



# CS 590: Algorithms

## Lecture 5 – Sorting and Order Statistics I

# Outline



## 5.1. Heapsort

### 5.1.1. Tree & Binary Tree

### 5.1.2. Heap

### 5.1.3. Heapsort

## 5.2. Quick Sort

## 5.3. Counting Sort

## 5.4. Radix Sort

## 5.5. Bucket Sort

## 5.6. Conclusion



# 5.1. Heapsort

## 5.1. Heapsort

### 5.1.1. Tree & Binary Tree

### 5.1.2. Heap

### 5.1.3. Heapsort

## 5.2. Quick Sort

## 5.3. Counting Sort

## 5.4. Radix Sort

## 5.5. Bucket Sort

## 5.6. Conclusion

## 5.1.1. Tree and Binary Tree

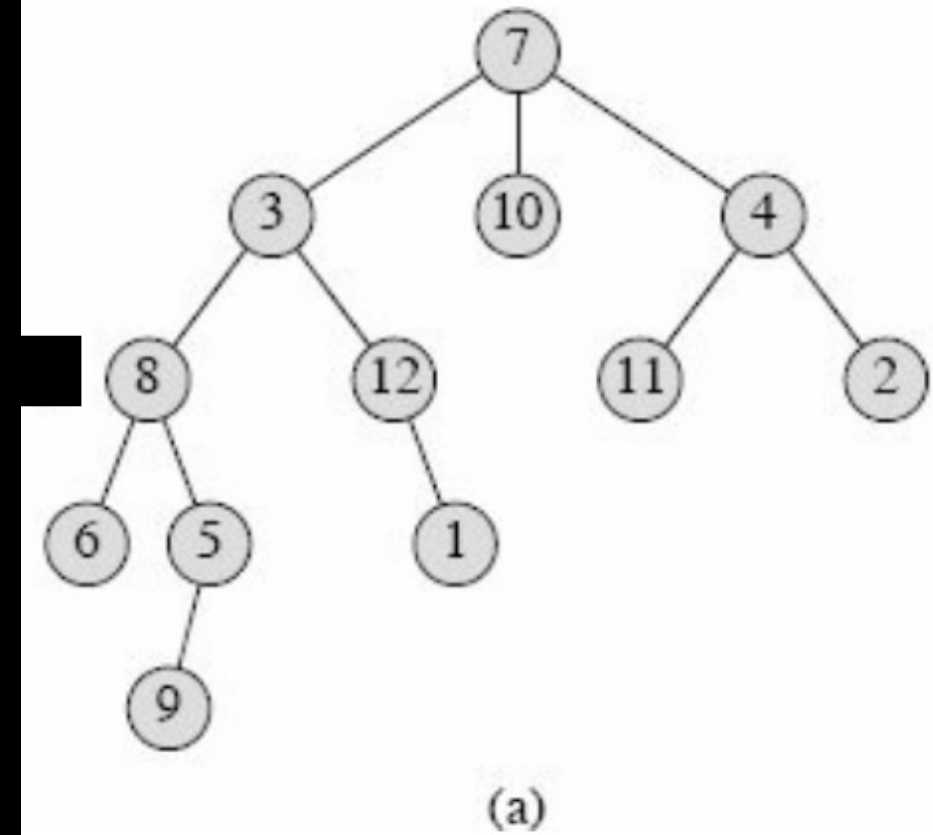


### Tree:

- Tree is a graphical way of algorithm.

### Rooted tree:

a tree in which one of the vertices is distinguished from the others. The distinguished vertex is called the **root** of the tree. A vertex of a rooted tree as a **node**.

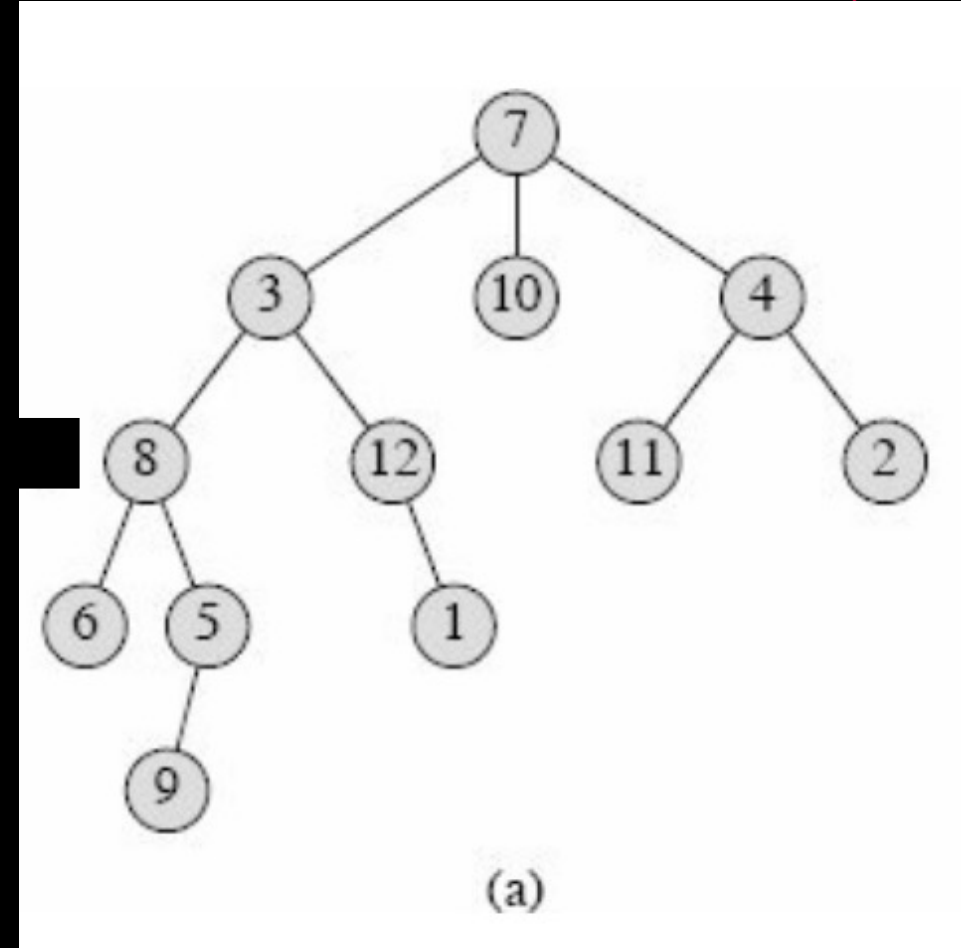


## 5.1.1. Tree and Binary Tree

### Properties:

Consider a node  $x$  in a rooted tree  $T$  with root  $r$ .

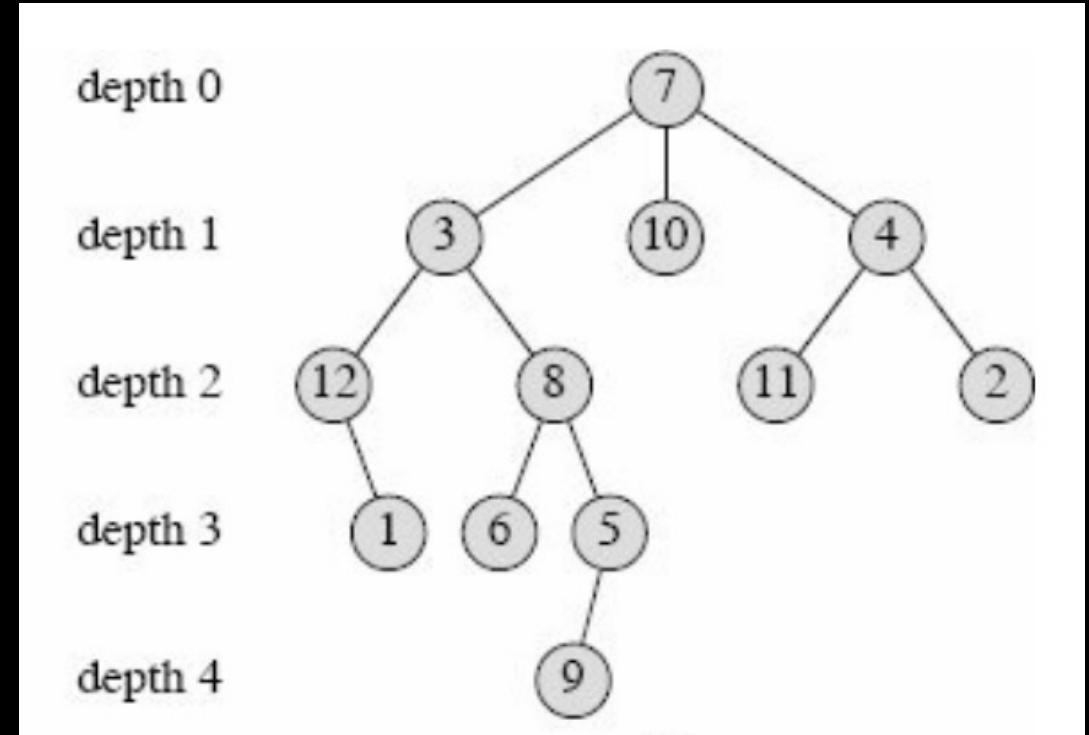
- **Ancestor:** Any node  $y$  on the unique simple path from  $r$  to  $x$ ,  $y$  is an **ancestor** of  $x$ .
- **Descendant:** If  $y$  is an ancestor of  $x$ , then  $x$  is a descendant of  $y$ .
- Every node is both an ancestor and a descendant of itself.
- **Subtree** rooted at  $x$ : A tree induced by descendant of  $x$ .
- In a simple path from  $r$  to  $x$  is  $(y, x)$ ,  $y$  is the **parent** of  $x$ , and  $x$  is a **child** of  $y$ .
- If two nodes have the same parent, they are **siblings**.



## 5.1.1. Tree and Binary Tree

### Characteristics:

- **Degree** of  $x$ : the number of children of a node  $x$ .
- **Depth** of  $x$ : the length of the simple path.
- **Level**: consists of all nodes at the same depth.
- **Height**: the number of edges on the **longest simple downward path** from the node to a leaf. It is also equal to the largest depth of any node in the tree.

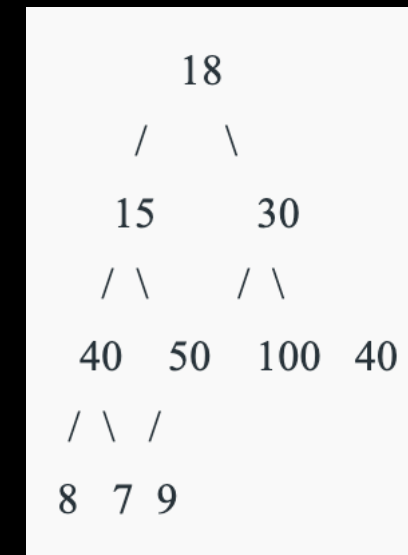
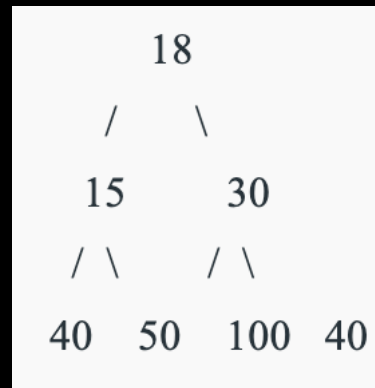
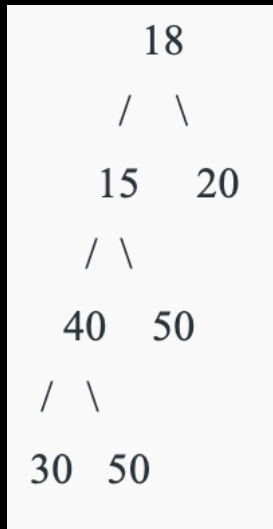




## 5.1.1. Tree and Binary Tree

A **binary tree** is a structure defined on a finite set of nodes that either

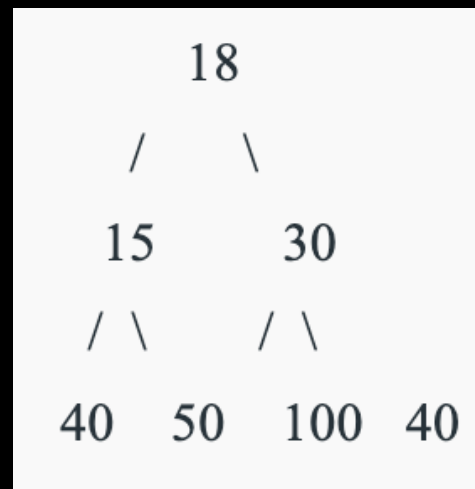
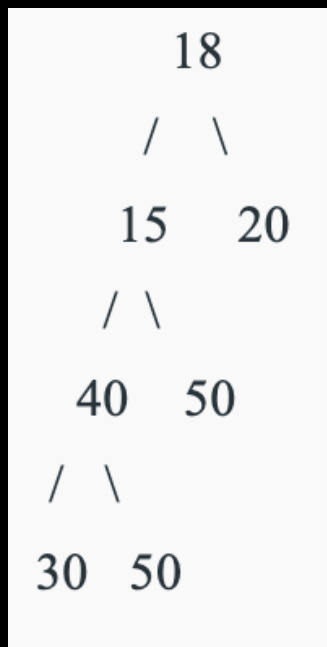
- contains no nodes or
- composed of three disjoint sets of nodes – a root node, left subtree, right subtree.





## 5.1.1. Tree and Binary Tree

- A **binary tree** is not simply an ordered tree in which each node has degree at most of 2.
- **Full binary tree**: each node is either a leaf or has degree of exactly 2. (there is no degree-1 nodes).
- **Perfect binary tree**: a full binary tree of height  $h$  with exactly  $2^h - 1$  nodes.
- **Complete binary tree**: a perfect binary tree through  $h - 1$  with some extra leaf nodes at depth  $h$ , all toward the left.



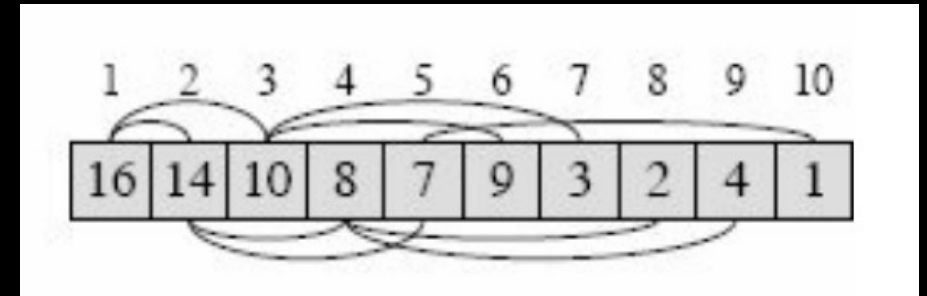
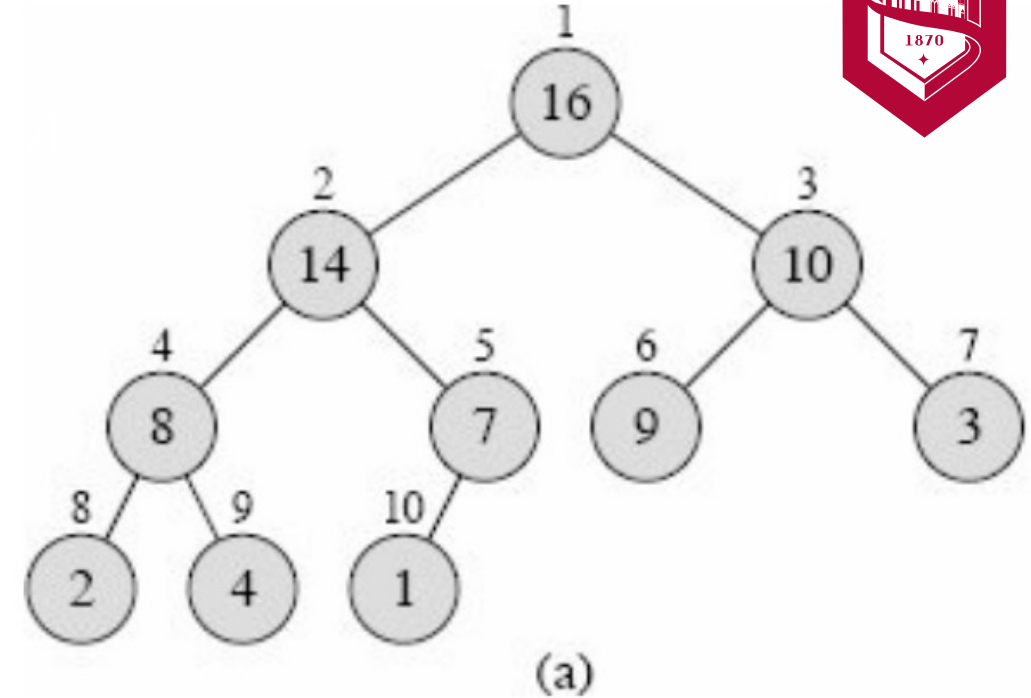


## 5.1.2. Heap

A **heap** (or binary heap) data structure is an array object that is a complete binary tree with following properties

- The root of the tree is  $A[1]$
- Every subtree is a heap. (e.g.,  $0 \leq A.heapsize \leq A.length$ )

The indices of its parent, left child, and right child are  $A[\lfloor \frac{i}{2} \rfloor]$ ,  $A[2i]$ ,  $A[2i + 1]$ , respectively.

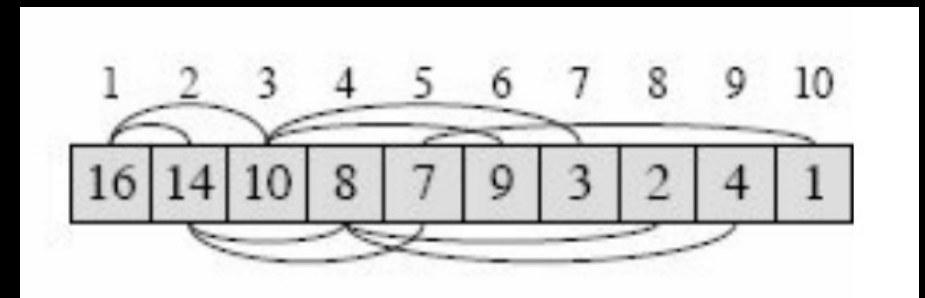
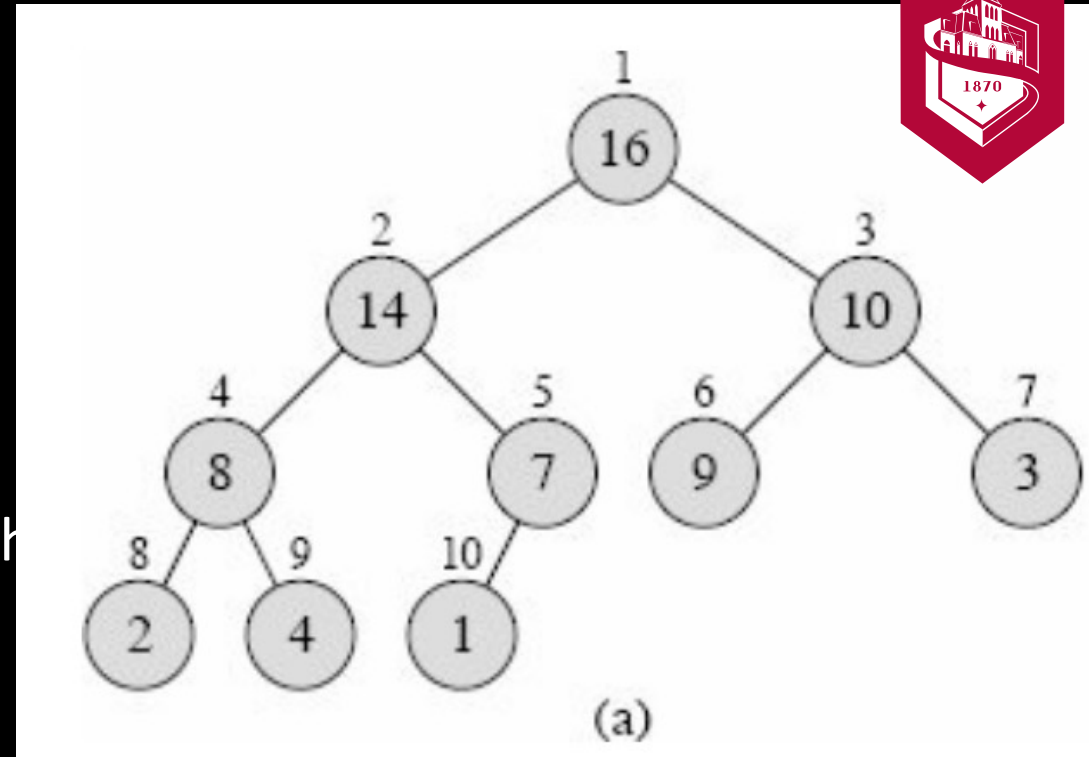


## 5.1.2. Heap



### Two kinds of binary heaps:

- Max-heap:
  - The value of a node is the most the value of its parents,  $A[\text{Parent}(i)] \geq A[i]$ .
  - The largest element in a max-heap is stored at the root.
- Min-heap:
  - Opposite of Max-heap:  $A[\text{Parent}(i)] \leq A[i]$ .
  - The smallest element in a min-heap is at the root.





## 5.1.2. Heap

The height of heap is the height of its root.

- A heap with  $n$  elements based on a complete binary tree runs  $\Theta(\lg n)$  time.

### Proof:

Given an  $n$ -element heap of height  $k$  and is an almost-complete binary tree,

- it has at most  $2^{k+1} - 1$  elements and at least  $2^k$  elements (if the lowest level has just 1 element and the other levels are complete).
- then  $2^k \leq n \leq 2^{k+1} - 1 < 2^{k+1}$ .
- Thus,  $k \leq \lg n < k + 1$ . Since  $k$  is an integer,  $k = \lfloor \lg n \rfloor$ .

### General Algorithmic Procedure of Heapsort:

1. The MAX-HEAPIFY procedure is the key to maintaining the max-heap property.
2. The BUILD-MAX-HEAP procedure produces a max-heap from an unordered input array.
3. The HEAPSORT sorts an array in place.
4. The MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, and HEAP-MAXIMUM allow the heap data structure to implement a priority queue. (Not going to discuss)



## 5.1.2. Heap

The MAX-HEAPIFY assumption:

- the binary trees rooted at  $LEFT(i)$  and  $RIGHT(i)$  are max-heaps
- but that  $A[i]$  might be smaller than its children and violates the max-heap property.
- It lets have value at  $A[i]$  “float down” in the max-heap – so the subtree rooted at index  $i$  obeys the max-heap property.

### Algorithm (MAX-HEAPIFY( $A, i, n$ ))

```
(1)  $l = LEFT(i)$   $r = RIGHT(i)$ 
(2) if  $(l \leq n \text{ and } A[l] > A[i])$  then
(3)    $largest = l$ 
(4) else
(5)    $largest = i$ 
(6) if  $(r \leq n \text{ and } A[r] > A[largest])$  then
(7)    $largest = r$ 
(8) if  $(largest \neq i)$  then
(9)   swap  $A[i]$  and  $A[largest]$ 
(10)  MAX-HEAPIFY( $A, largest, n$ )
```

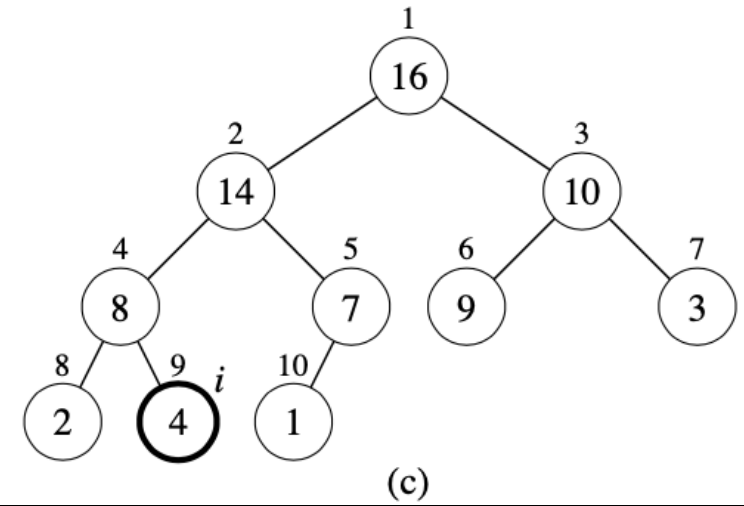
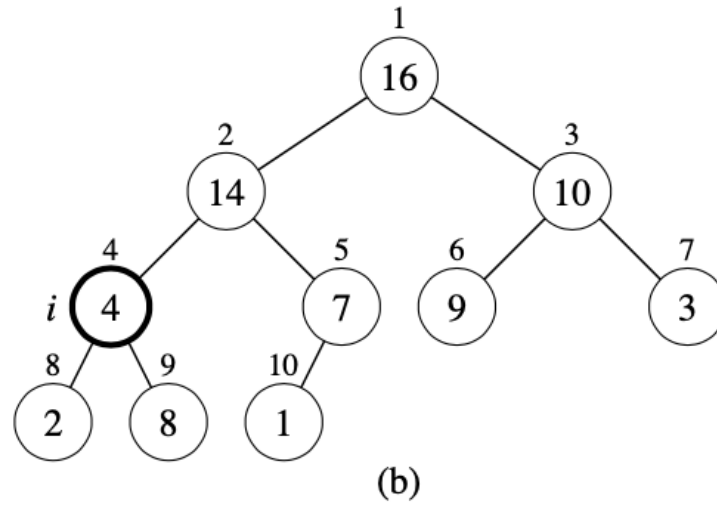
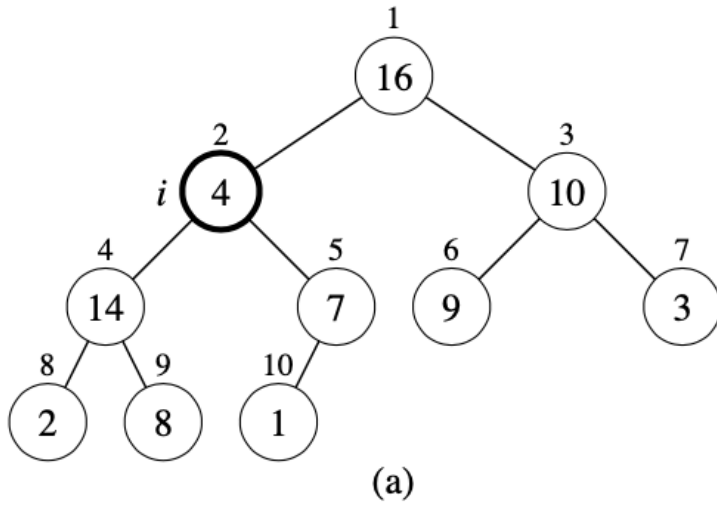
- Compare  $A[i]$ ,  $A[LEFT(i)]$ , and  $A[RIGHT(i)]$ .
- If necessary, swap  $A[i]$  with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at  $i$  is max-heap. If we hit a leaf, then the subtree rooted at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



## 5.1.2. Heap

- Heap is almost-complete binary tree.
- Compares 3 items and maybe swapping 2.
- Fixing up the relationships among the element  $A[i]$ ,  $A[LEFT(i)]$ , and  $A[RIGHT(i)]$  at a given node  $i$  takes  $\Theta(1)$  time.
- The worst case occurs at the bottom level of the tree.
- The children's subtrees each have size at most  $2n/3$ . (only half full case)
- The running time  $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$ .
- Using MT2 ( $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$ ), we find that  $\log_{\frac{3}{2}} 1 = 0$ , so  $n^0 = 1$ .
- With  $k = 0$ , we can see that  $\lg n \geq 1$  if  $n \geq 2$ . Therefore, we can say that  $T(n) = O(\lg n)$ .

## 5.1.2. Heap





## 5.1.2. Heap

### Building a heap

- Using MAX-HEAPIFY in a *bottom-up* manner to convert an array  $A[1 \dots n]$  into a max-heap.
- $A \left[ \left( \left\lfloor \frac{n}{2} \right\rfloor + 1 \right) \dots n \right]$  are all leaves of the tree so each is a 1-element heap to begin with.

#### Algorithm (BUILD-MAX-HEAP( $A, n$ ))

```
(1)  for ( $\lfloor \frac{n}{2} \rfloor \geq i \geq 1$ ) do
(2)    MAX-HEAPIFY( $A, i, n$ )
```

#### Correctness:

- Initialization – since  $i = \left\lfloor \frac{n}{2} \right\rfloor$  before the first iteration of the for loop, the invariant is initially true.
- Maintenance – Decrementing  $i$  reestablishes the loop invariant at each iteration.
- Termination – When  $i = 0$ , the loop terminates.



## 5.1.2. Heap

### Building a heap – Analysis

- **simple bound:** Each call to MAX-HEAPIFY makes  $O(\lg n)$  and BUILD-MAX-HEAP makes  $O(n)$ . Thus, the running time is  $O(n \lg n)$ .
- **tighter analysis:** Can observe that the running time for MAX-HEAPIFY is linear in the height of the node its running on. Most nodes have small heights.
- We have  $\leq \left\lceil \frac{n}{2^{k+1}} \right\rceil$  nodes of height  $k$  and height of heap is  $\lfloor \lg n \rfloor$ .

#### How?

- The tree leaves (nodes at height 0) are at depth  $K$  and  $K - 1$ .
- They consist of all nodes at depth  $K$ , and the nodes at depth  $K - 1$  that are not parents of depth- $K$  nodes.





## 5.1.2. Heap

### Building a heap - Analysis

- Let  $x$  be the number of nodes at depth  $K$  in the bottom level. (if  $n$  is odd,  $x$  is even or vice versa).
- If  $x$  is even, there are  $x/2$  nodes at  $K - 1$  that are parent of  $K$  depth nodes.
- Hence,  $2^{K-1} - x/2$  nodes at  $K - 1$  depth are not parents of  $K$  depth nodes.
- The total nodes =  $x + 2^{K-1} - \frac{x}{2} = \frac{2^K + x}{2} = \left\lceil \frac{(2^K + x - 1)}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil$ .
- If  $x$  is odd, using the similar argument, the total nodes =  $x + 2^{K-1} - \frac{x+1}{2} = \frac{2^K + x - 1}{2} = \left\lceil \frac{n}{2} \right\rceil$ .
- Let  $n_h$  be the number of nodes at height  $h$  in the  $n$ -node tree  $T$ .
- Considering the tree  $T'$  formed after removing the leaves of tree  $T$ .
- Then it has  $n' = n - n_h$  nodes. Since we know that  $n_h = \left\lceil \frac{n}{2} \right\rceil$  from the base case,  $n' = \left\lfloor \frac{n}{2} \right\rfloor$ .



## 5.1.2. Heap

### Building a heap - Analysis

- Let  $n'_{h-1}$  be the number of nodes at height  $k - 1$  in  $T'$ .
- Then  $n_h = n'_{h-1}$ . We can bound that  $n_h = n'_{h-1} \leq \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\lfloor n/2 \rfloor}{2^k} \right\rceil \leq \left\lceil \frac{n/2}{2^k} \right\rceil = \left\lceil \frac{n}{2^{k+1}} \right\rceil$ .
- The time required by MAX-HEAPIFY when called on a node of height  $h$  is  $O(h)$ , so the total cost of BUILD-MAX-HEAP is

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

- Thus, the running time of BUILD-MAX-HEAP is  $O(n)$ .
- The same argument works for min-heap and MIN-HEAPIFY.

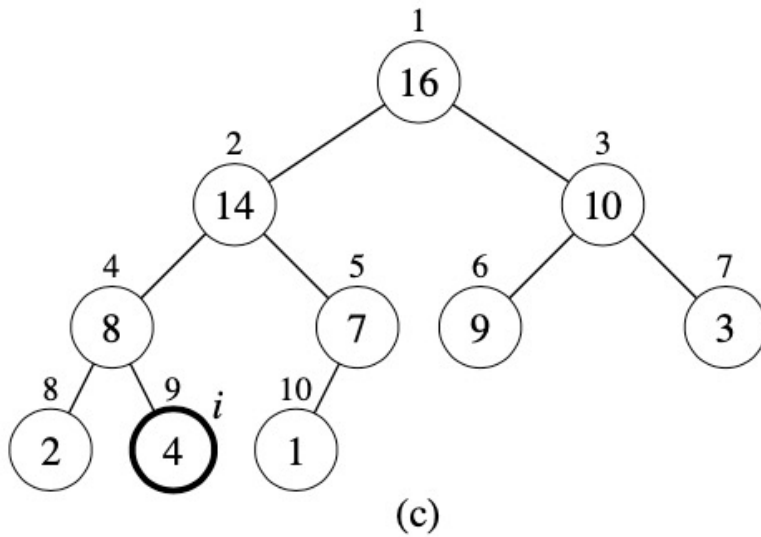
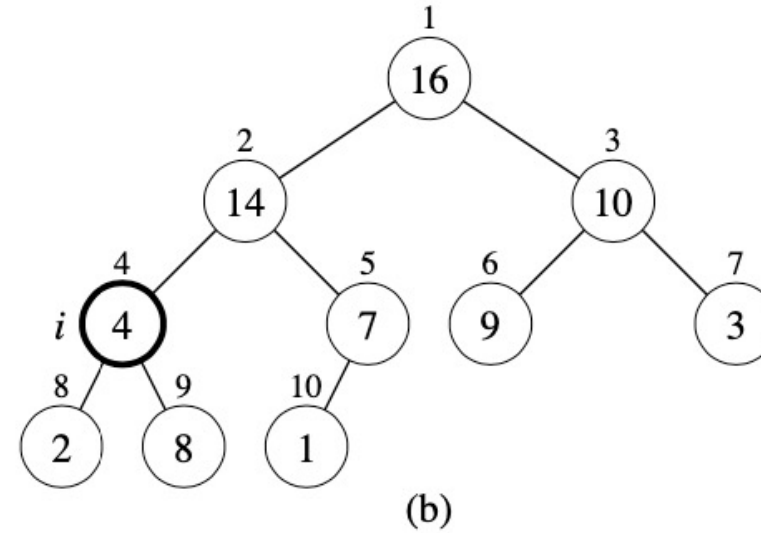
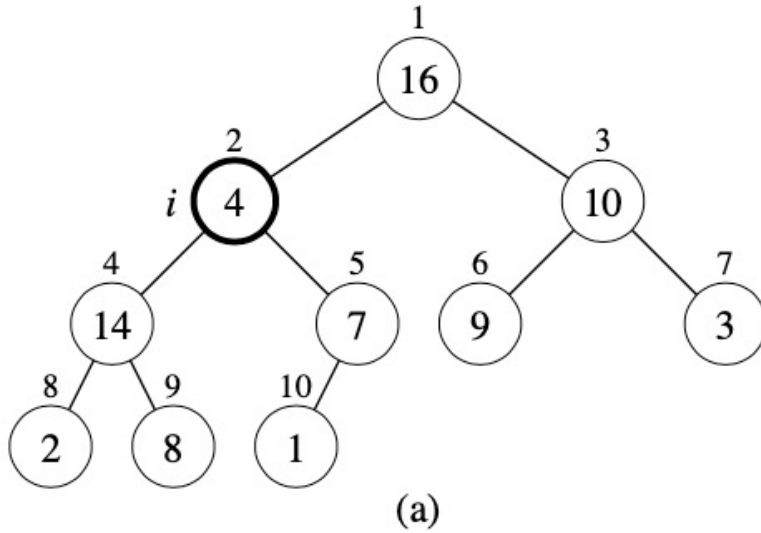


## 5.1.3. Heapsort

For a given input array, the Heapsort works as follows:

- We build a max-heap from the array.
- We start with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- We discard this last node by decreasing the heap size and call MAX-HEAPIFY on the new and possibly incorrectly placed root.
- We repeat this process until only one node (the smallest element) remains, which is in the correct place.

## 5.1.3. Heapsort





## 5.1.3. Heapsort

### Algorithm (HEAPSORT( $A, n$ ))

```
(1)  BUILD-MAX-HEAP( $A, n$ )
(2)  for ( $n \geq i \geq 2$ ) do
(3)    swap  $A[1]$  and  $A[i]$ 
(4)    MAX-HEAPIFY( $A, 1, i - 1$ )
(5)  od
```

- BUILD-MAX-HEAP:  $O(n)$
- for loop:  $n - 1$  times
- exchange elements:  $O(1)$
- MAX-HEAPIFY:  $O(\lg n)$

The running time is  $O(n \lg n)$ .

- Compare to Insertion Sort  $\Theta(n^2)$  and Merge-Sort  $\Theta(n \lg n)$
- The heapsort is slower than the merge sort. **But it does not require massive recursion or multiple arrays to work.**

# Outline



## 5.1. Heapsort

### 5.1.1. Tree & Binary Tree

### 5.1.2. Heap

### 5.1.3. Heapsort

## 5.2. Quick Sort

## 5.3. Counting Sort

## 5.4. Radix Sort

## 5.5. Bucket Sort

## 5.6. Conclusion



## 5.2. Quick Sort

Quicksort is another sorting algorithm based on the divide-and-conquer process.

- Divide: the partition  $A[p \dots r]$  into two subarrays  $A[p \dots q - 1]$  and  $A[q + 1 \dots r]$ , such that each element in  $A[p \dots q - 1]$  is  $\leq A[q]$  and  $A[q]$  is  $\leq$  in each element in  $A[q + 1 \dots r]$ .
- Conquer: We sort the two subarrays by recursive calls to QUICKSORT.
- Combine: No need to combine the subarrays, because they are sorted in place.

### Algorithm (QUICKSORT( $A, p, r$ ))

```
(1)  if ( $p < r$ ) then
(2)     $q = \text{PARTITION}(A, p, r)$ 
(3)    QUICKSORT( $A, p, q - 1$ )
(4)    QUICKSORT( $A, q + 1, r$ )
(5)  fi
```

\*Initial call is QUICKSORT( $A, 1, n$ )

## 5.2. Quick Sort



### Algorithm (PARTITION( $A, p, r$ ))

```
(1)  $x = A[r]$ 
(2)  $i = p - 1$ 
(3) for  $(p \leq j \leq r - 1)$  do
(4)   if  $(A[j] \leq x)$  then
(5)      $i = i + 1$ 
(6)      $SWAP(A[i], A[j])$ 
(7)   fi
(8) od
(9)  $SWAP(A[i + 1], A[r])$ 
(10) return  $i + 1$ 
```

- PARTITION rearranges the subarray in place.
- PARTITION always selects the last element  $A[r]$  in the subarray  $A[p \dots r]$  as the pivot (the element around which to partition).
- As the procedure executes, the array is partitioned into four regions, some which may be empty:



## 5.2. Quick Sort



### Algorithm (PARTITION( $A, p, r$ ))

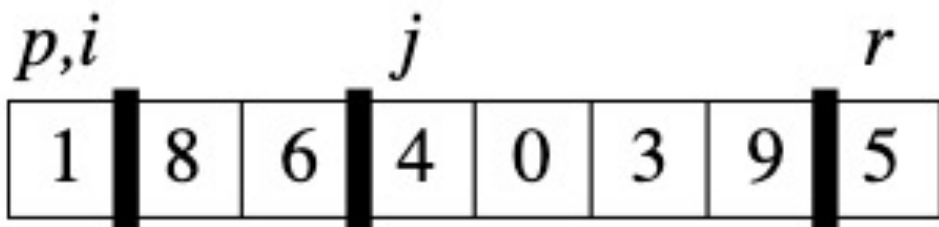
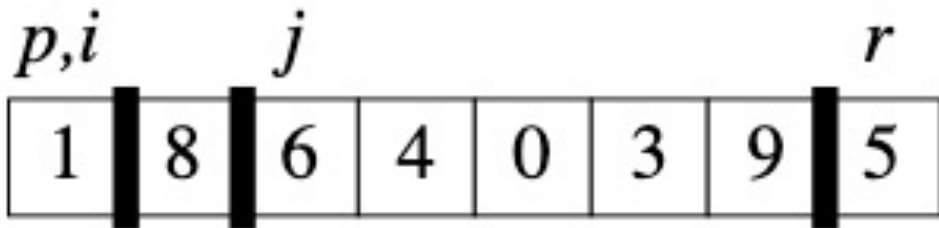
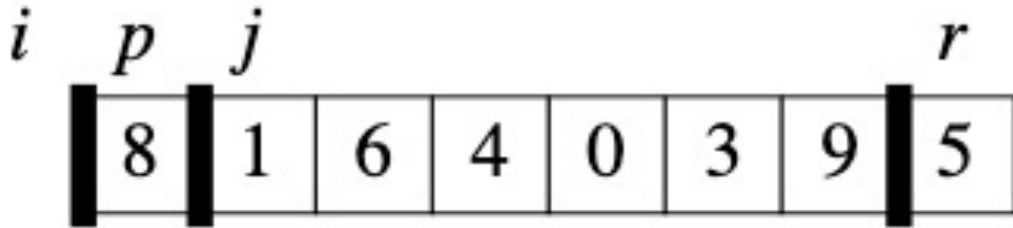
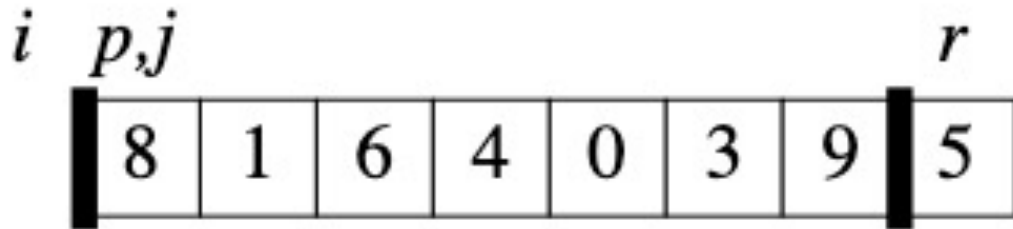
```
(1)  $x = A[r]$ 
(2)  $i = p - 1$ 
(3) for  $(p \leq j \leq r - 1)$  do
(4)   if  $(A[j] \leq x)$  then
(5)      $i = i + 1$ 
(6)     SWAP( $A[i], A[j]$ )
(7)   fi
(8) od
(9) SWAP( $A[i + 1], A[r]$ )
(10) return  $i + 1$ 
```

#### Loop invariant:

1. All entries in  $A[p \dots i]$  are  $\leq$  pivot.
2. All entries in  $A[i + 1 \dots j - 1]$  are  $>$  pivot.
3.  $A[r] = \text{pivot}$ .

The additional region  $A[j \dots r - 1]$  consists of elements that have not yet been processed. We do not yet know how they compare to the pivot element.

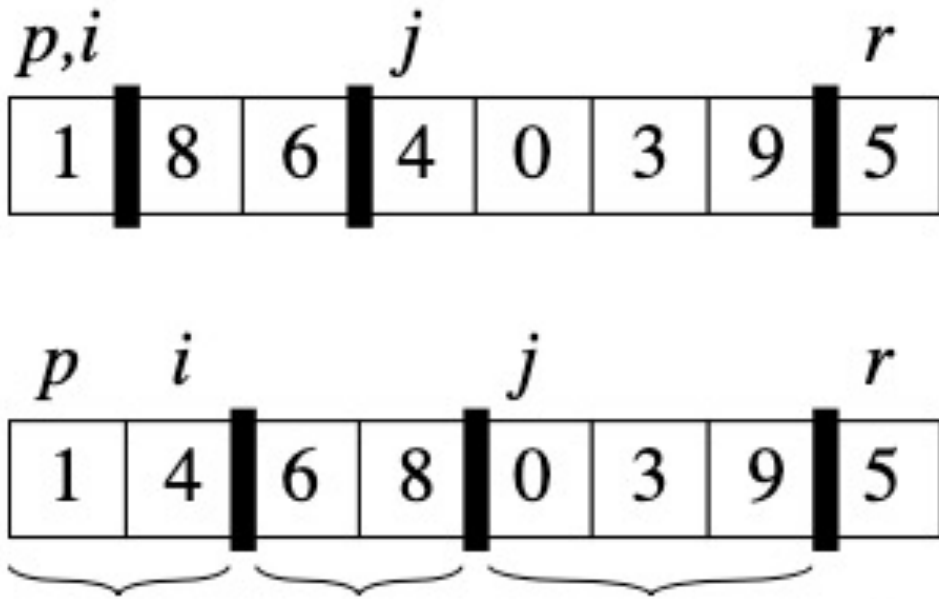
## 5.2. Quick Sort



### Algorithm (PARTITION(A,p,r))

```
(1)  $x = A[r]$ 
(2)  $i = p - 1$ 
(3) for ( $p \leq j \leq r - 1$ ) do
(4)   if ( $A[j] \leq x$ ) then
(5)      $i = i + 1$ 
(6)      $SWAP(A[i], A[j])$ 
(7)   fi
(8) od
(9)  $SWAP(A[i + 1], A[r])$ 
(10) return  $i + 1$ 
```

## 5.2. Quick Sort



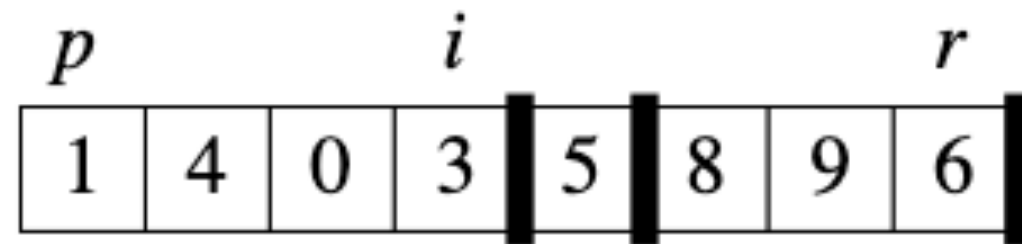
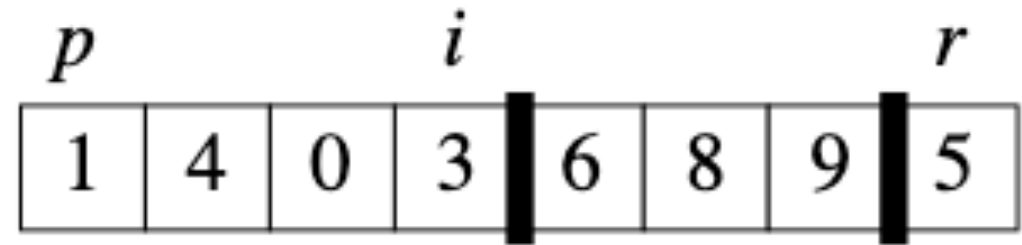
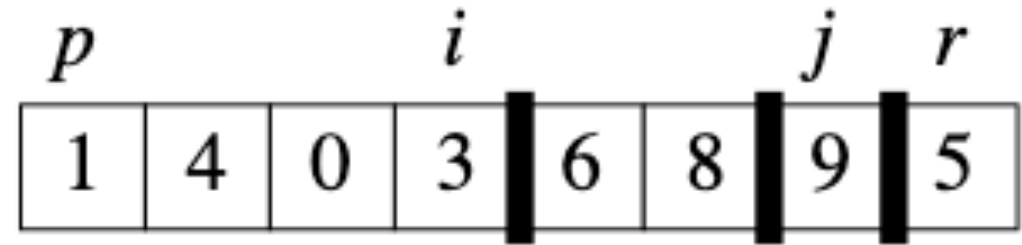
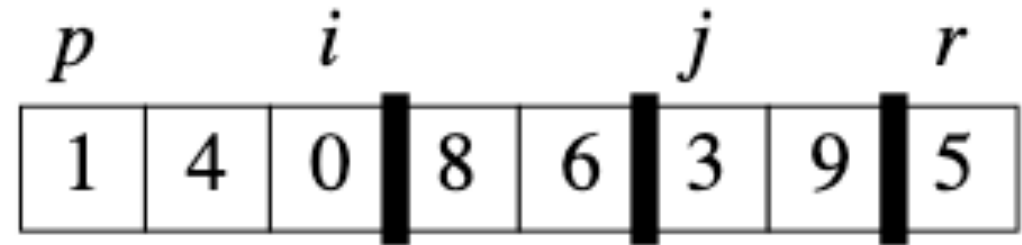
### Algorithm (PARTITION( $A, p, r$ ))

```
(1)  $x = A[r]$ 
(2)  $i = p - 1$ 
(3) for ( $p \leq j \leq r - 1$ ) do
(4)   if ( $A[j] \leq x$ ) then
(5)      $i = i + 1$ 
(6)     SWAP( $A[i], A[j]$ )
(7)   fi
(8) od
(9) SWAP( $A[i + 1], A[r]$ )
(10) return  $i + 1$ 
```

## 5.2. Quick Sort

### Algorithm (PARTITION(A,p,r))

```
(1)  $x = A[r]$ 
(2)  $i = p - 1$ 
(3) for  $(p \leq j \leq r - 1)$  do
(4)   if  $(A[j] \leq x)$  then
(5)      $i = i + 1$ 
(6)     SWAP( $A[i], A[j]$ )
(7)   fi
(8) od
(9) SWAP( $A[i + 1], A[r]$ )
(10) return  $i + 1$ 
```





## 5.2. Quick Sort

- **PARTICIPATION Correctness: We use the loop invariant.**
- **Initialization:** The subarray  $A[p \dots i]$  and  $A[i + 1 \dots j - 1]$  are empty before the start of the loop. The loop invariant conditions are therefore satisfied.
- **Maintenance:** Inside of the loop – we swap  $A[j]$  and  $A[i + 1]$  and increase  $i$  if  $A[j] \leq$  pivot element. The loop counter  $j$  is incremented.
- **Termination:** Loop terminates for  $j = r$ . We have three regions  $A[p \dots i] \leq$  pivot,  $A[i + 1 \dots r - 1] >$  pivot and  $A[r] =$  pivot element.
- At the end of PARTITION we swap the pivot element in its correct position and then return its position.
- Time:  $\Theta(n)$  to partition the subarray of size  $n$ .



## 5.2. Quick Sort

The running time of QUICKSORT depends on the partitioning of the subarrays.

- QUICKSORT is fast as MERGE-SORT if the partitioned subarrays are balanced (even sized).
- QUICKSORT is slow as INSERTION SORT if the partitioned subarrays are unbalanced (uneven sized).

**Worst case:** Subarrays completely unbalanced.

- Have 0 elements in one subarray and  $n - 1$  elements in the other subarray.
- The recurrence running time:

$$\begin{aligned} T(n) &= T(n - 1) + T(0) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

- The running time is like INSERTION-SORT.



## 5.2. Quick Sort

**Best case:** Subarrays are always completely balanced.

- Each subarray has  $\leq n/2$  elements.
- The recurrence running time:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$



## 5.2. Quick Sort

### Balanced partitioning:

- We assume that PARTITION always produces a 9 to 1 split.
- Then the recurrence is

$$\begin{aligned} T(n) &\leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) \\ &= O(n \lg n) \end{aligned}$$

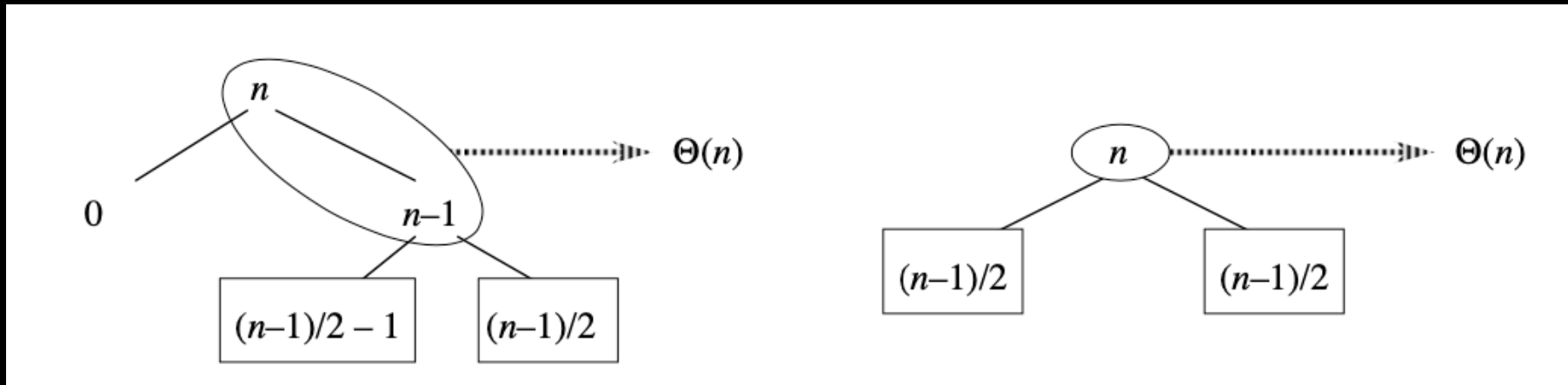
- In the recursion tree, we get  $\log_{10} n$  full levels and  $\log_{10/9} n$  levels that are nonempty.
- As long as it's a constant, the base of the log does not matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth  $\Theta(\lg n)$ .



## 5.2. Quick Sort

### Average case:

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of “good” and “bad” splits throughout the recursion tree.
- It does not affect the asymptotic running time – assume that levels alternate between best- and worst-case splits.
- The bad split only adds to constant hidden in  $\Theta$  notation.
- The same number of subarrays to sort but twice as much work is needed to this point.
- Both splits result in  $\Theta(n \lg n)$  time, though the constant on the bad split is higher.





## 5.2. Quick Sort Randomized Quicksort

- We assumed so far that all input permutation are equally likely which is not always the case.
- We introduce randomization in order to improve on the quicksort algorithm.
- One option would be to use a random permutation of the input array.
- We use random sampling instead which is picking one element at random.
- Instead of using  $A[r]$  as the pivot element we randomly pick an element from the subarray.

```
RANDOMIZED-PARTITION( $A, p, r$ )  
1   $i = \text{RANDOM}(p, r)$   
2   $\text{EXCHANGE } A[r] \leftrightarrow A[i]$   
3  return PARTITION( $A, p, r$ )
```

\*Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

```
RANDOMIZED-QUICKSORT( $A, p, r$ )  
1  if  $p < r$   
2     $q = \text{RANDOMIZED-PARTITION}(A, p, r)$   
3    RANDOMIZED-QUICKSORT( $A, p, q - 1$ )  
4    RANDOMIZED-QUICKSORT( $A, q + 1, r$ )
```

\*Randomization of quicksort stops any specific type of array from causing worst-case behavior. For example, an already-sorted array causes worst-case behavior in non-randomized quicksort, but not in randomized-quicksort.