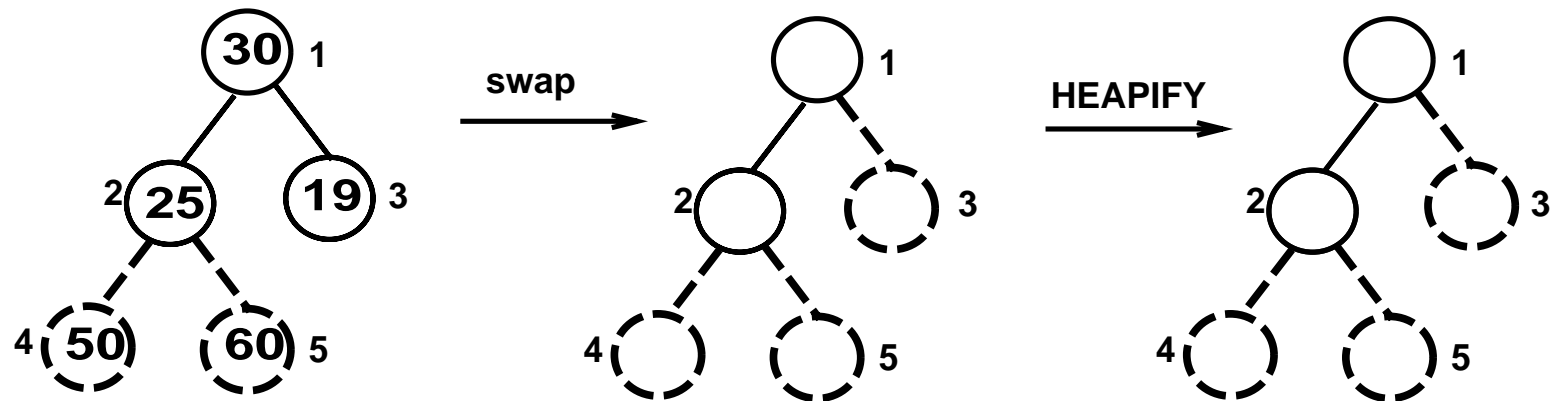
**A=[50, 30, 19, 25, 60]**



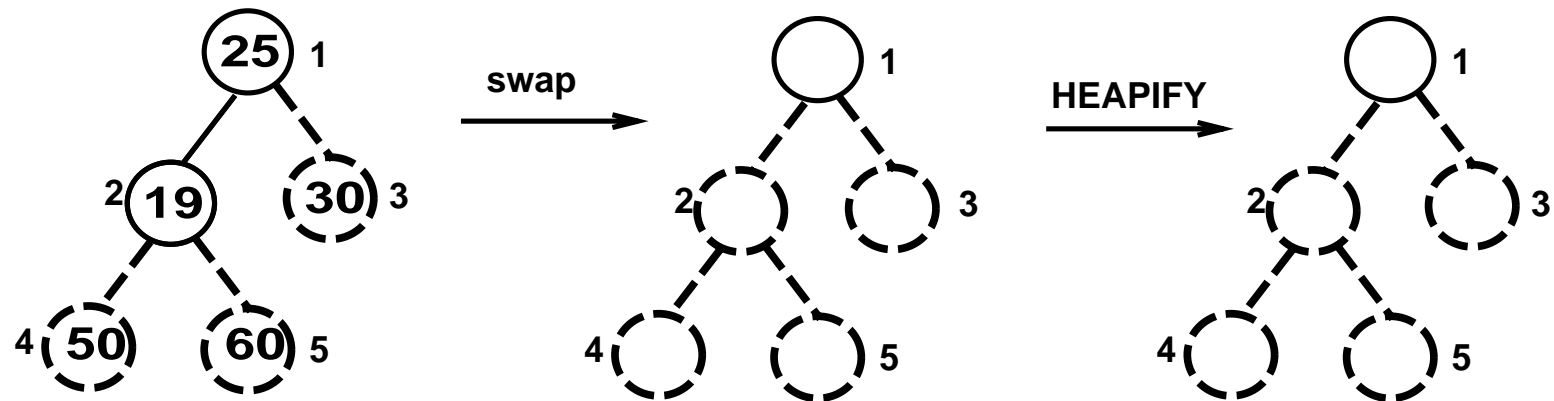
## Heapsort Example *continued*

---

**A=[30, 25, 19, 50, 60]**



**A=[ 25, 19, 30, 50, 60]**



## Priority Queues

---

A **priority queue** is a data structure for maintaining a set of elements,  $S$ , which supports the following operations:

- $\text{INSERT}(S, x)$ : insert element  $x$  into the set  $S$ .
- $\text{MAXIMUM}(S)$ : returns element with the largest key from the set  $S$ .
- $\text{EXTRACT-MAX}(S)$ : removes and returns the element with the largest key from the set  $S$ .
- $\text{INCREASE-KEY}(S, x, k)$ : increases the value of element  $x$ 's key to the new value  $k$ , which is assumed to be at least as large as  $x$ 's current key value.

For example, a priority queue can be used in scheduling jobs with priorities on a computer system.

A heap is a good data structure for implementing a priority queue. We will examine the routines  $\text{HEAP-EXTRACT-MAX}$ ,  $\text{HEAP-INSERT}$  and  $\text{HEAP-INCREASE-KEY}$ . The worst-case running time of these algorithms is related to the height of the heap,  $O(\lg n)$ .

## The way HEAP-INSERT Algorithm work

---

Say that we want to insert element  $x$  into a heap with  $(n - 1)$  elements .

- Add  $x$  at the highest number level leaf.
- Increase heap size.
- Restore Heap property.
  - Repeat:
    - Compare  $x$  to its parent
    - If  $x$  is larger than its parent, then swap  $x$  with its parent until no swap is needed or  $x$  is at the root.

## The way HEAP-EXTRACT-MAX Algorithm work

---

We want to delete the max element from the  $(n)$  elements heap .

- Delete the root (We need to restore Heap property).
- Decrease heap size.
- Put the right most element in the last level of the heap tree in the root position.
- Restore Heap property.

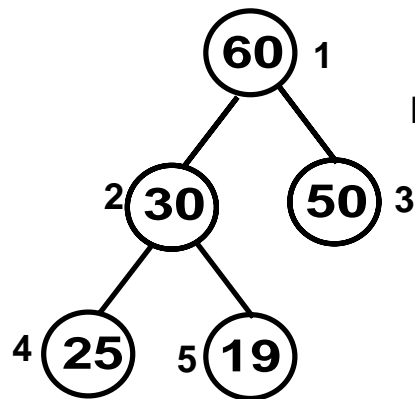
## Priority Queues *continued*

---

HEAP-EXTRACT-MAX( $A$ )

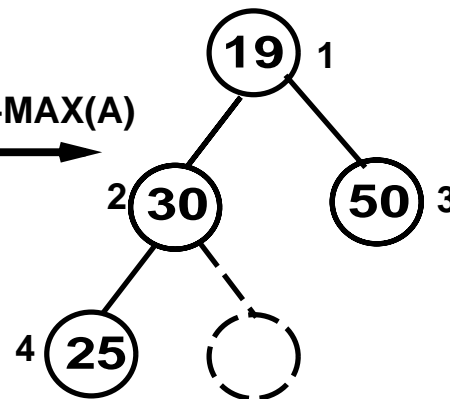
1. **if**  $\text{heap-size}[A] < 1$
2.   **then error** "heap underflow"
3.  $\text{max} \leftarrow A[1]$
4.  $A[1] \leftarrow A[\text{heap-size}[A]]$
5.  $\text{heap-size}[A] = \text{heap-size}[A] - 1$
6. HEAPIFY( $A, 1$ )
7. **return**  $\text{max}$

$A = [60, 30, 50, 25, 19]$

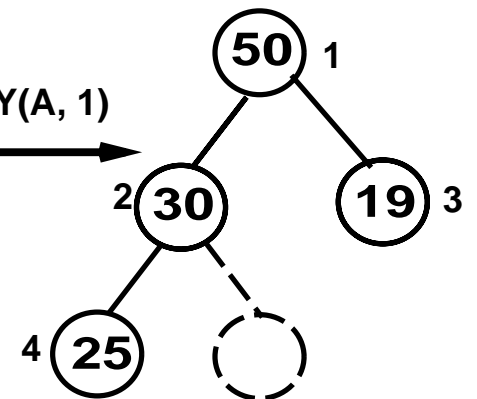


HEAP-EXTRACT-MAX( $A$ )

return 60



HEAPIFY( $A, 1$ )



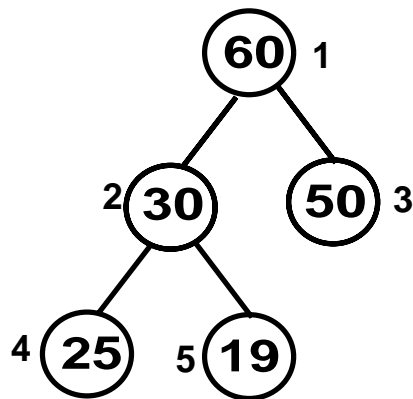
## Priority Queues *continued*

---

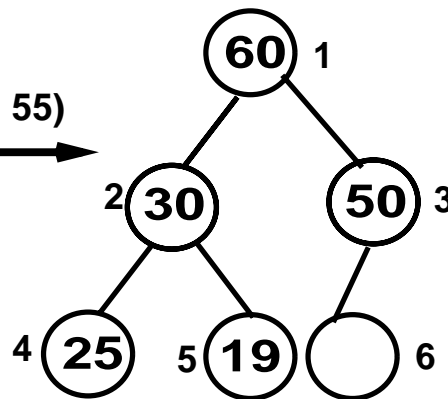
HEAP-INSERT( $A, key$ )

1.  $heap-size[A] = heap-size[A] + 1$
2.  $i \leftarrow heap-size[A]$
3. **while**  $i > 1$  and  $A[PARENT(i)] < key$
4.     **do**  $A[i] \leftarrow A[PARENT(i)]$
5.      $i \leftarrow [PARENT(i)]$
6.  $A[i] \leftarrow key$

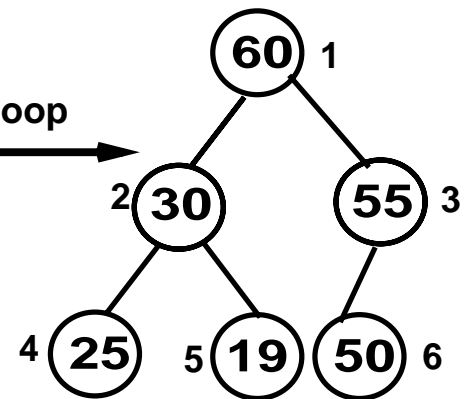
$A=[60, 30, 50, 25, 19]$



HEAP-INSERT( $A, 55$ )



While loop





## Priority Queues *continued*

---

HEAP-INCREASE-KEY ( $A, i, key$ )

1. **if**  $key < A[i]$
2.     **then error** / "new key is smaller than current key"
3.  $A[i] \leftarrow key$
4. **while**  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$
5.     exchange  $A[i]$  with  $A[\text{PARENT}(i)]$
6.      $i \leftarrow [\text{PARENT}(i)]$

**Exercise:** Modify the above routines to design a min-priority queue that supports the following operations INSERT, MINIMUM, EXTRACT-MIN, and DECREASE-KEY.