Design and Analysis of Algorithms

Part I: Divide and Conquer

Lecture 9: Heapsort, Lower Bound for Sorting and Sorting in Linear Time

Ke Xu and Yongxin Tong

（许可 与 童咏昕）

School of CSE, Beihang University

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

Review to Divide-and-Conquer Paradigm

- Divide-and-conquer (D&C) is an important algorithm design paradigm.
  - **Divide**

    Dividing a given problem into two or more subproblems (ideally of approximately equal size)
  - **Conquer**

    Solving each subproblem (directly if small enough or recursively)
  - **Combine**

    Combining the solutions of the subproblems into a global solution

Review to Divide-and-Conquer Paradigm

- In Part I, we will illustrate Divide-and-Conquer using several examples:
  - Maximum Contiguous Subarray (最大子数组)
  - Counting Inversions (逆序计数)
  - Polynomial Multiplication (多项式乘法)
  - QuickSort and Partition (快速排序与划分)
  - Randomized Selection (随机化选择)
  - Supplement Topic of Sorting (排序问题补充主题)
    - Heapsort (堆排序)
    - Lower Bound for Sorting (基于比较的排序下界)
    - Sorting in Linear Time (线性时间排序)

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

Priority Queue: Motivating Example

3 jobs have been submitted to a printer in the order A, B, C. Consider the printing pool at this moment.

Sizes: Job A — 100 pages

Job B — 10 pages

Job C — 1 page

Priority Queue: Motivating Example

3 jobs have been submitted to a printer in the order A, B, C. Consider the printing pool at this moment.

Sizes: Job A — 100 pages

Job B — 10 pages

Job C — 1 page

Average finish time with FIFO service:

$$(100 + 110 + 111)\ /\ 3\ =\ 107\ time\ units$$

Priority Queue: Motivating Example

3 jobs have been submitted to a printer in the order A, B, C. Consider the printing pool at this moment.

Sizes: Job A — 100 pages

Job B — 10 pages

Job C — 1 page

Average finish time with FIFO service:

$$(100 + 110 + 111)\ /\ 3\ =\ 107\ time\ units$$

Average finish time for shortest-job-first service:

$$(1 + 11 + 111)\ /\ 3\ =\ 41\ time\ units$$

Priority Queue: Motivating Example

- The elements in the queue are printing jobs, each with the associated number of pages that serves as its priority
- Processing the shortest job first corresponds to extracting the smallest element from the queue
- Insert new printing jobs as they arrive

Priority Queue: Motivating Example

- The elements in the queue are printing jobs, each with the associated number of pages that serves as its priority
- Processing the shortest job first corresponds to extracting the smallest element from the queue
- Insert new printing jobs as they arrive

A queue is capable of supporting two operations: Insert and Extract-Min?

Priority Queue

Priority queue is an abstract data structure that supports two operations

- Insert: inserts the new element into the queue
- Extract-Min: removes and returns the smallest element from the queue

Possible Implementations

- Unsorted list + a pointer to the smallest element
  - Insert in O(1) time
  - Extract-Min in O(n) time, since it requires a linear scan to find the new minimum

Possible Implementations

- Unsorted list + a pointer to the smallest element
  - Insert in O(1) time
  - Extract-Min in O(n) time, since it requires a linear scan to find the new minimum
- Sorted array
  - Insert in O(n) time
  - Extract-Min in O(1) time

Possible Implementations

- Unsorted list + a pointer to the smallest element
  - Insert in O(1) time
  - Extract-Min in O(n) time, since it requires a linear scan to find the new minimum
- Sorted array
  - Insert in O(n) time
  - Extract-Min in O(1) time
- Sorted doubly linked list
  - Insert in O(n) time
  - Extract-Min in O(1) time

Possible Implementations

- Unsorted list + a pointer to the smallest element
  - Insert in O(1) time
  - Extract-Min in O(n) time, since it requires a linear scan to find the new minimum
- Sorted array
  - Insert in O(n) time
  - Extract-Min in O(1) time
- Sorted doubly linked list
  - Insert in O(n) time
  - Extract-Min in O(1) time

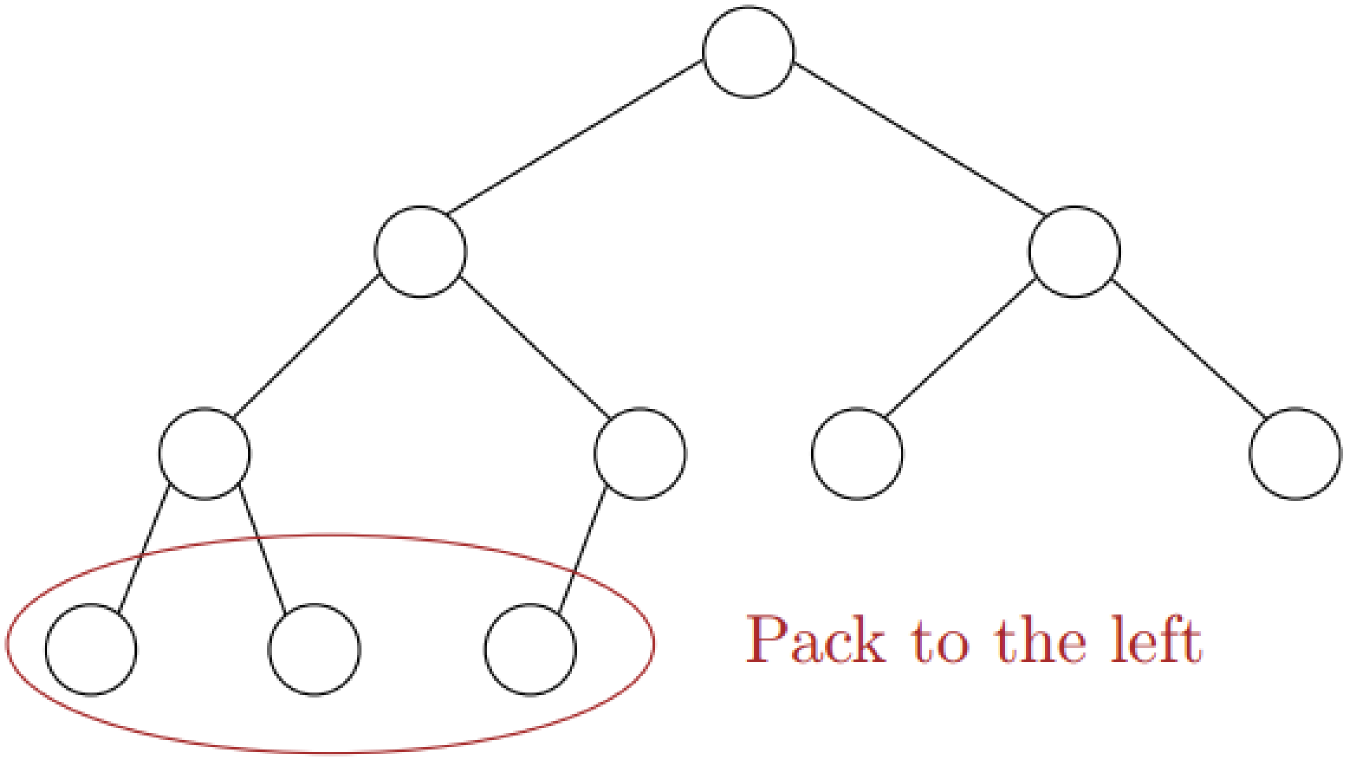
Question

Is there any data structure that supports both these priority queue operations in $O(\log n)$ time?

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

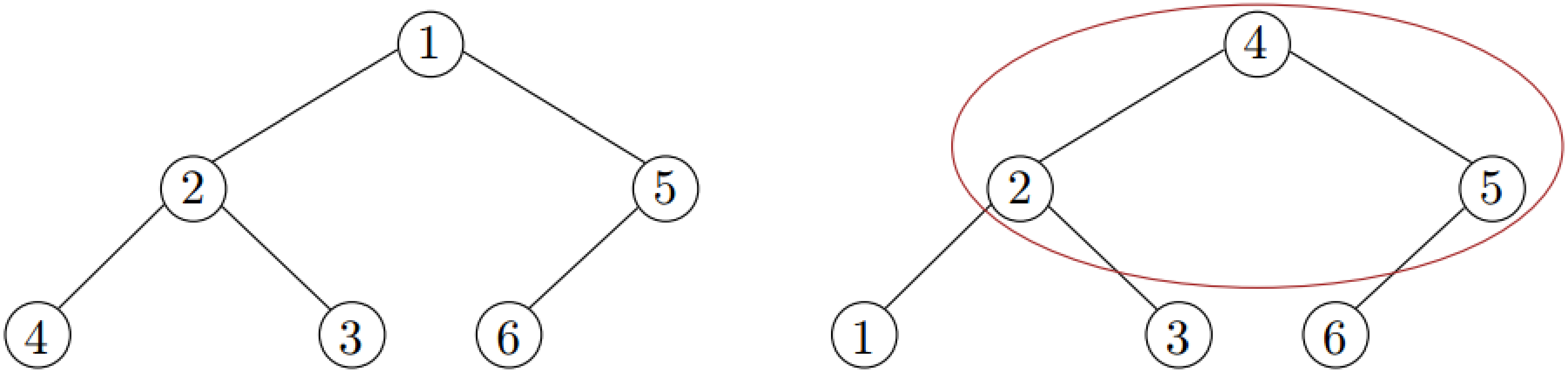
(Binary) Heap



Heaps are "almost complete binary trees"

- All levels are full except possibly the lowest level.
- If the lowest level is not full, then nodes must be packed to the left.

Heap-order Property



A min-heap   Not a heap

Heap-order property (*Min-heap*):

The value of a node is at least the value of its parent.

A[Parent(i)] ≤ A[i]

Heap Properties

- If the heap-order property is maintained, heaps support the following operations efficiently (assume there are n elements in the heap)
  - Insert in O(log n) time
  - Extract-Min in O(log n) time

Heap Properties

- If the heap-order property is maintained, heaps support the following operations efficiently (assume there are *n* elements in the heap)
  - Insert in O(log *n*) time
  - Extract-Min in O(log *n*) time

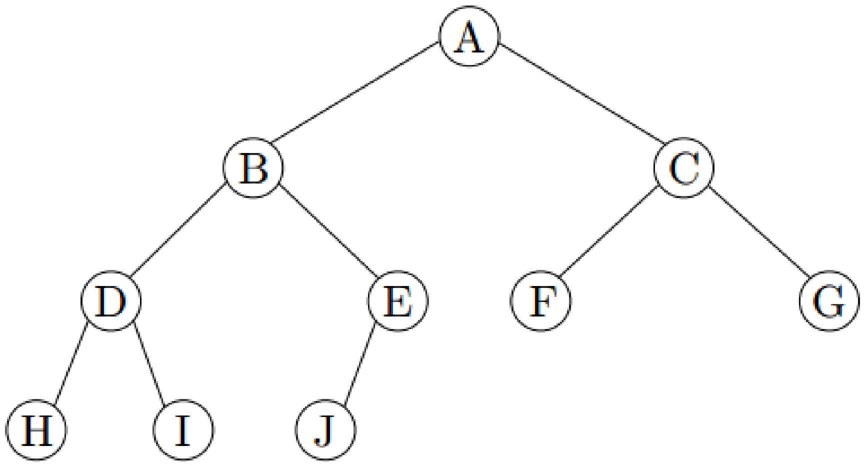
- Structure properties
  - A heap of height *h* has between 2^h to $2^{h+1}-1$ nodes. Thus, an n-element heap has height Θ(log n).

Heap Properties

- If the heap-order property is maintained, heaps support the following operations efficiently (assume there are *n* elements in the heap)
  - Insert in O(log *n*) time
  - Extract-Min in O(log *n*) time

- Structure properties
  - A heap of height *h* has between 2^h to $2^{h+1}-1$ nodes. Thus, an n-element heap has height Θ(log n).
  - The structure is so regular, it can be represented in an array and no links are necessary !

Array Implementation of Heap

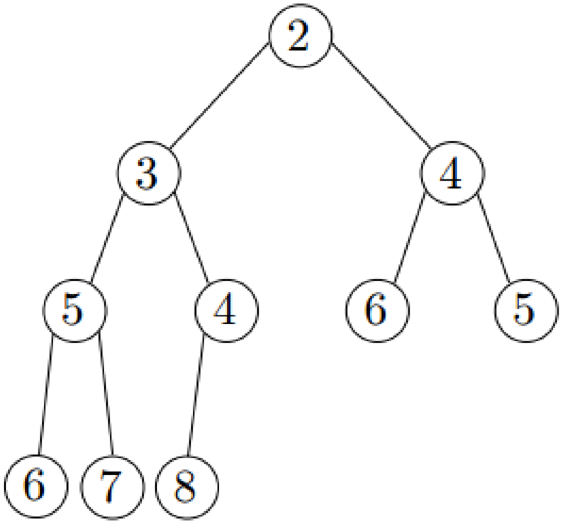


1	2	3	4	5	6	7	8	9	10
A	B	C	D	E	F	G	H	I	J

- The root is in array position 1.
- For any element in array position i,
  - The left child is in position 2i.
  - The right child is in position 2i+1.
  - The parent is in position $\lfloor i/2 \rfloor$.
- We will draw the heaps as trees, with the understanding that an actual implementation will use simple arrays.

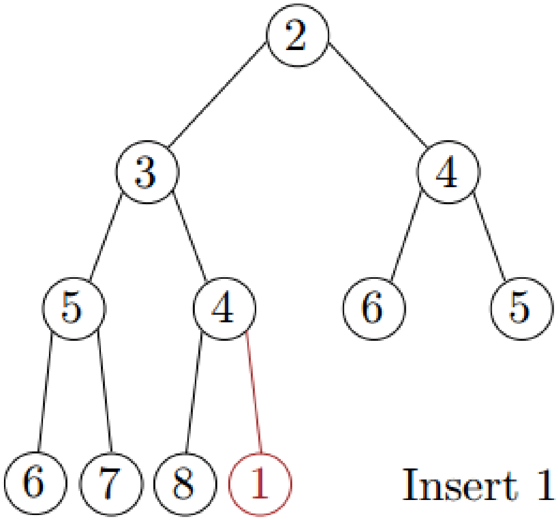
Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



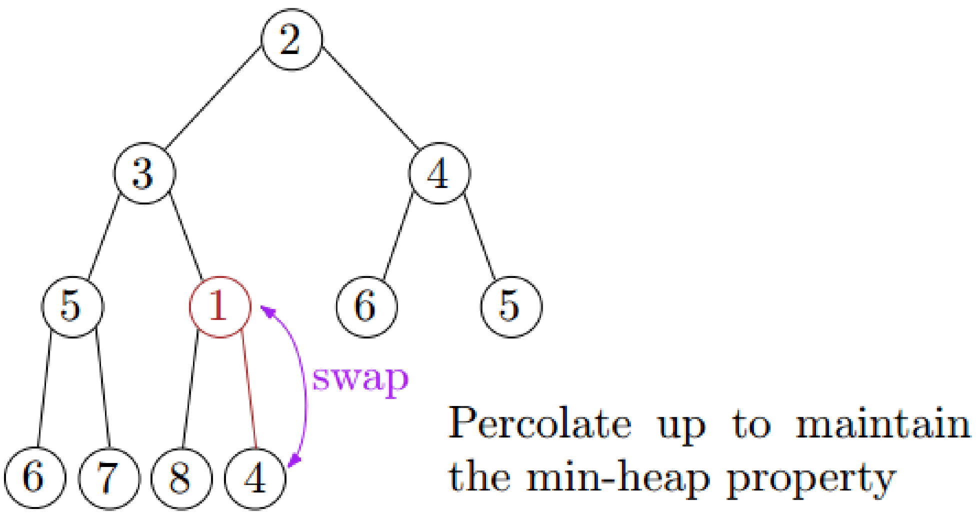
Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



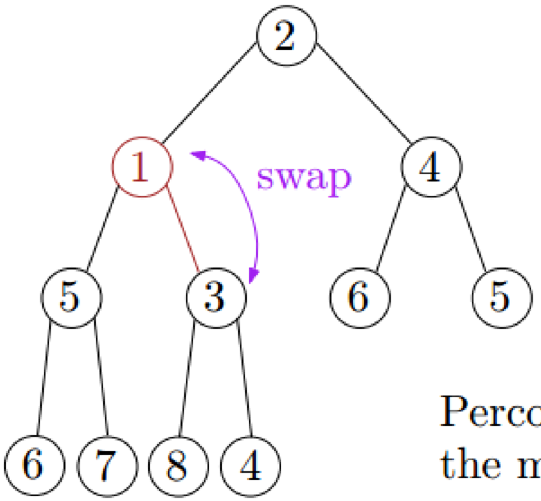
Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Insertion

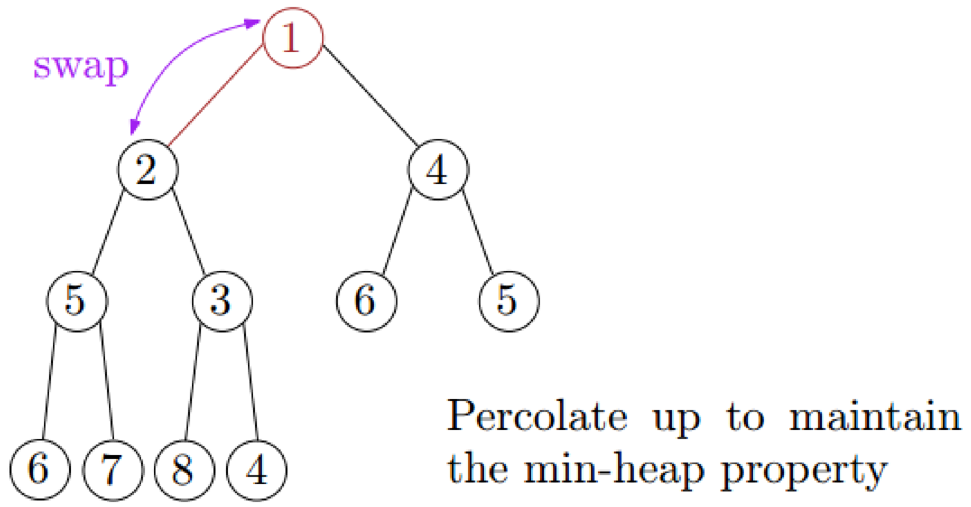
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



Percolate up to maintain the min-heap property

Insertion

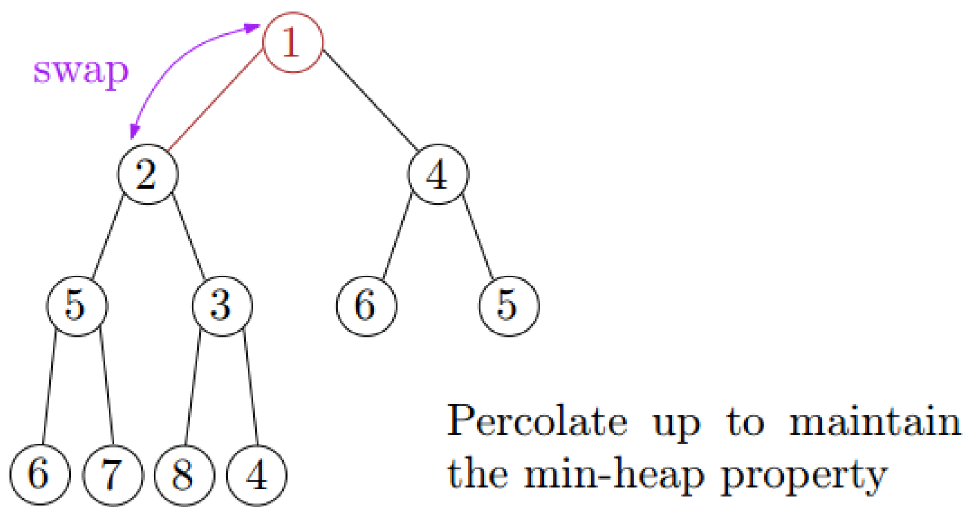
- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.



- Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element

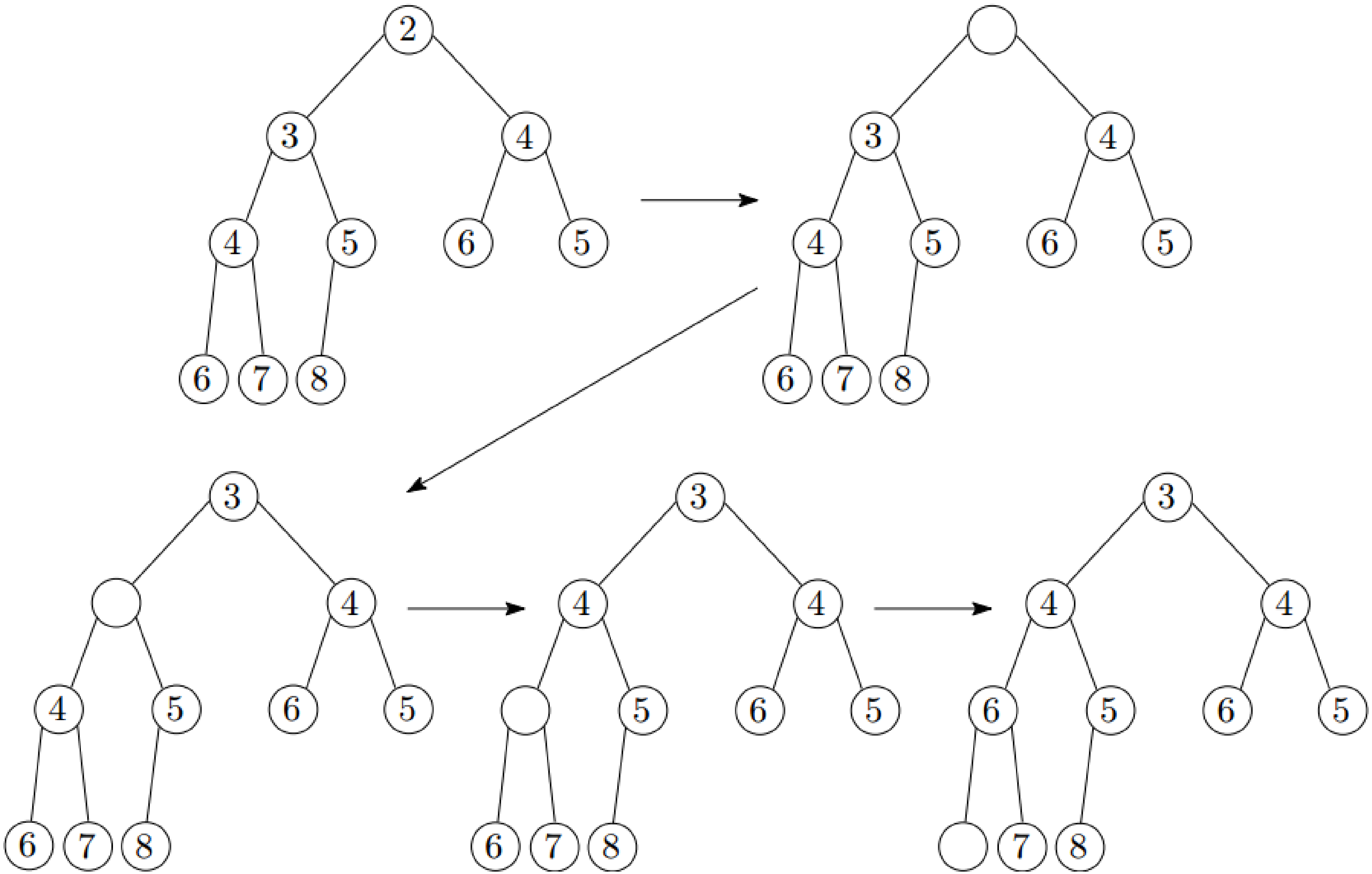
Insertion

- Add the new element to the next available position at the lowest level
- Restore the min-heap property if violated
  - General strategy is percolate up (or bubble up): if the parent of the element is larger than the element, then interchange the parent with child.

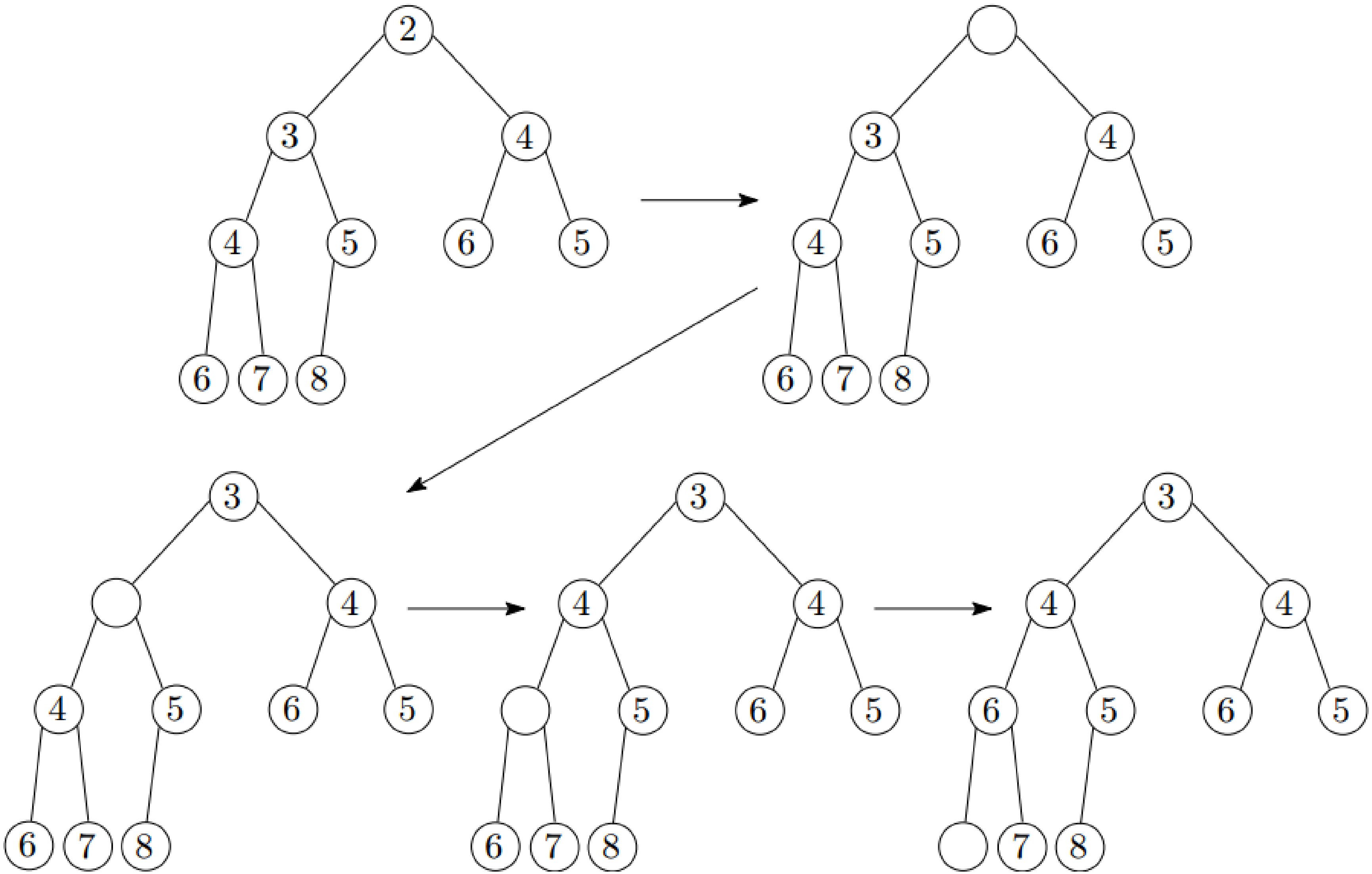


- Correctness: after each swap, the min-heap property is satisfied for the subtree rooted at the new element
- Time complexity = O(height) = O(log n)

Extract-Min: First Attempt



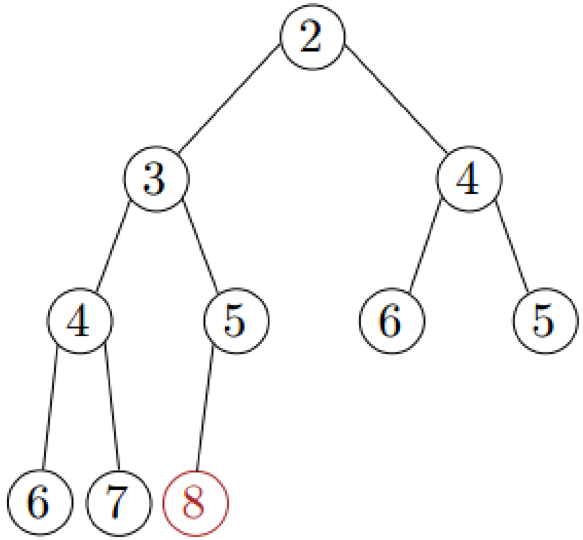
Extract-Min: First Attempt



Min-heap property preserved, but completeness not preserved!

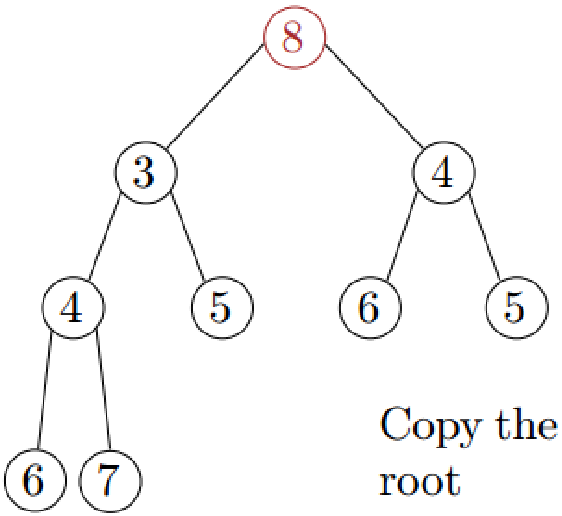
Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Extract-Min

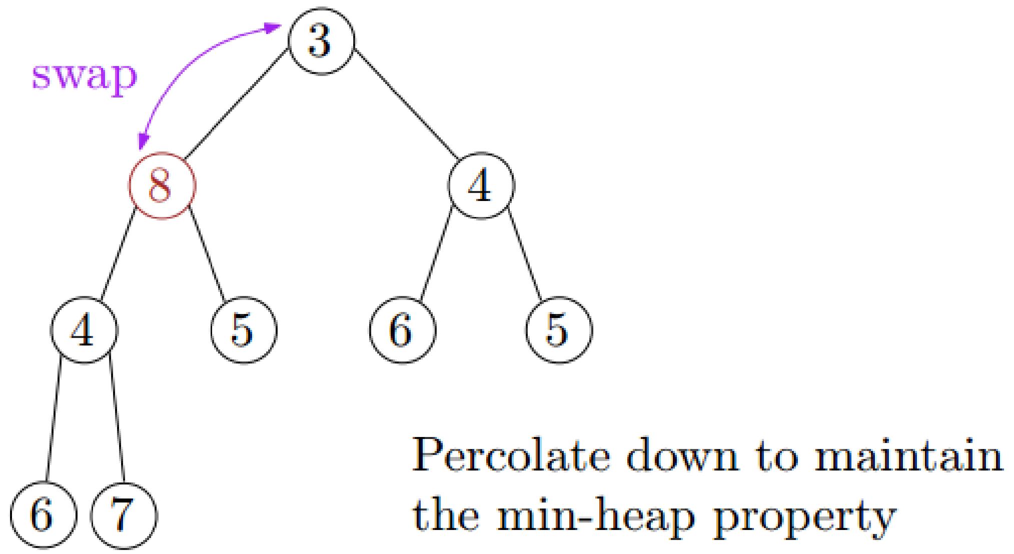
- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Copy the last element to the root

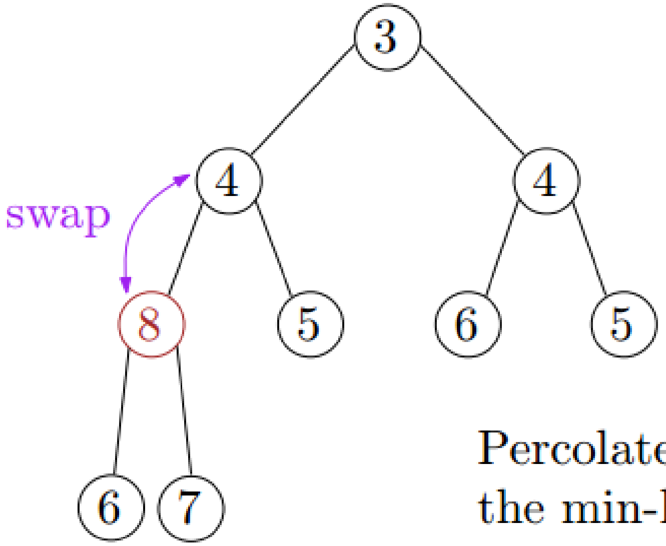
Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Extract-Min

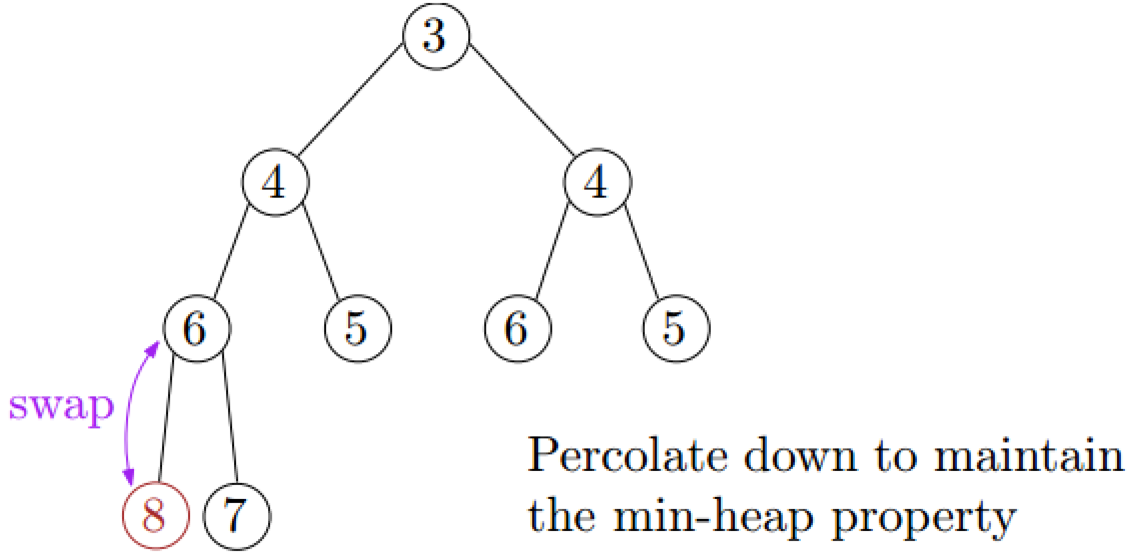
- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



Percolate down to maintain the min-heap property

Extract-Min

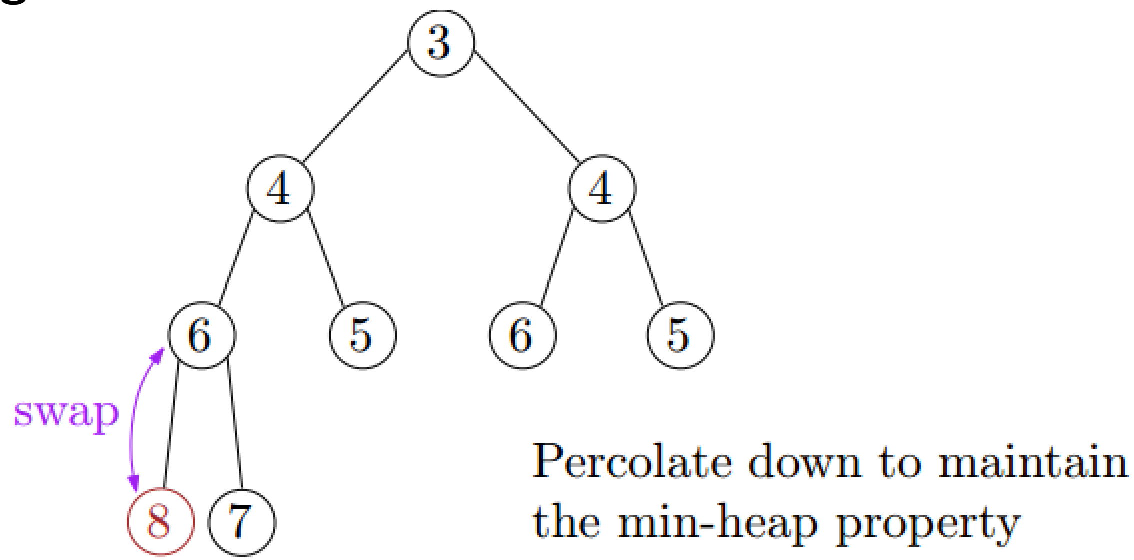
- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)

Extract-Min

- Copy the last element to the root (i.e., overwrite the minimum element stored there)
- Restore the min-heap property by percolate down (or bubble down): if the element is larger than either of its children, then interchange it with the smaller of its children.



- Correctness: after each swap, the min-heap property is satisfied for all nodes except the node containing the element (with respect to its children)
- Time complexity = O(height) = O(log n)

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

Heapsort

- Build a binary heap of n elements
  - the minimum element is at the top of the heap

Heapsort

- Build a binary heap of n elements
  - the minimum element is at the top of the heap
- Perform n Extract-Min operations
  - the elements are extracted in sorted order

Heapsort

- Build a binary heap of n elements
  - the minimum element is at the top of the heap
  - insert n elements one by one
    → O(n log n)
    (there is a more efficient way, check CLRS 6.3 if interested)

- Perform n Extract-Min operations
  - the elements are extracted in sorted order

Heapsort

- Build a binary heap of n elements
  - the minimum element is at the top of the heap
  - insert n elements one by one
    → O(n log n)
    (there is a more efficient way, check CLRS 6.3 if interested)

- Perform n Extract-Min operations
  - the elements are extracted in sorted order
  - each Extract-Min operation takes O(log n) time
    → O(n log n)

Heapsort

- Build a binary heap of n elements
  - the minimum element is at the top of the heap
  - insert n elements one by one
    → O(n log n)
    (there is a more efficient way, check CLRS 6.3 if interested)
- Perform n Extract-Min operations
  - the elements are extracted in sorted order
  - each Extract-Min operation takes O(log n) time
    → O(n log n)
- Total time complexity: O(n log n)

Heapsort - Example

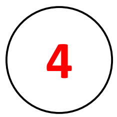
- Build a binary heap of n elements

4	1	3	2	16	9	10	14	8	7

Heapsort - Example

- Build a binary heap of n elements

4	1	3	2	16	9	10	14	8	7



Heapsort - Example

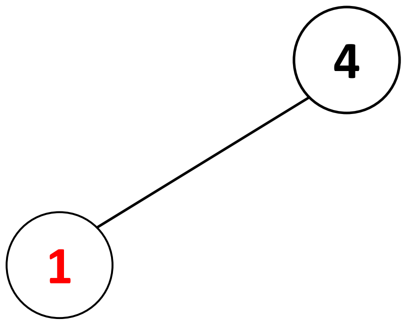
- Build a binary heap of n elements

4	1	3	2	16	9	10	14	8	7

Heapsort - Example

- Build a binary heap of n elements

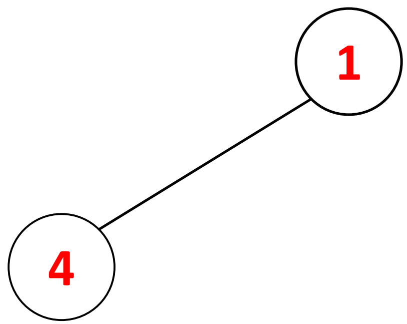
4	1	3	2	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

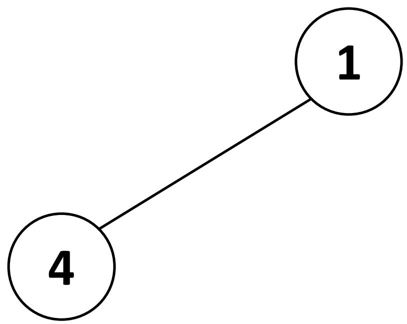
1	4	3	2	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

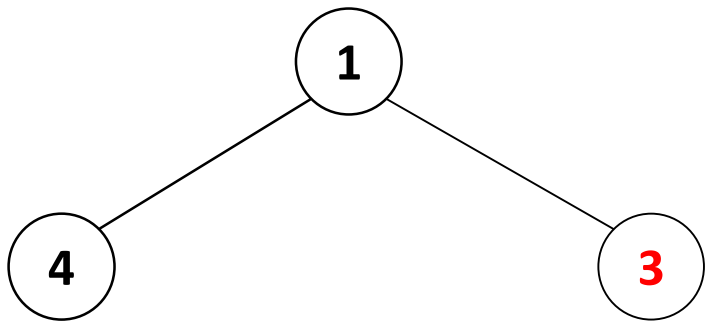
1	4	3	2	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

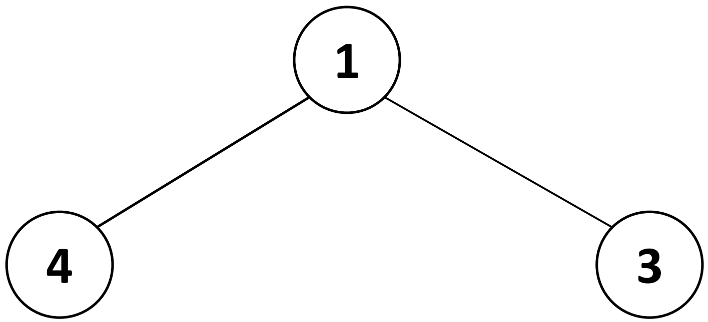
1	4	3	2	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

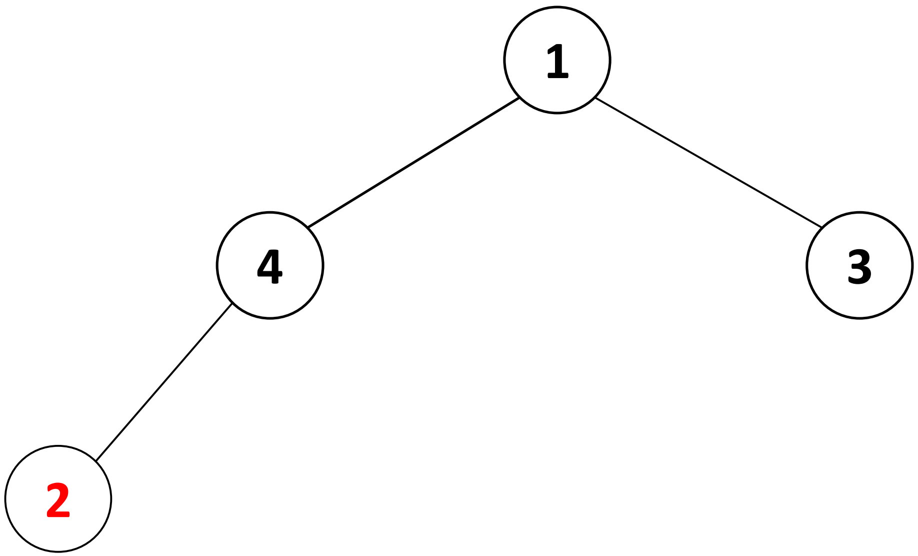
1	4	3	2	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

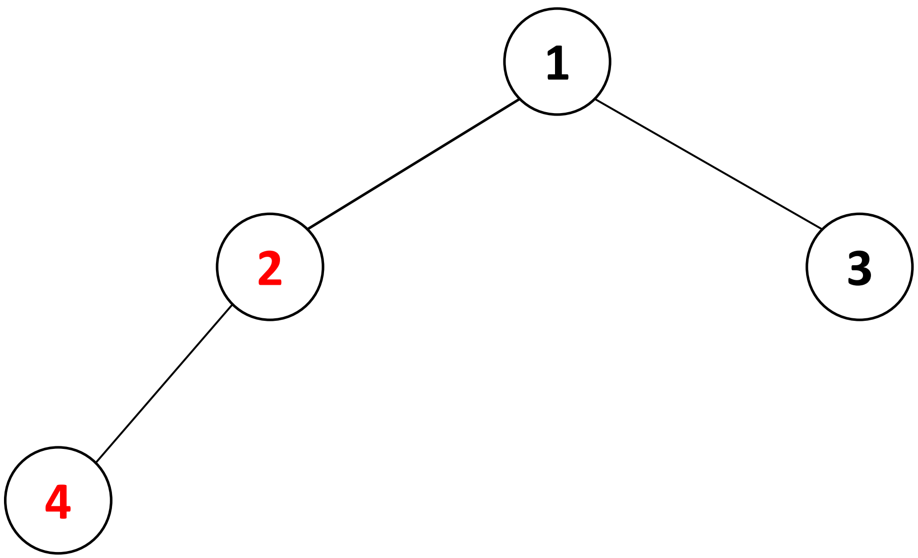
1	4	3	2	16	9	10	14	8	7



Heapsort - Example

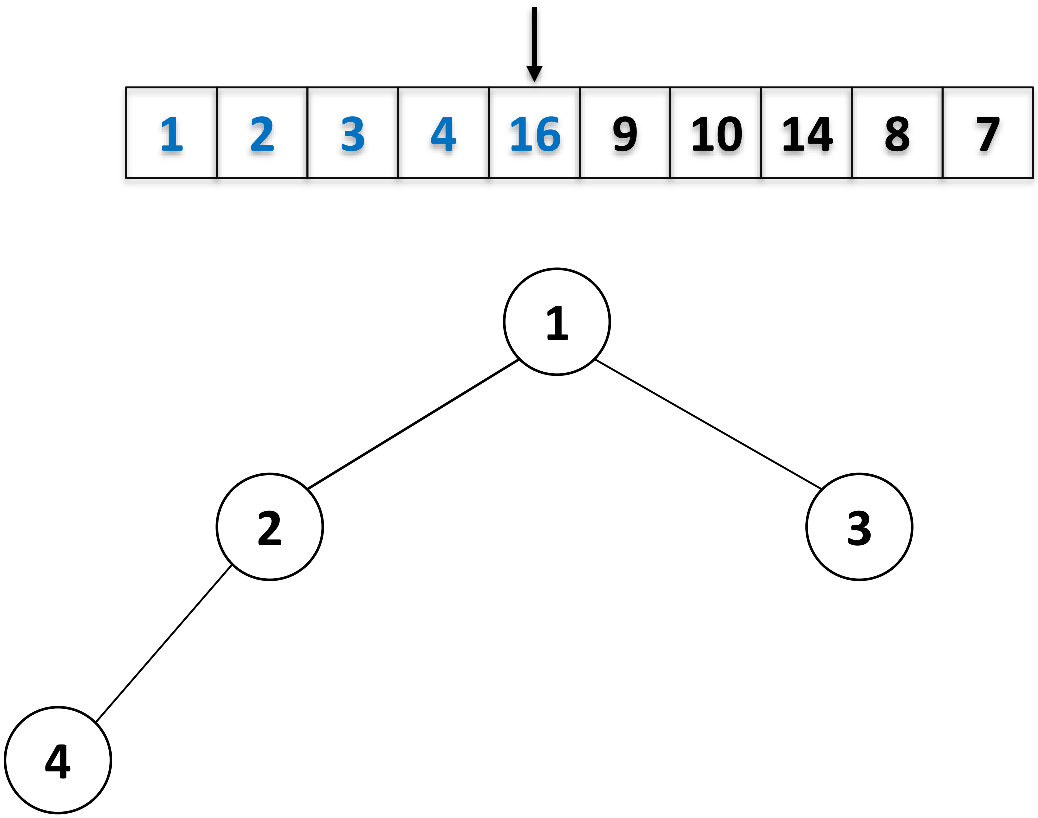
- Build a binary heap of n elements

1	2	3	4	16	9	10	14	8	7



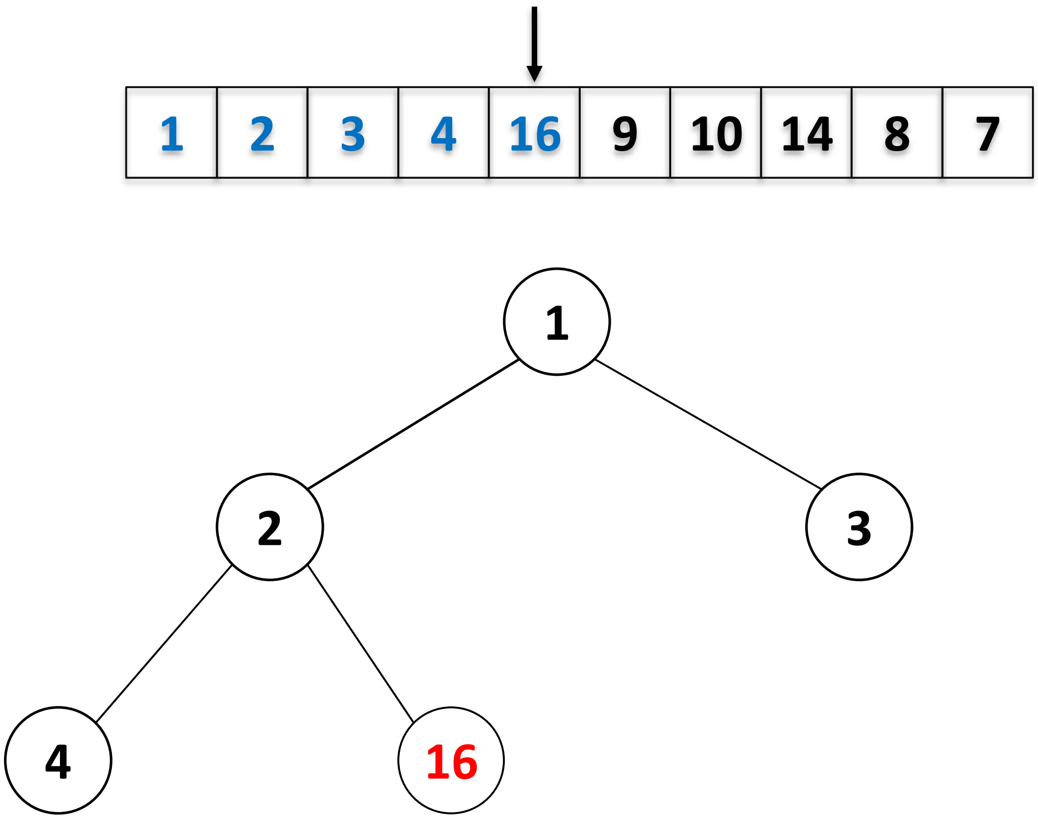
Heapsort - Example

- Build a binary heap of n elements



Heapsort - Example

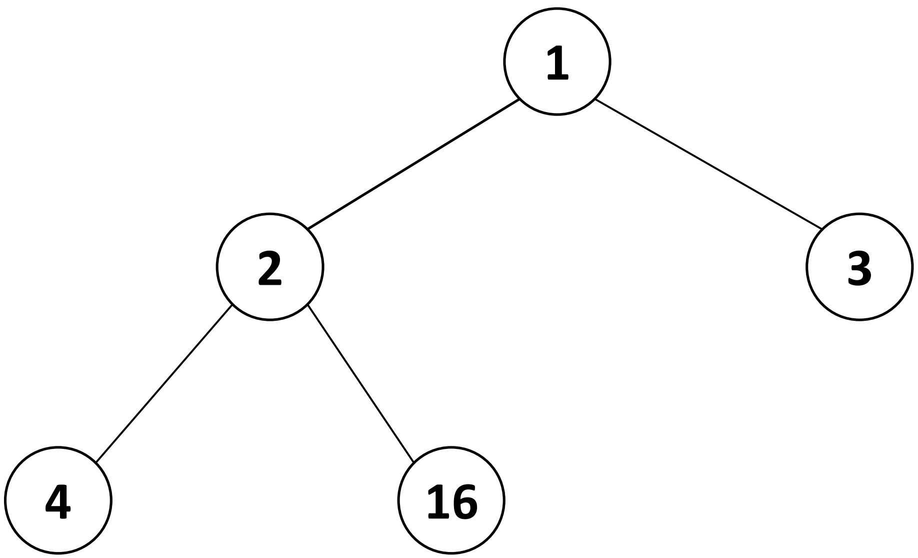
- Build a binary heap of n elements



Heapsort - Example

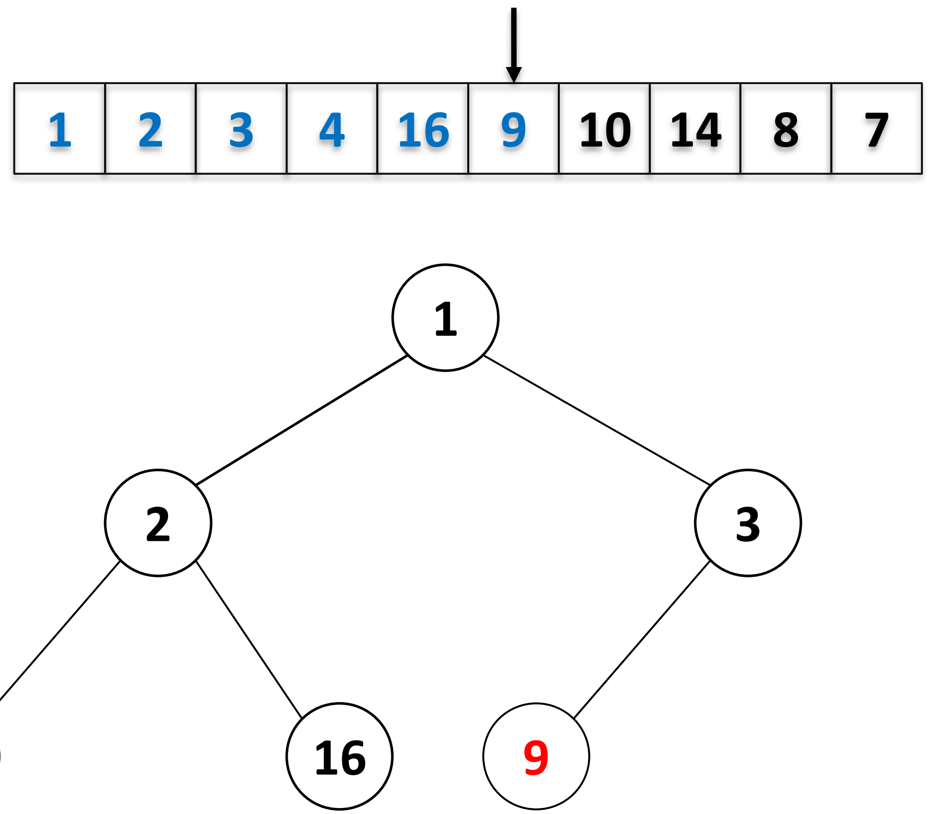
- Build a binary heap of n elements

1	2	3	4	16	9	10	14	8	7



Heapsort - Example

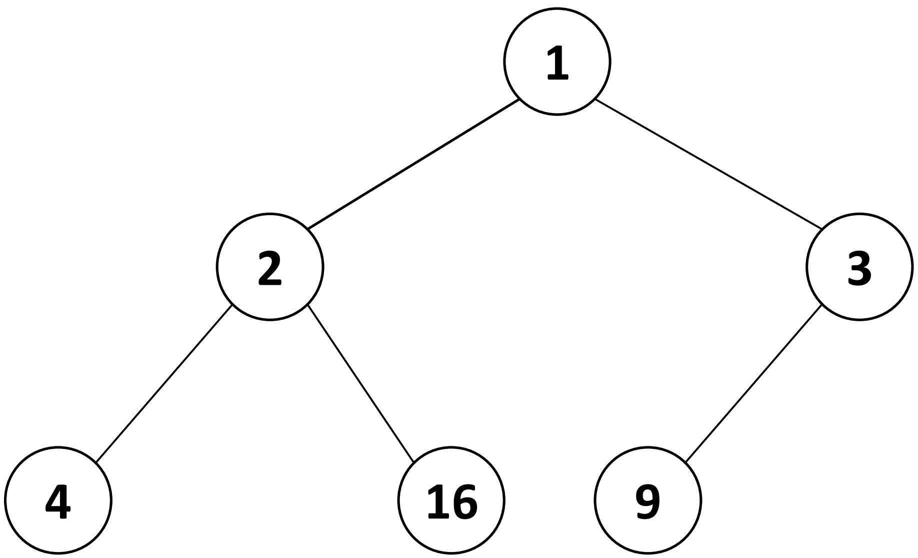
- Build a binary heap of n elements



Heapsort - Example

- Build a binary heap of n elements

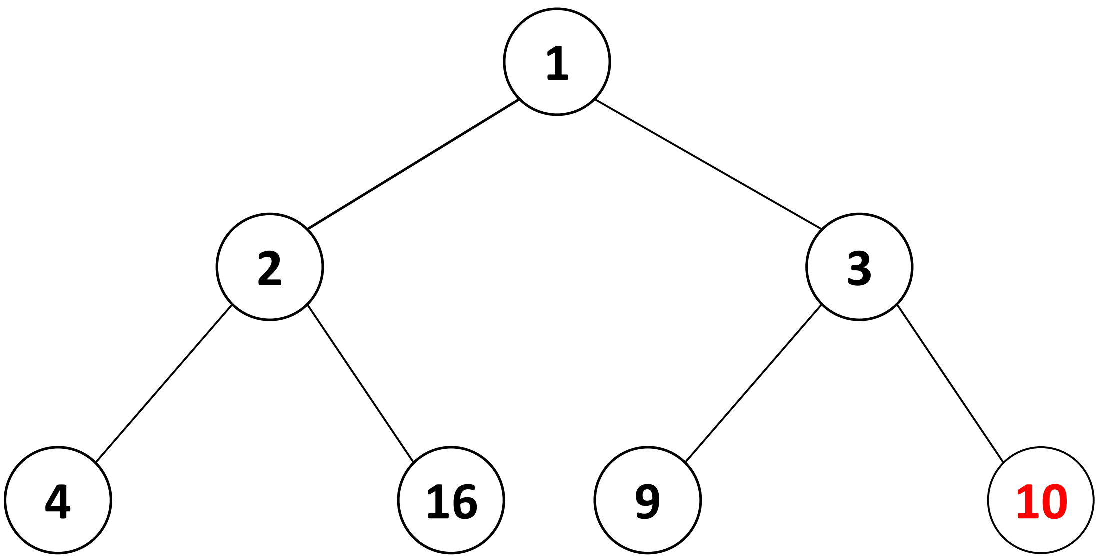
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

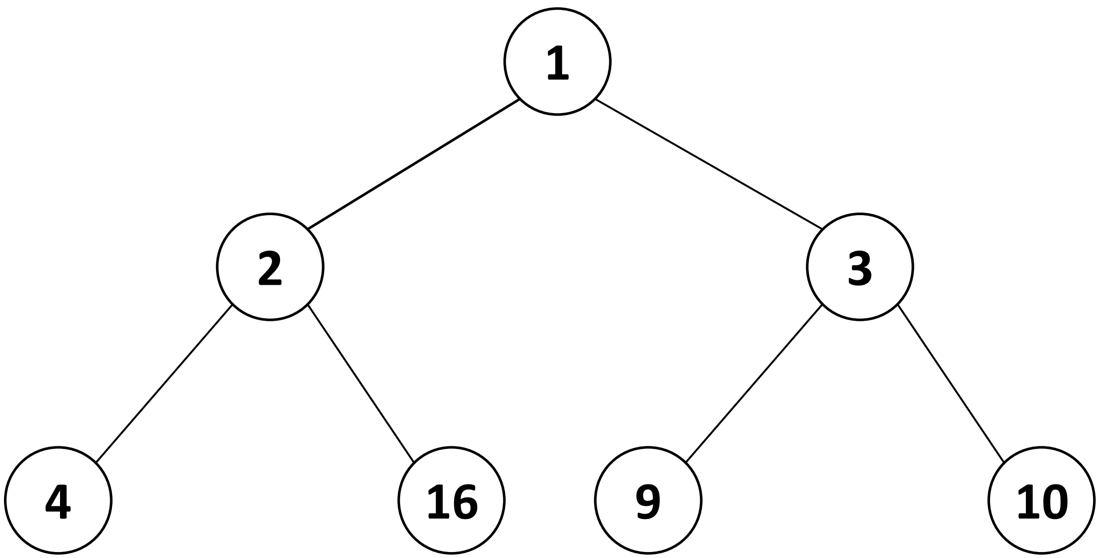
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

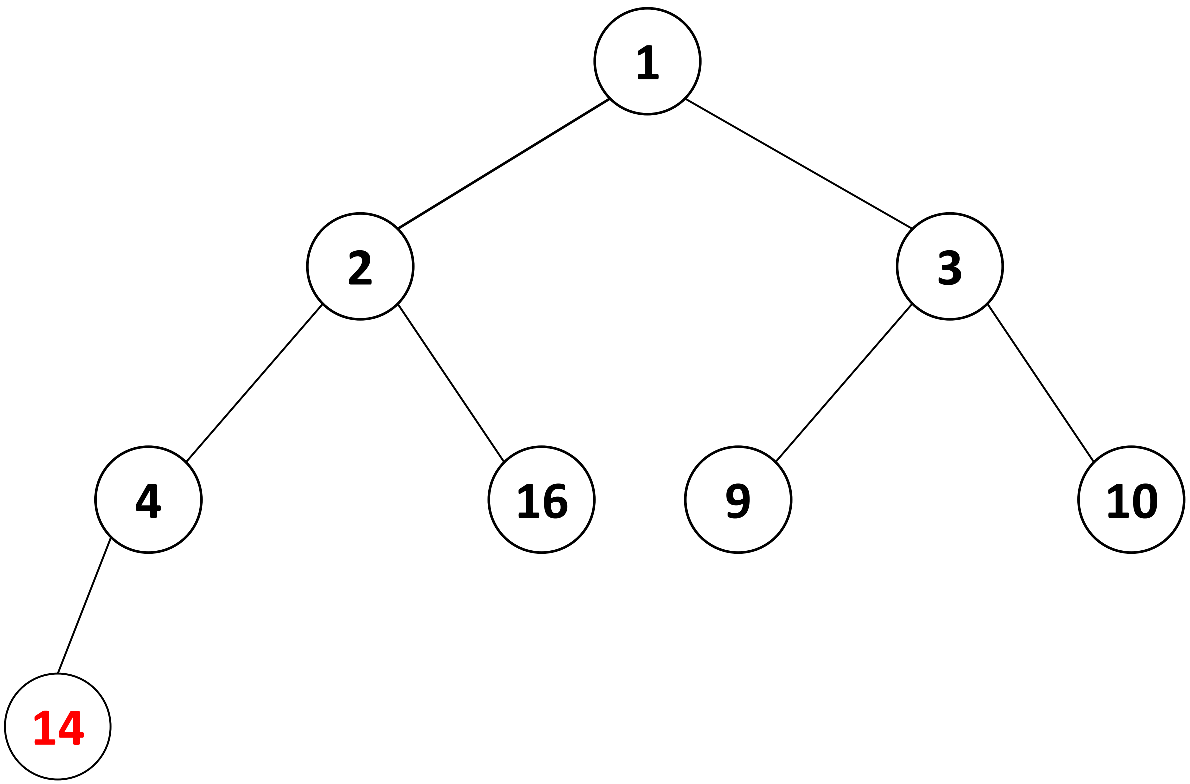
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

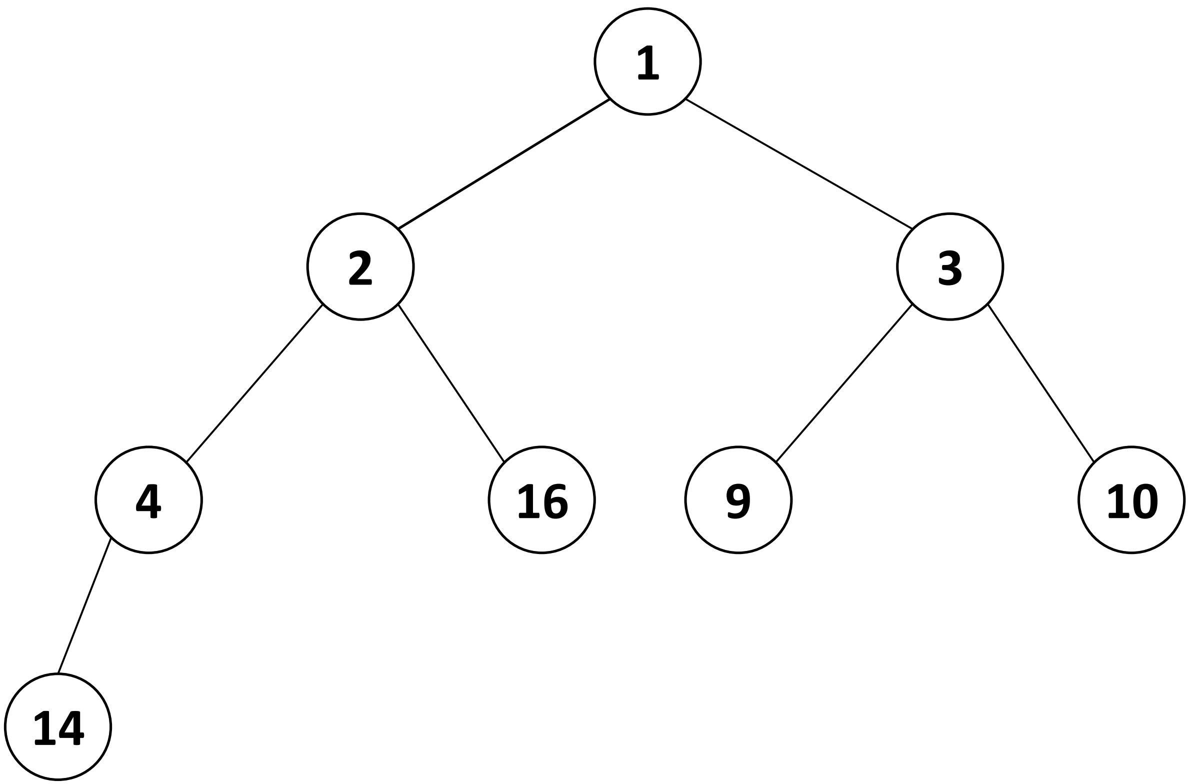
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

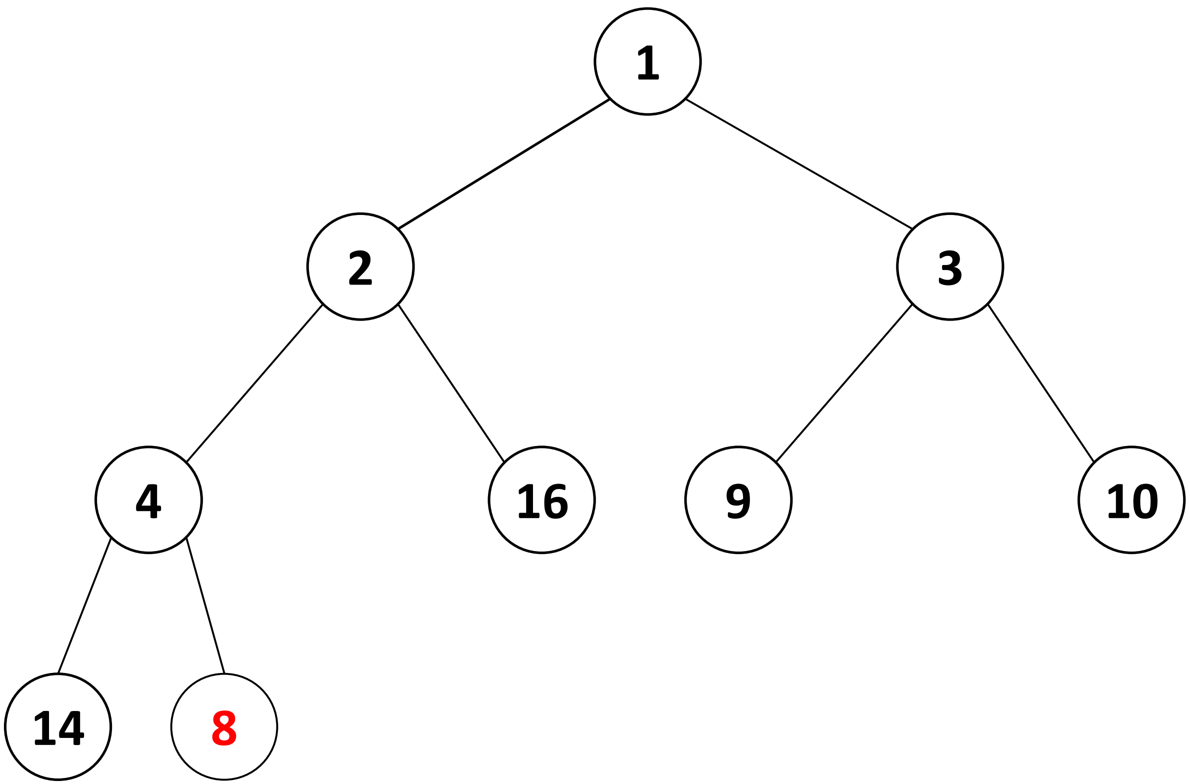
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

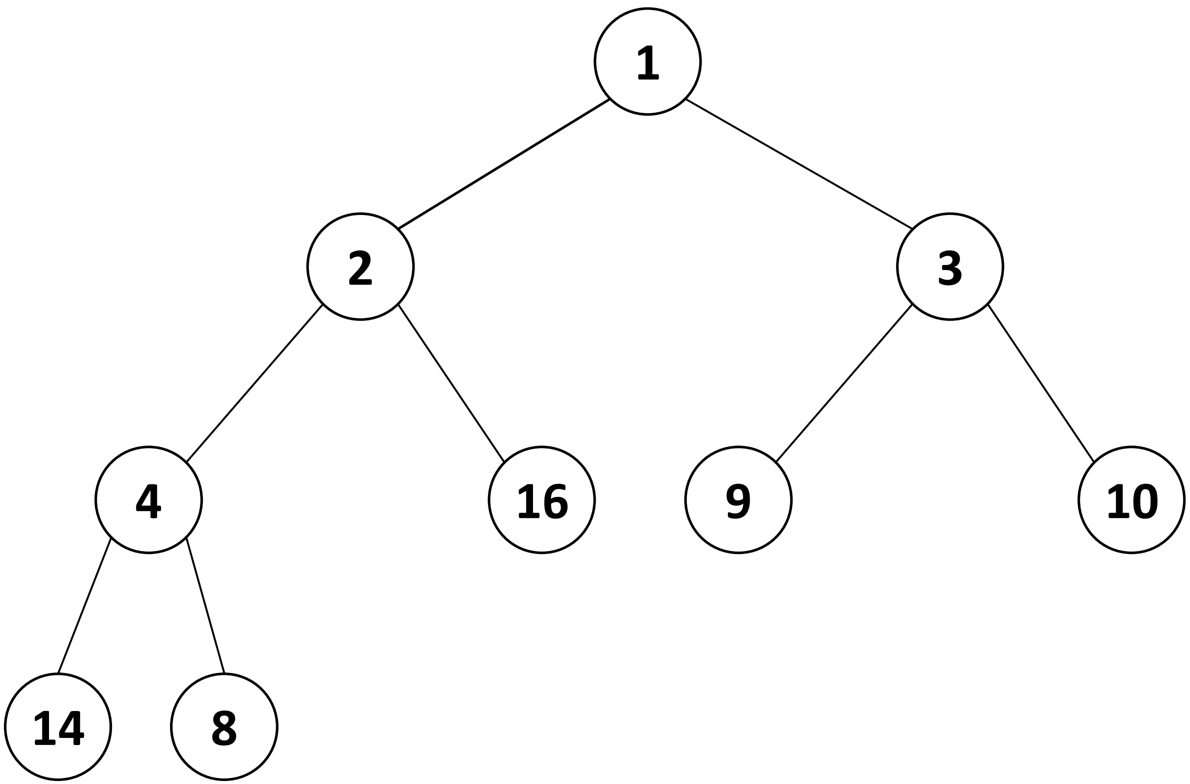
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

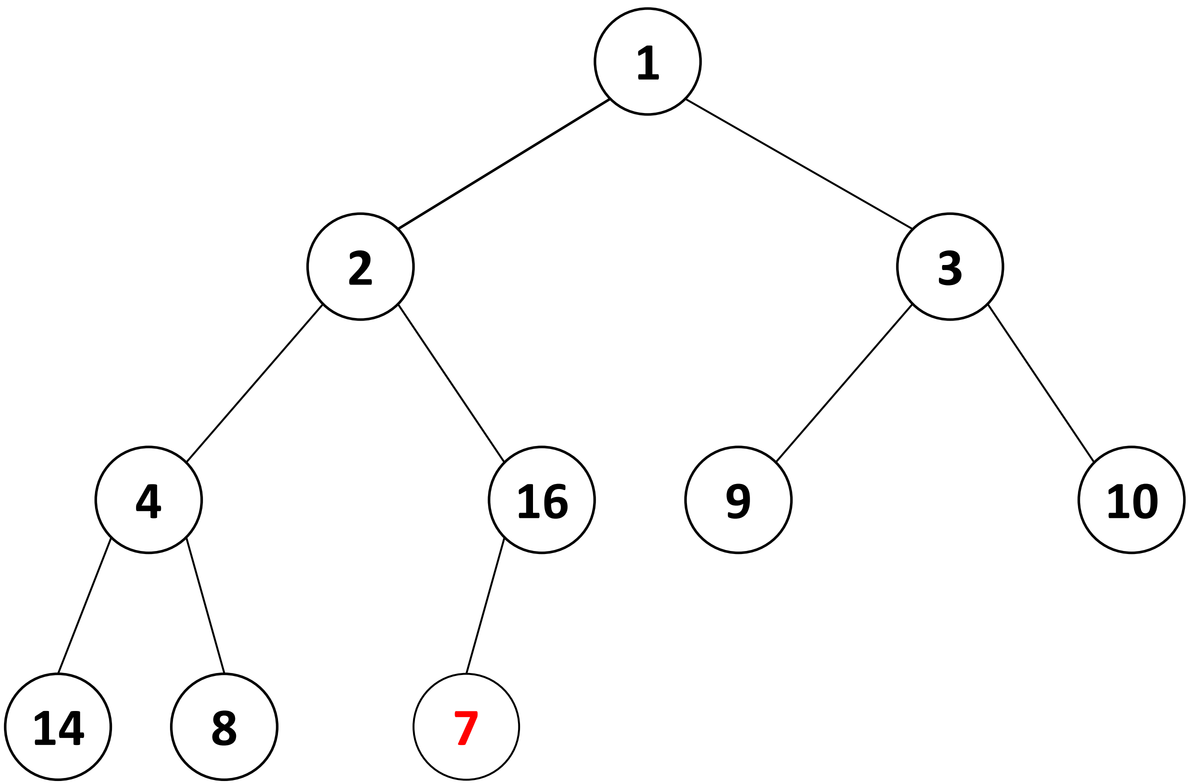
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

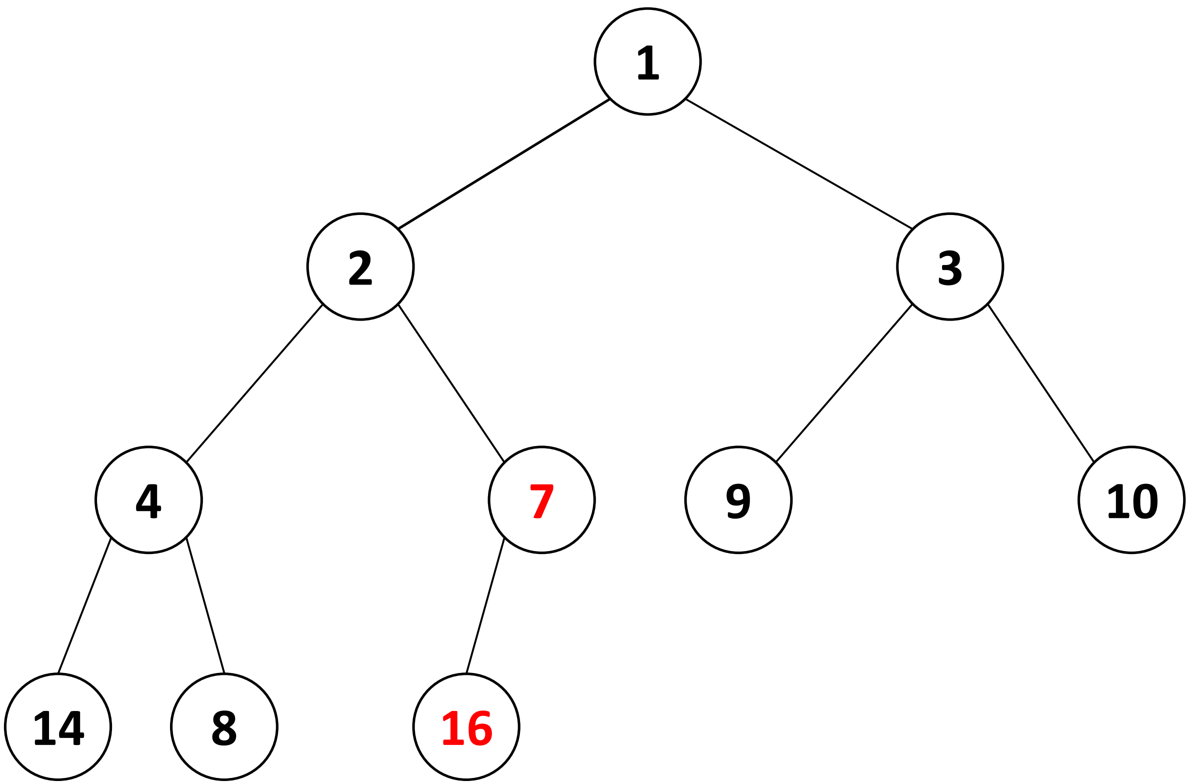
1	2	3	4	16	9	10	14	8	7



Heapsort - Example

- Build a binary heap of n elements

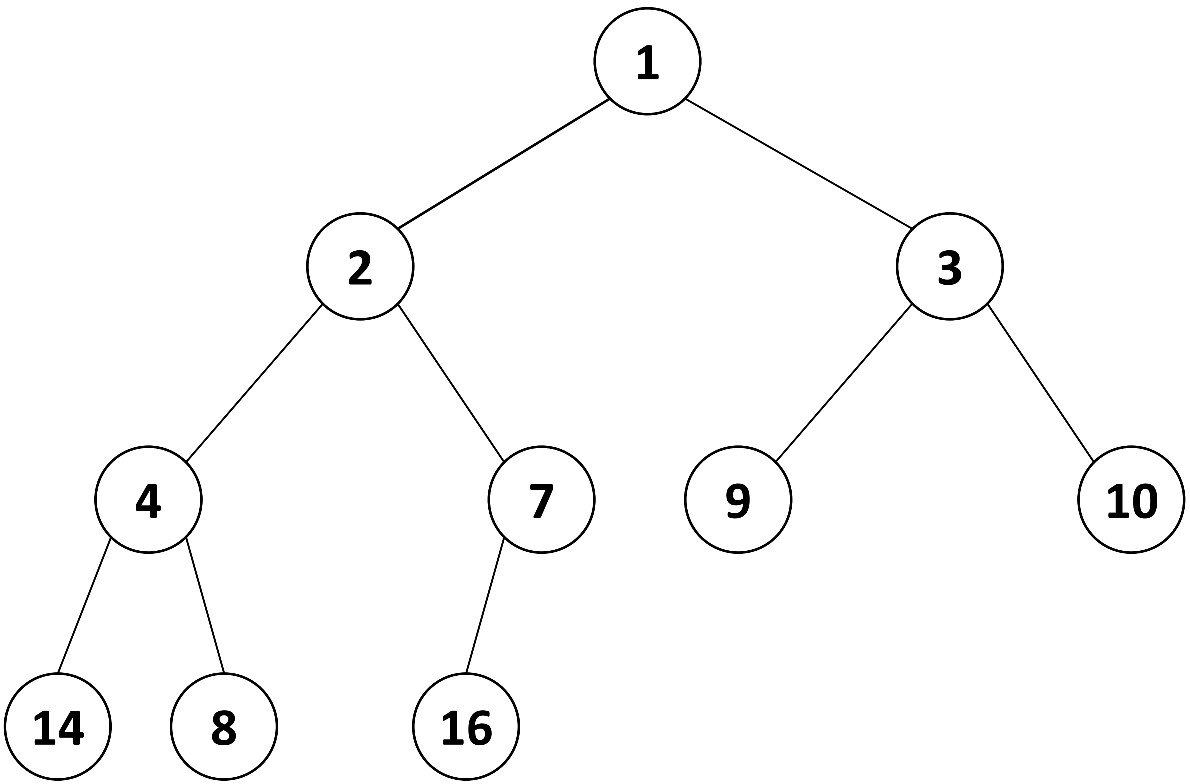
1	2	3	4	7	9	10	14	8	16



Heapsort - Example

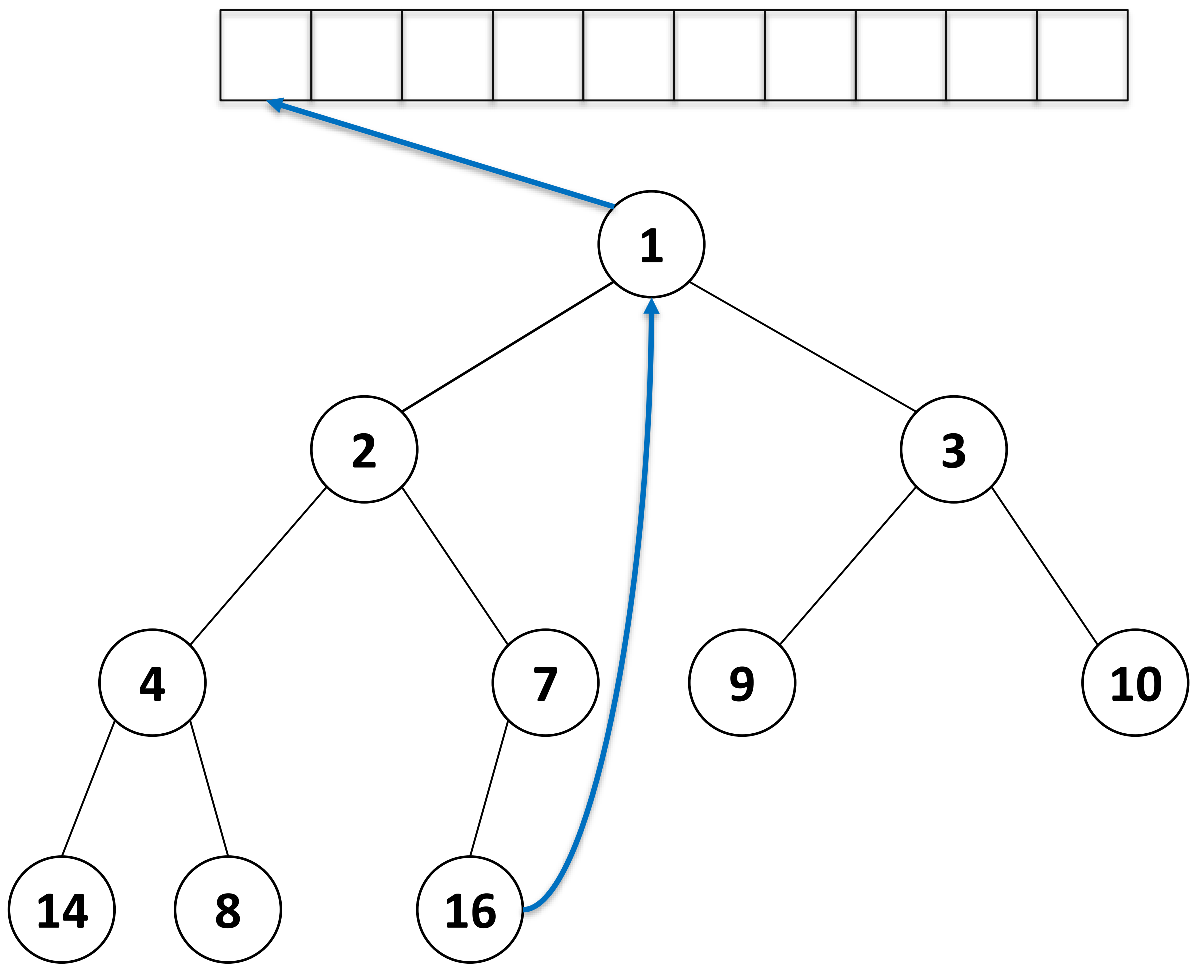
- Build a binary heap of n elements

1	2	3	4	7	9	10	14	8	16



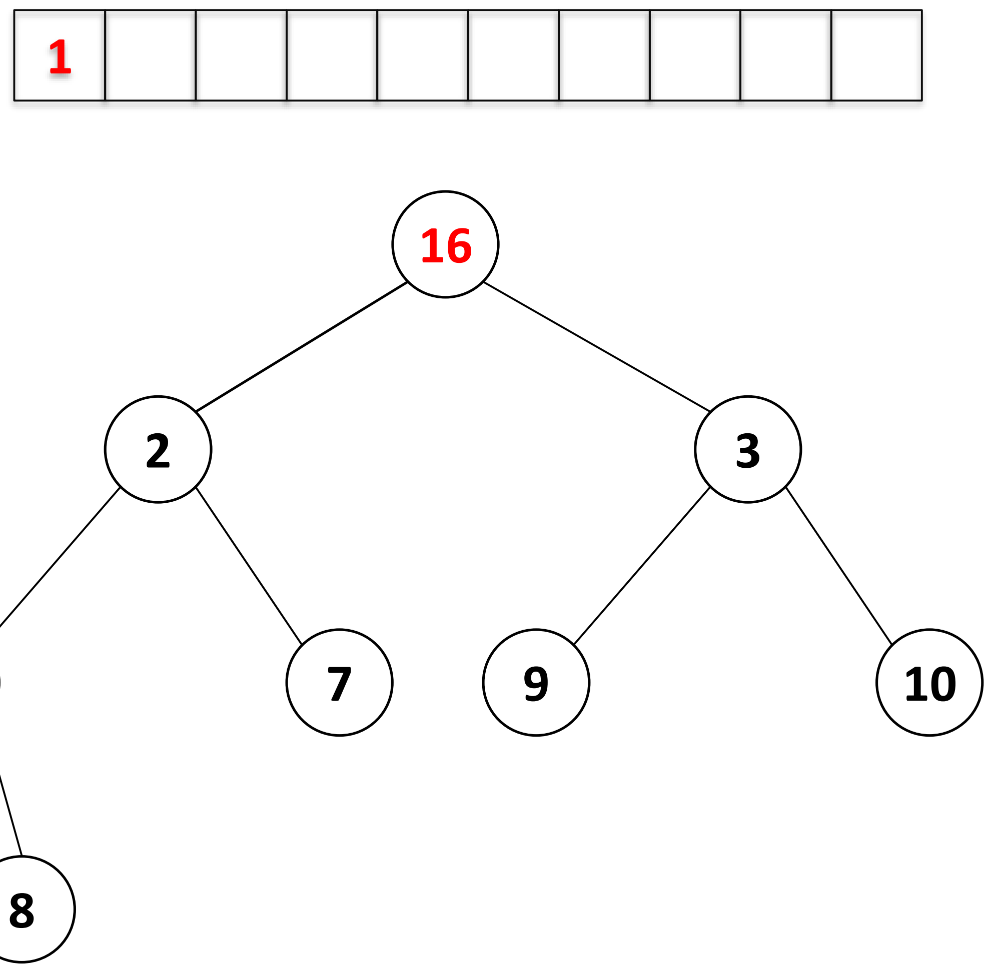
Heapsort - Example

- Perform n Extract-Min operations



Heapsort - Example

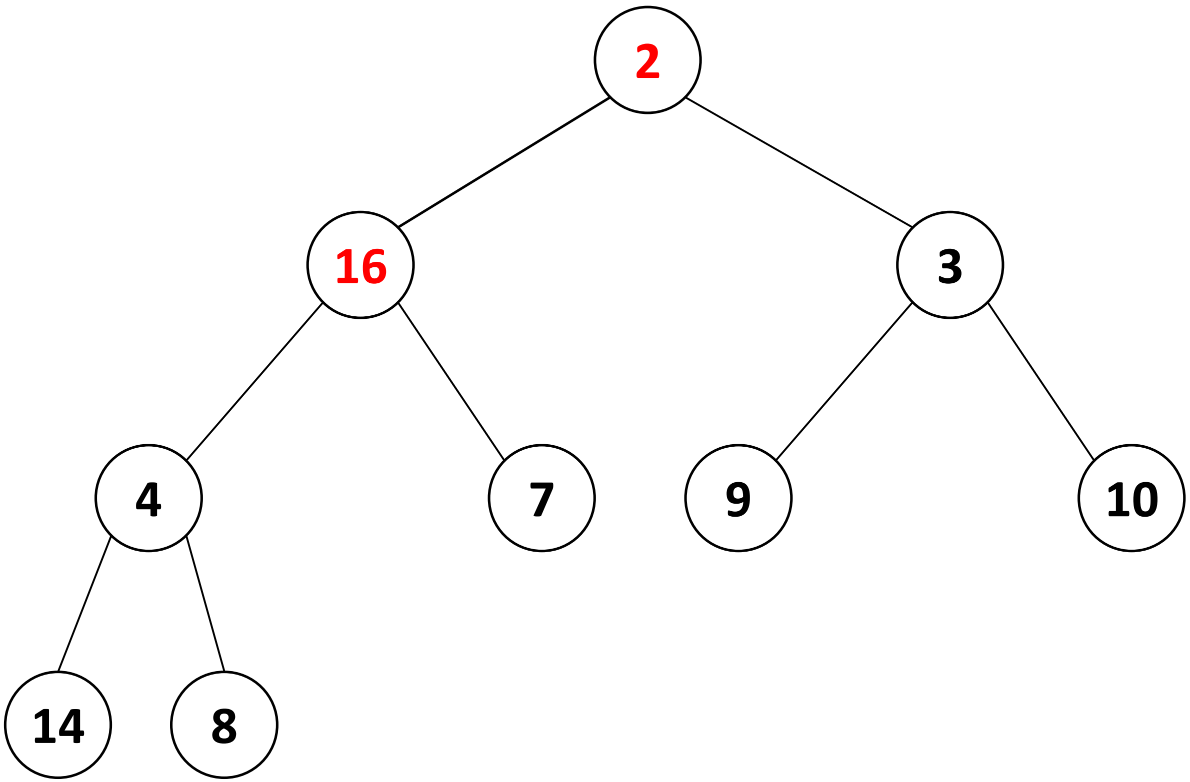
- Perform n Extract-Min operations



Heapsort - Example

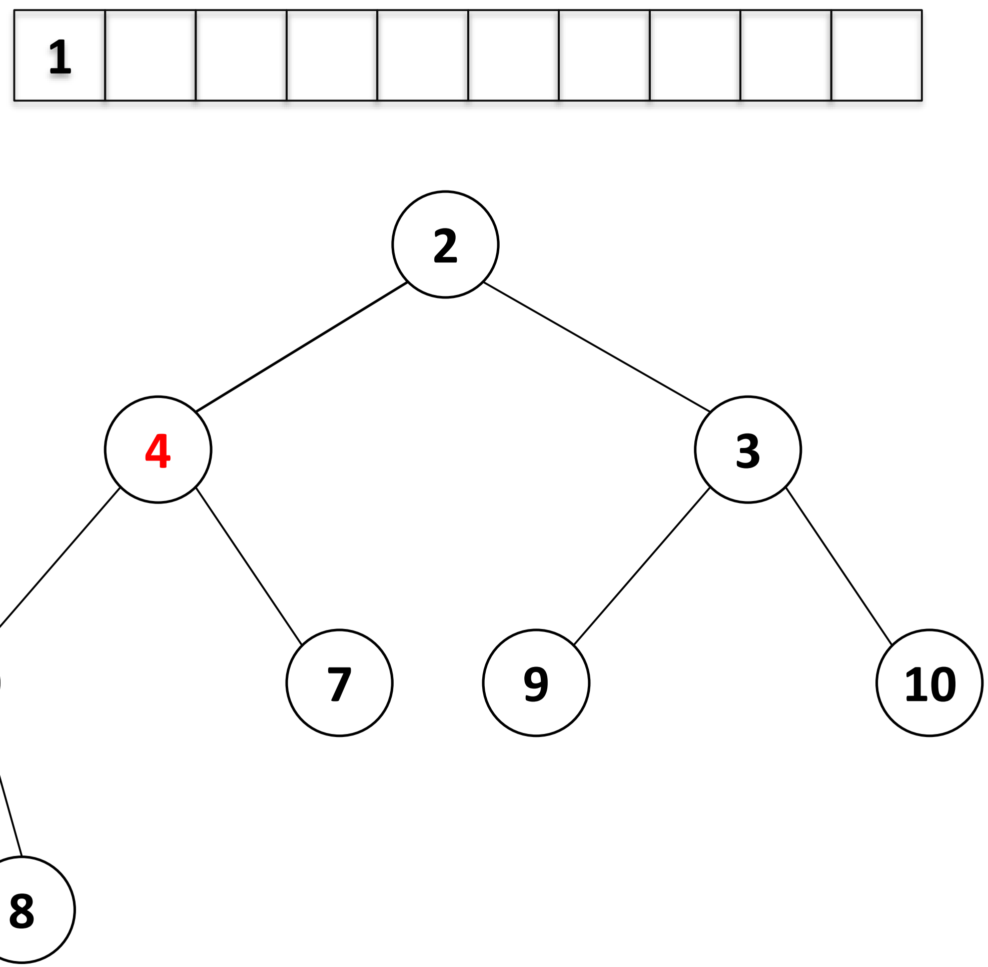
- Perform n Extract-Min operations

1									



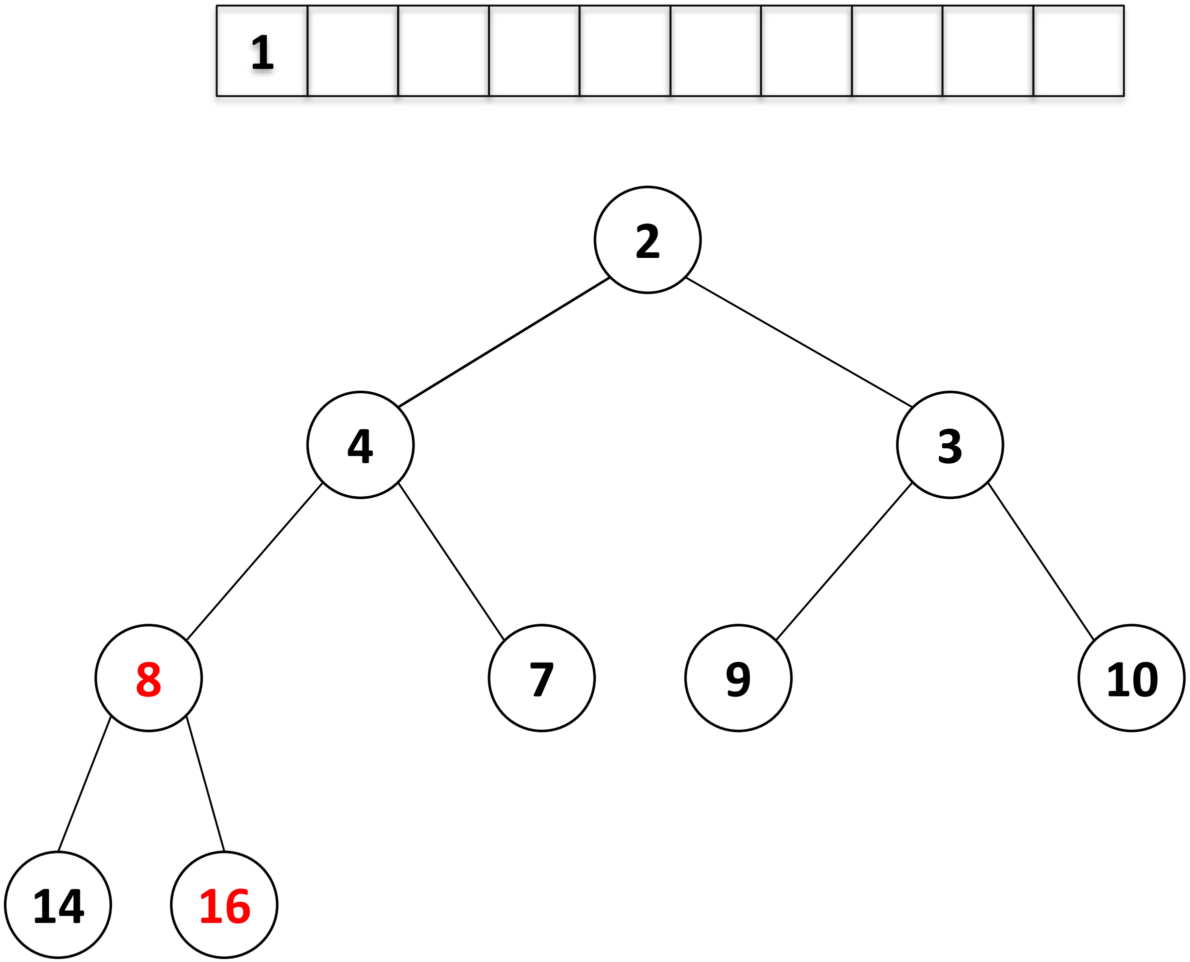
Heapsort - Example

- Perform n Extract-Min operations



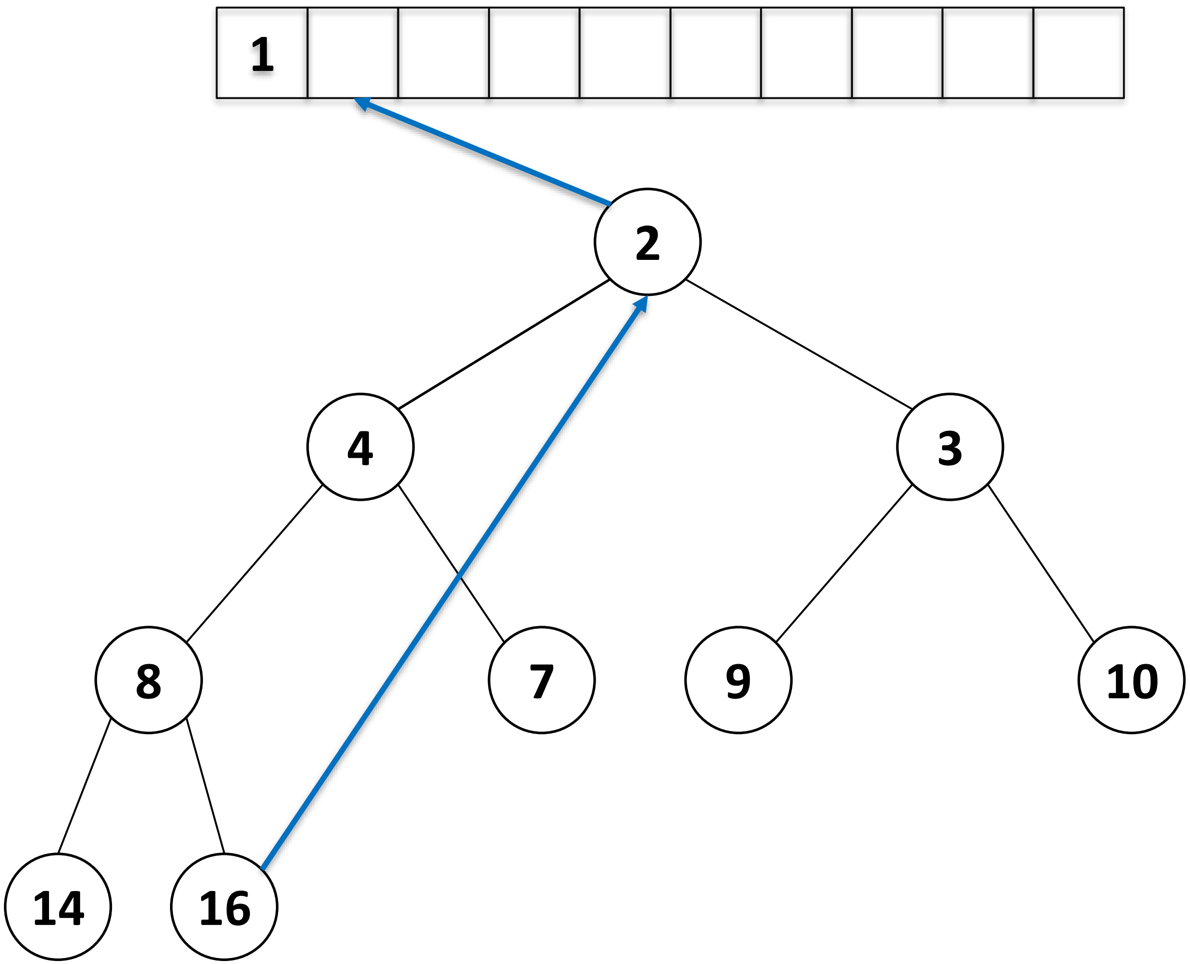
Heapsort - Example

- Perform n Extract-Min operations



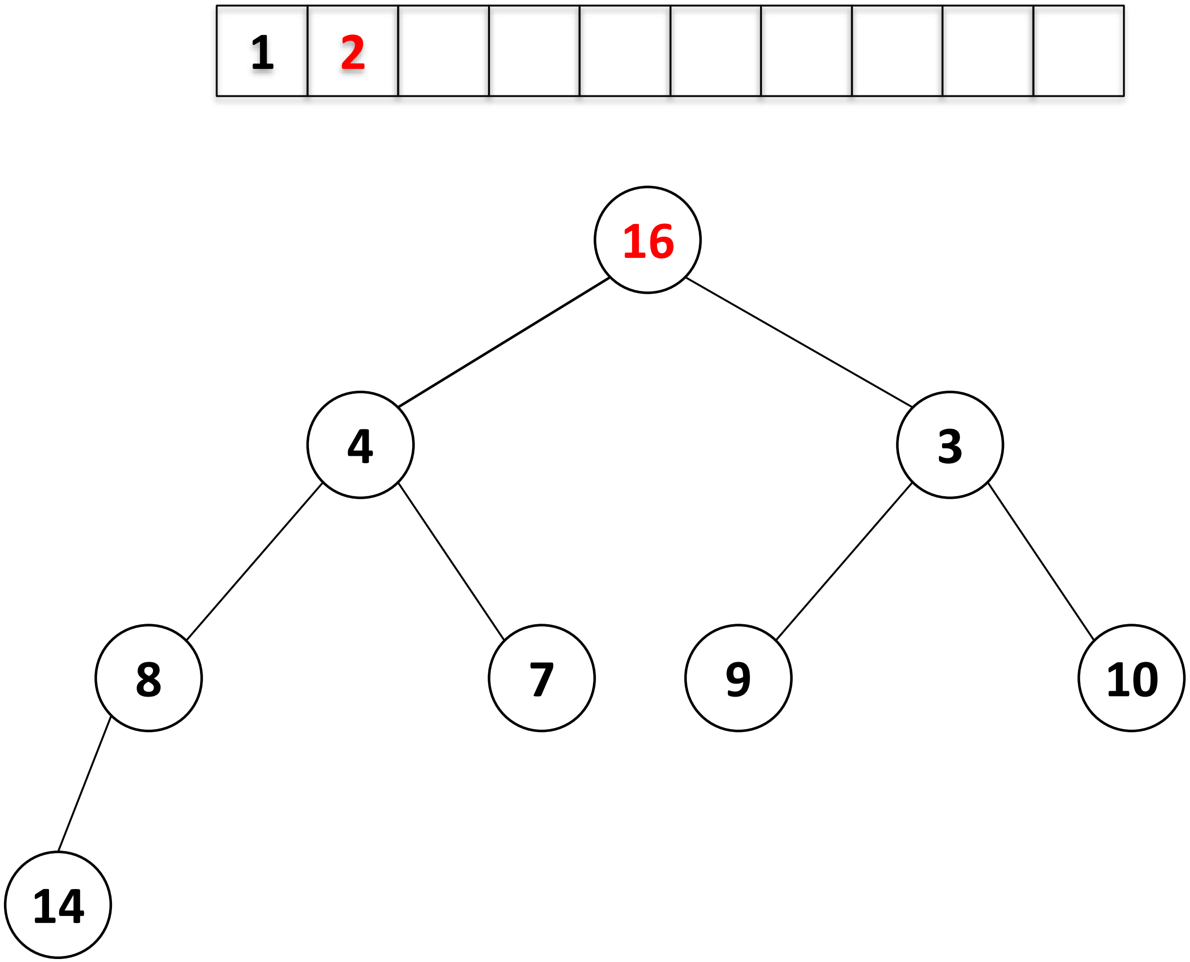
Heapsort - Example

- Perform n Extract-Min operations



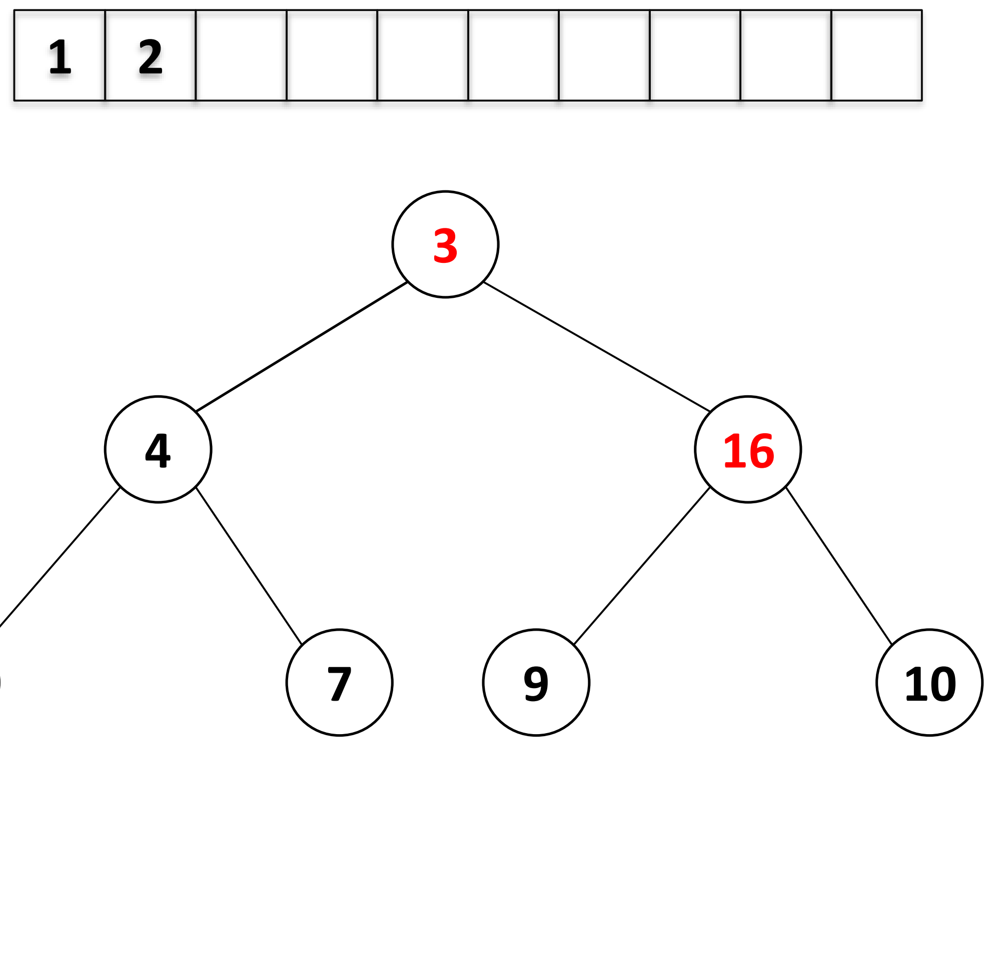
Heapsort - Example

- Perform n Extract-Min operations



Heapsort - Example

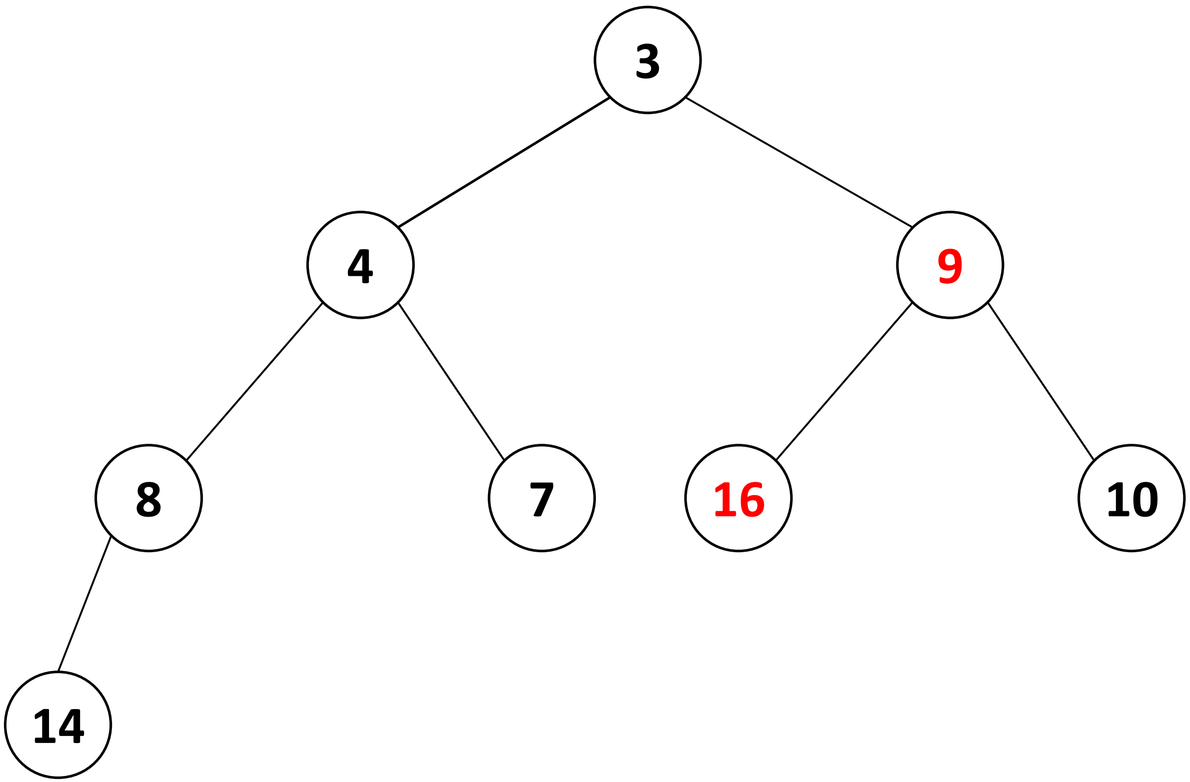
- Perform n Extract-Min operations



Heapsort - Example

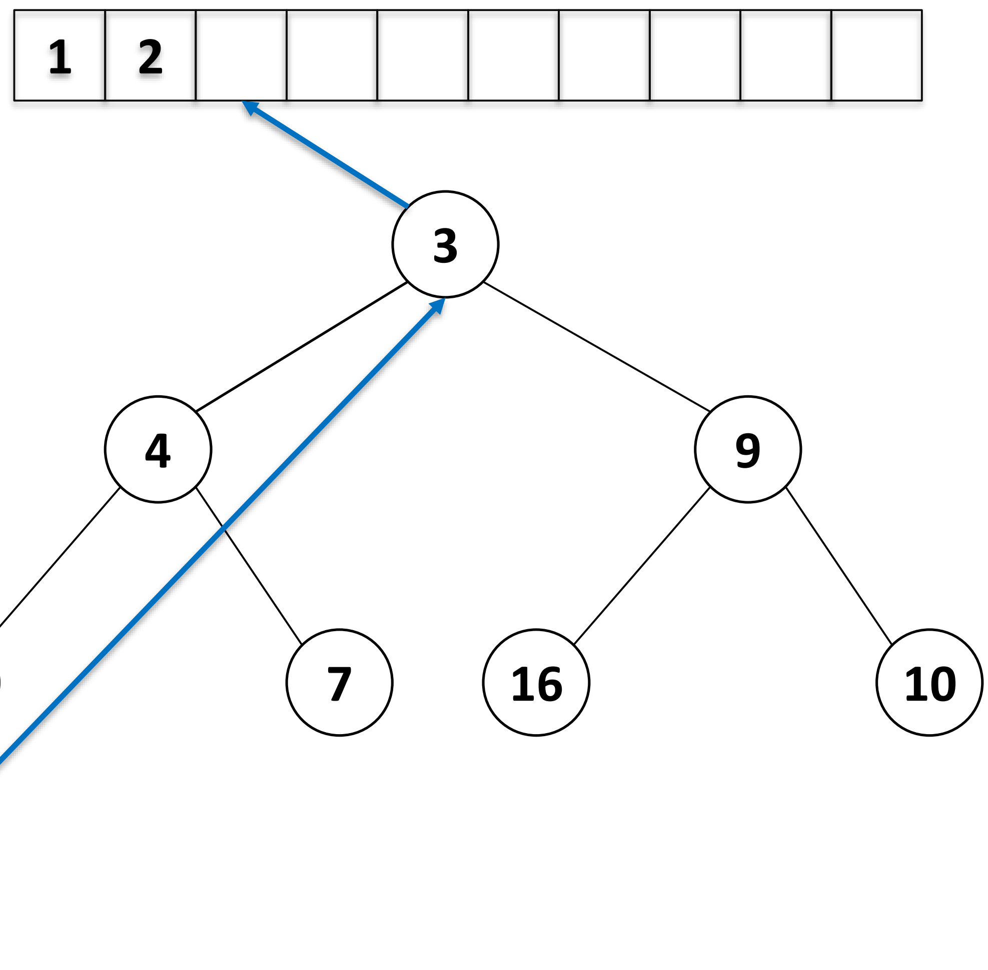
- Perform n Extract-Min operations

1	2								



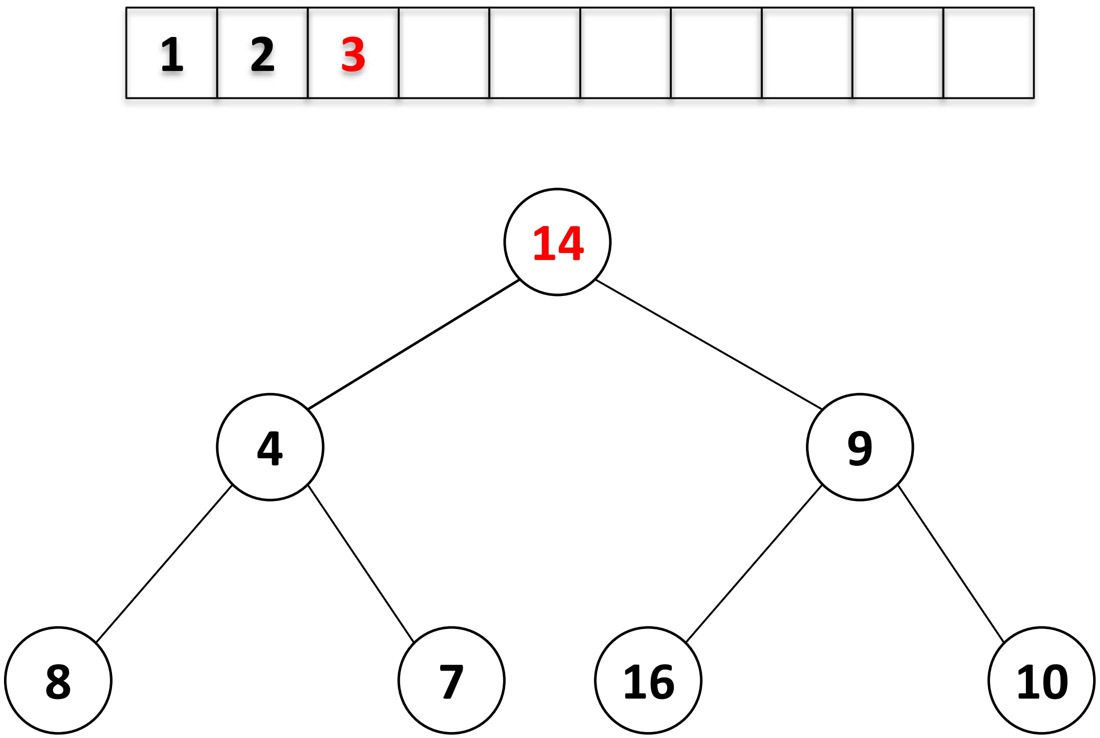
Heapsort - Example

- Perform n Extract-Min operations



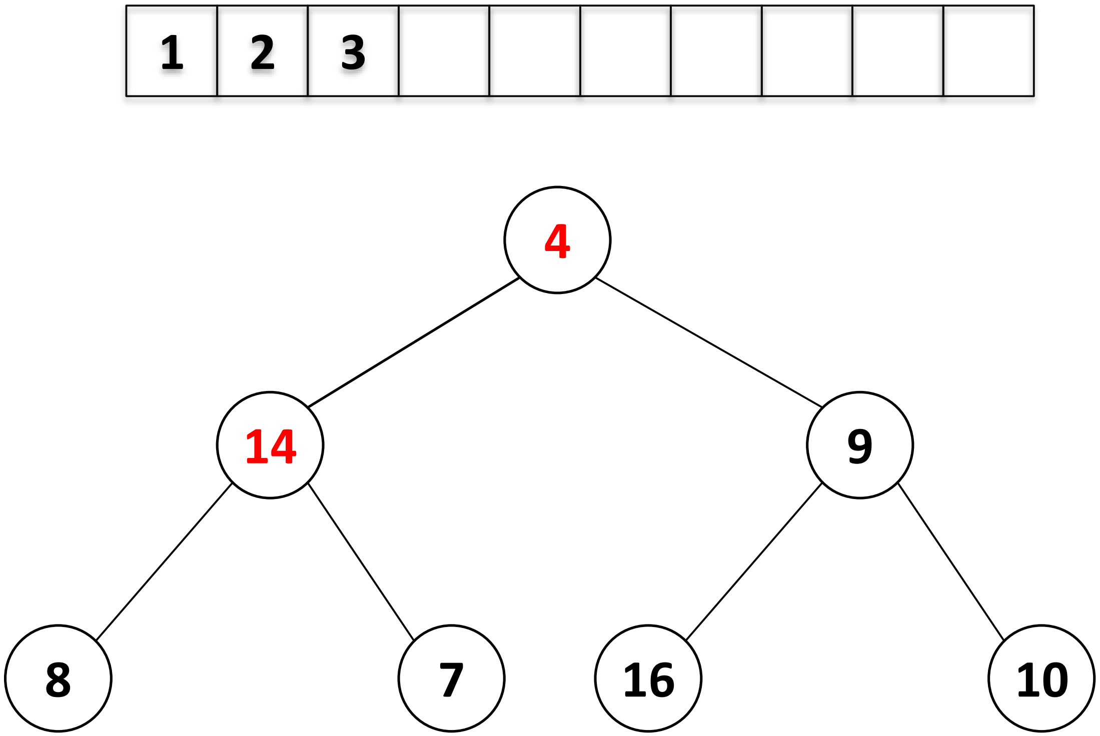
Heapsort - Example

- Perform n Extract-Min operations



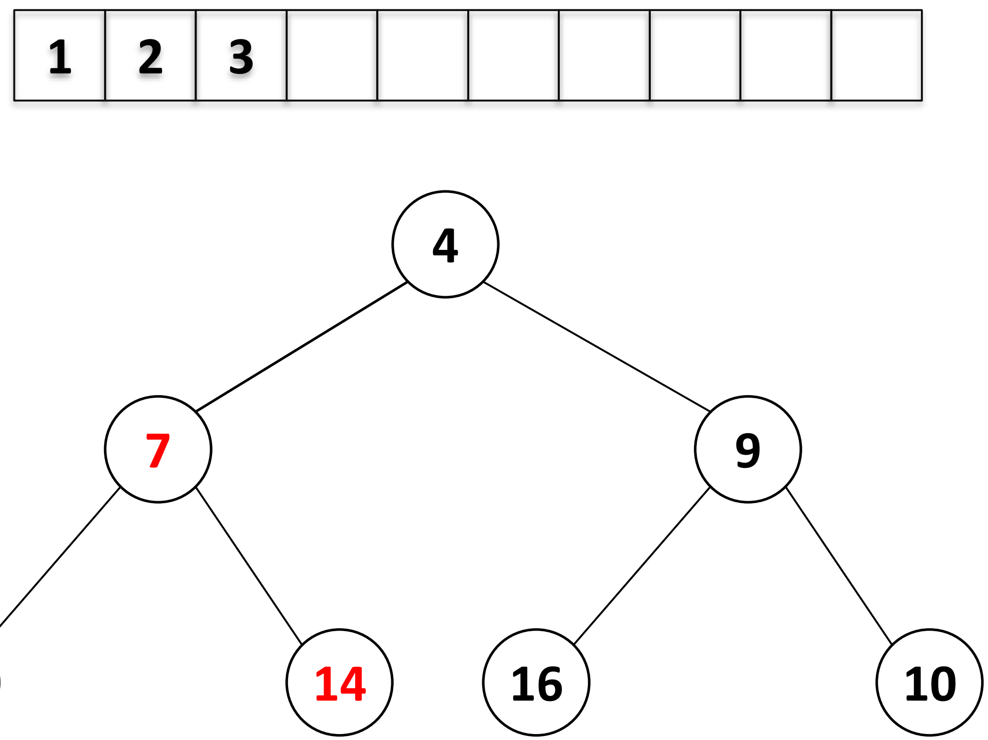
Heapsort - Example

- Perform n Extract-Min operations



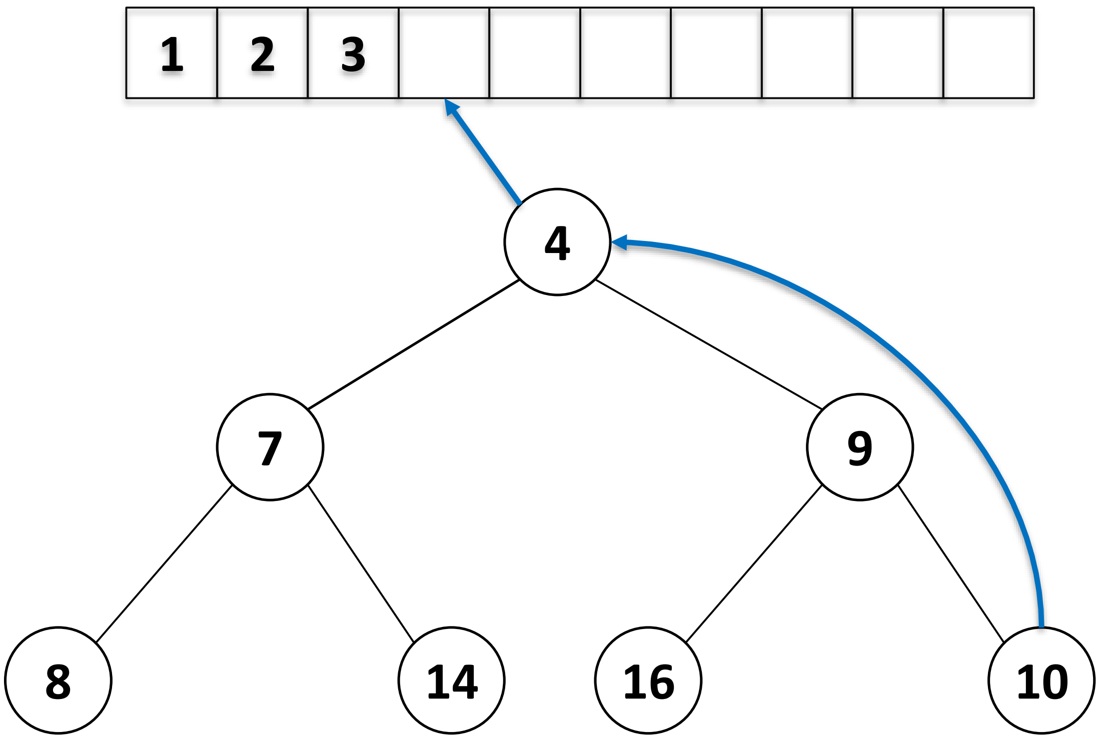
Heapsort - Example

- Perform n Extract-Min operations



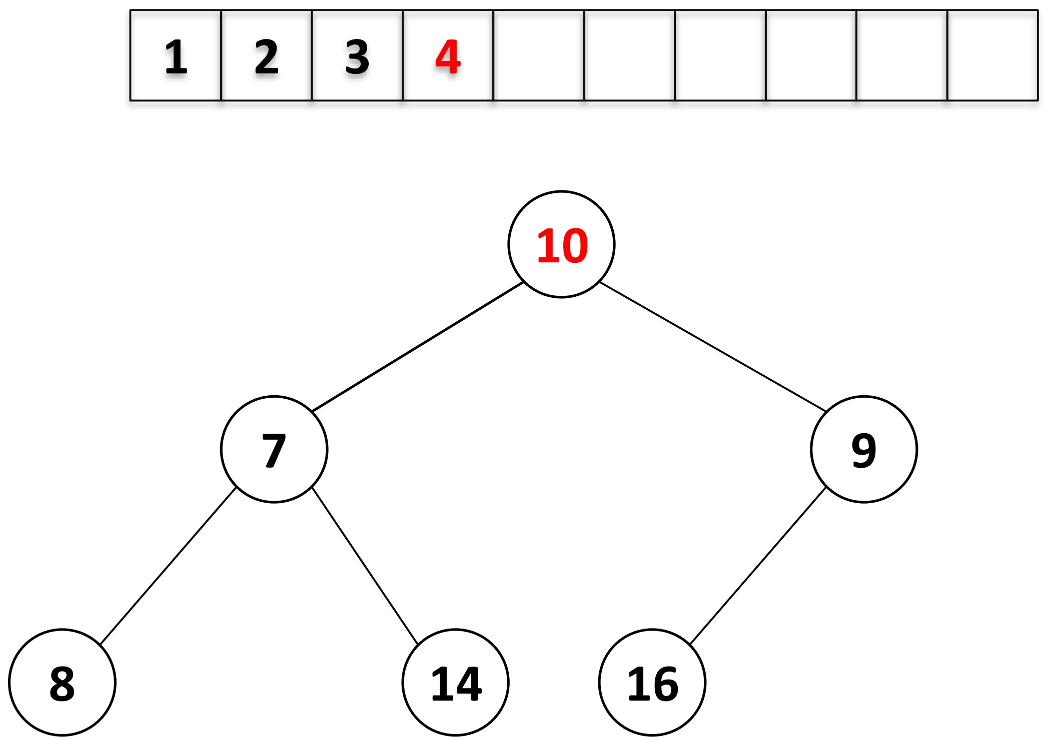
Heapsort - Example

- Perform n Extract-Min operations



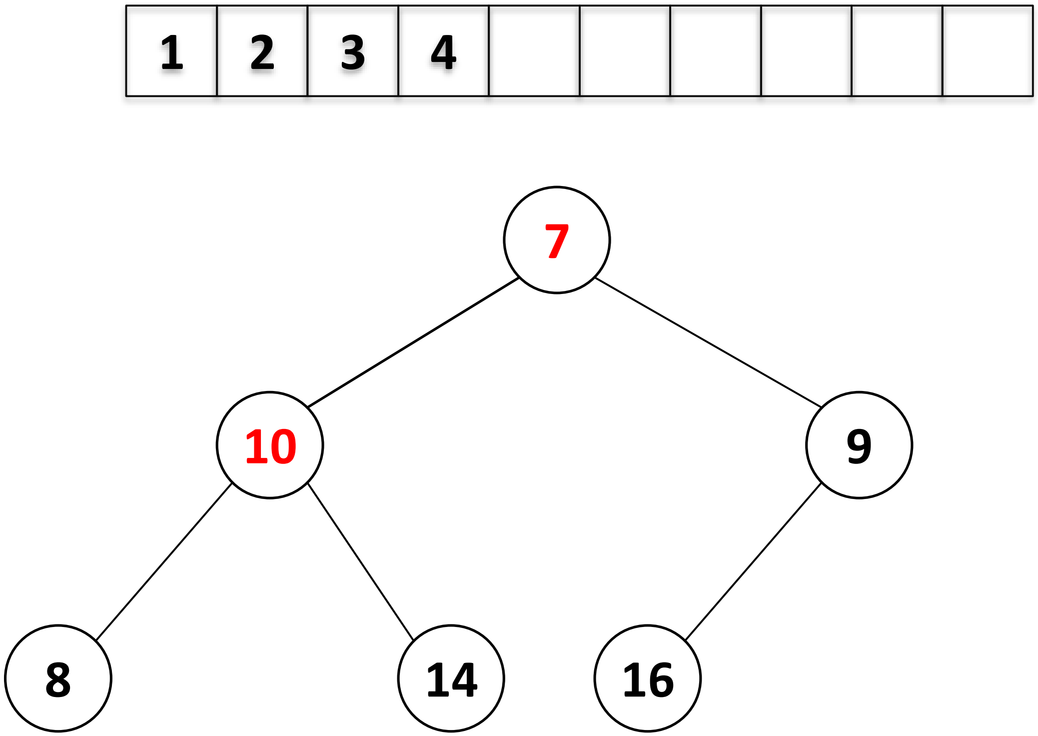
Heapsort - Example

- Perform n Extract-Min operations



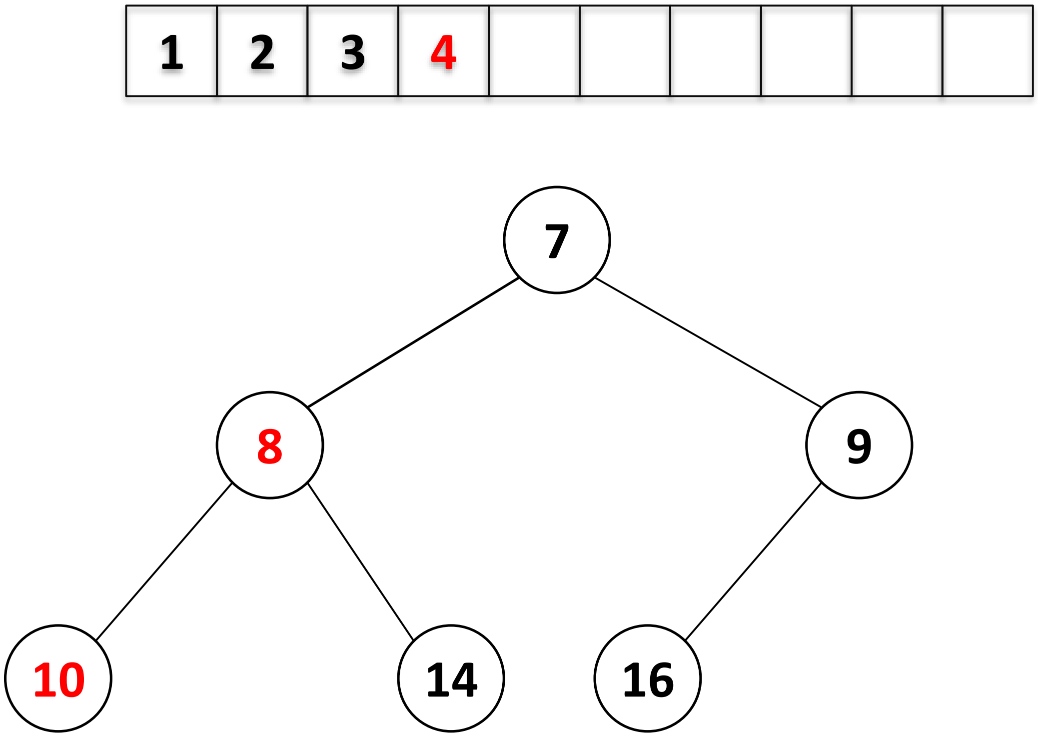
Heapsort - Example

- Perform n Extract-Min operations



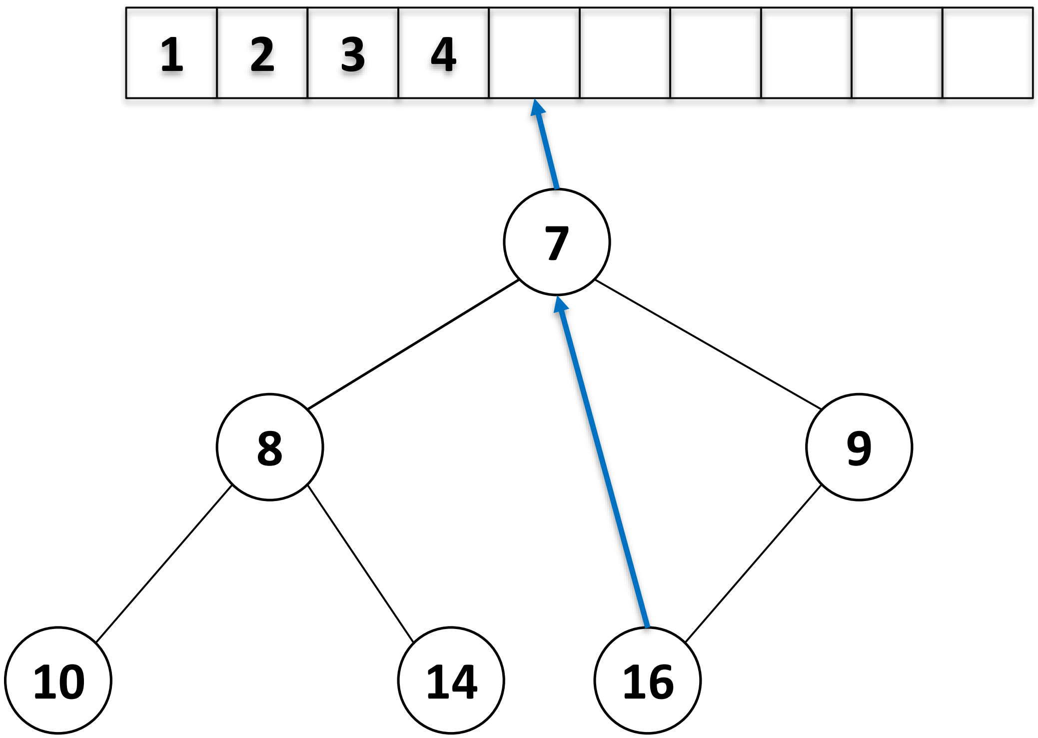
Heapsort - Example

- Perform n Extract-Min operations



Heapsort - Example

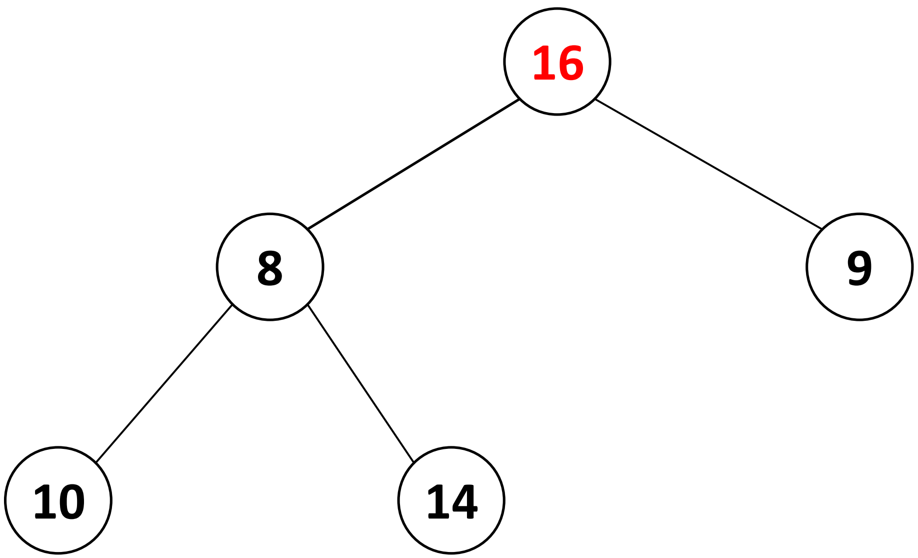
- Perform n Extract-Min operations



Heapsort - Example

- Perform n Extract-Min operations

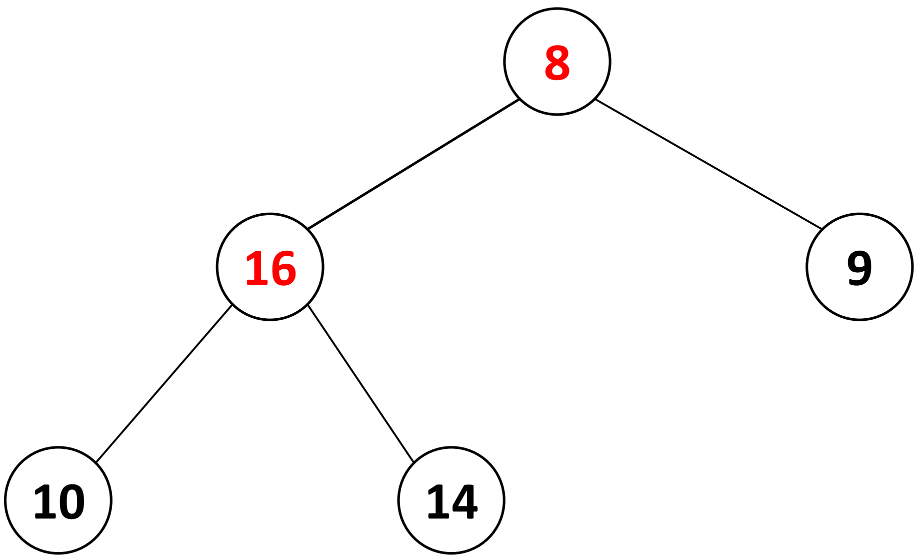
1	2	3	4	7					



Heapsort - Example

- Perform n Extract-Min operations

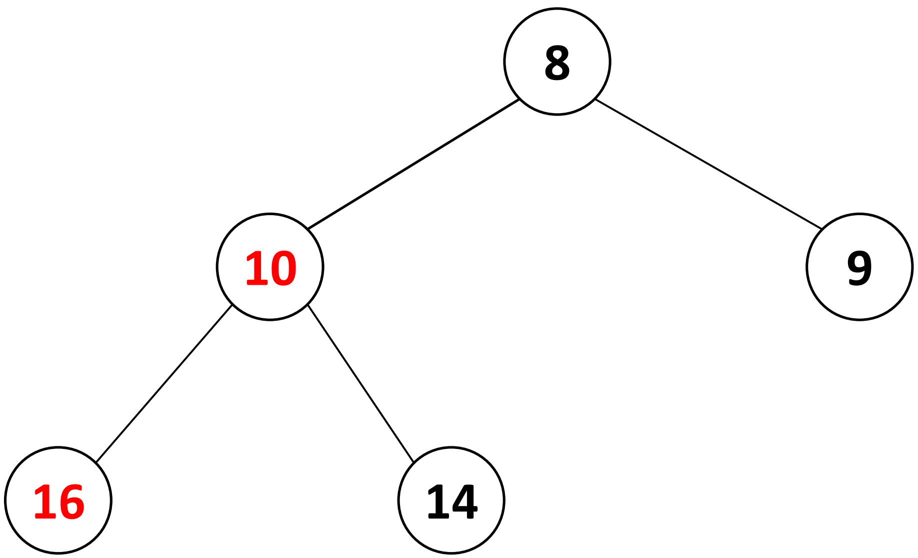
1	2	3	4	7					



Heapsort - Example

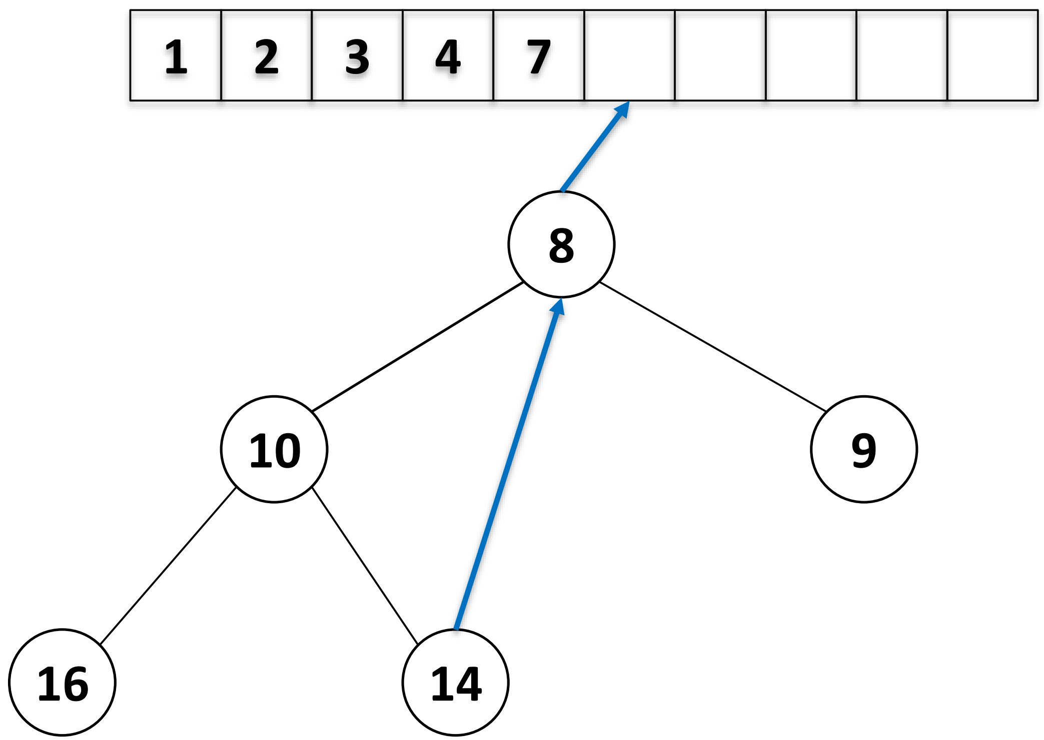
- Perform n Extract-Min operations

1	2	3	4	7					



Heapsort - Example

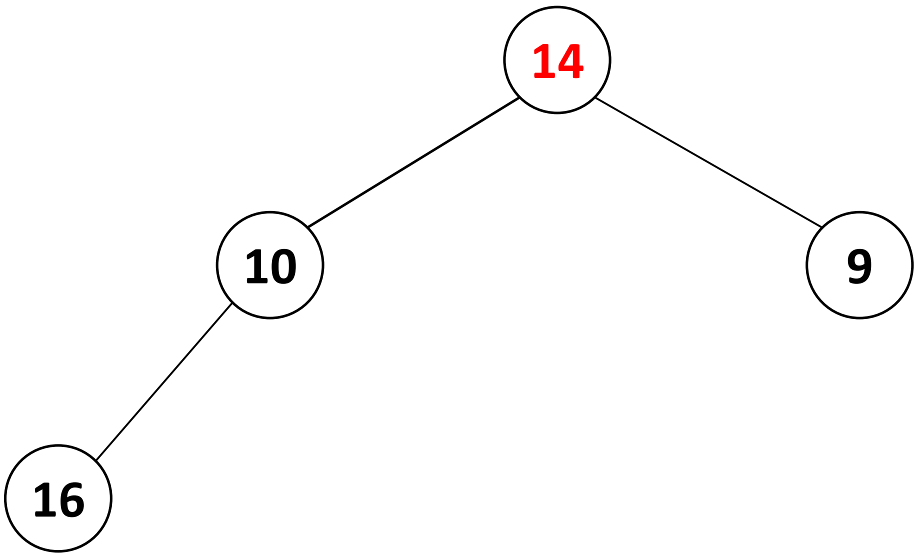
- Perform n Extract-Min operations



Heapsort - Example

- Perform n Extract-Min operations

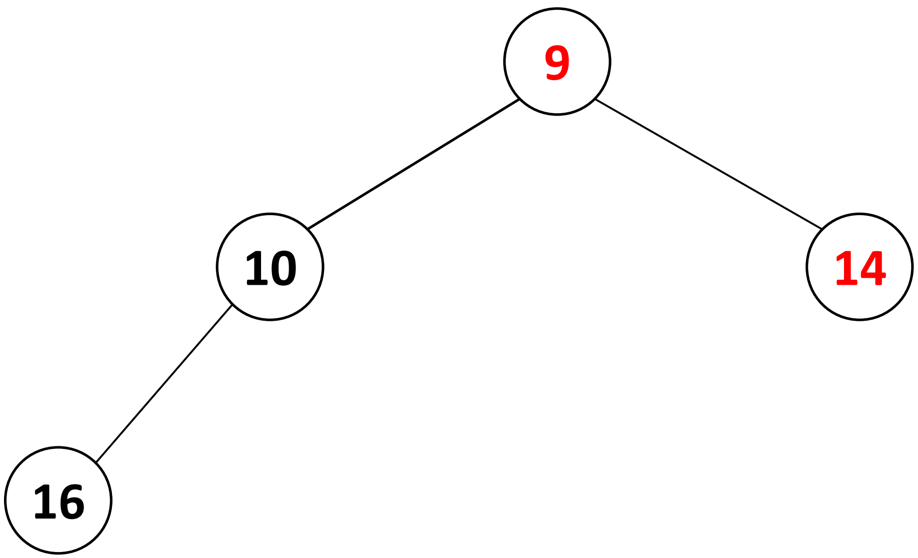
1	2	3	4	7	8				



Heapsort - Example

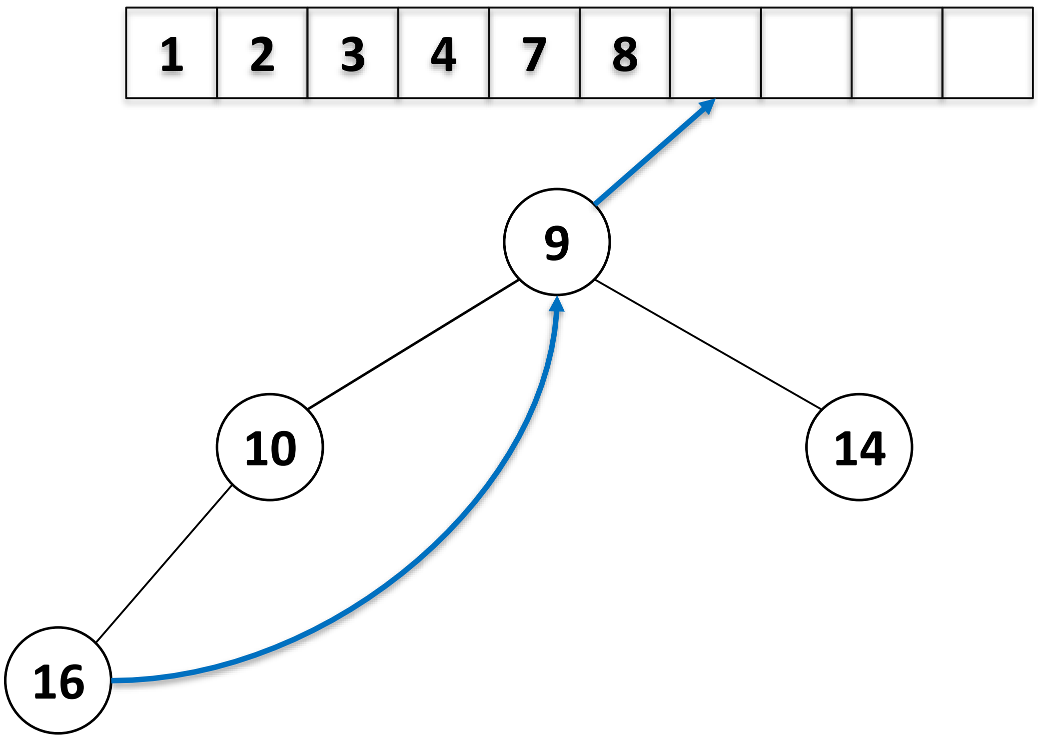
- Perform n Extract-Min operations

1	2	3	4	7	8				



Heapsort - Example

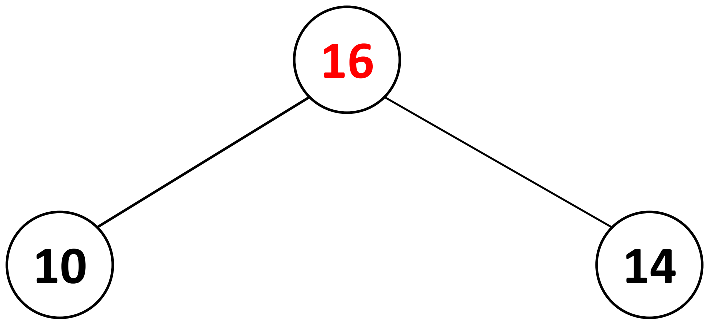
- Perform n Extract-Min operations



Heapsort - Example

- Perform n Extract-Min operations

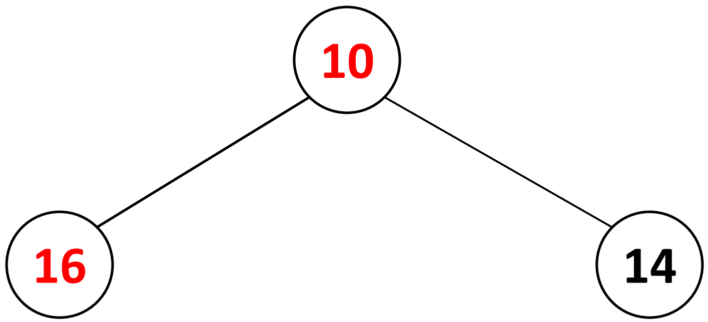
1	2	3	4	7	8	9			



Heapsort - Example

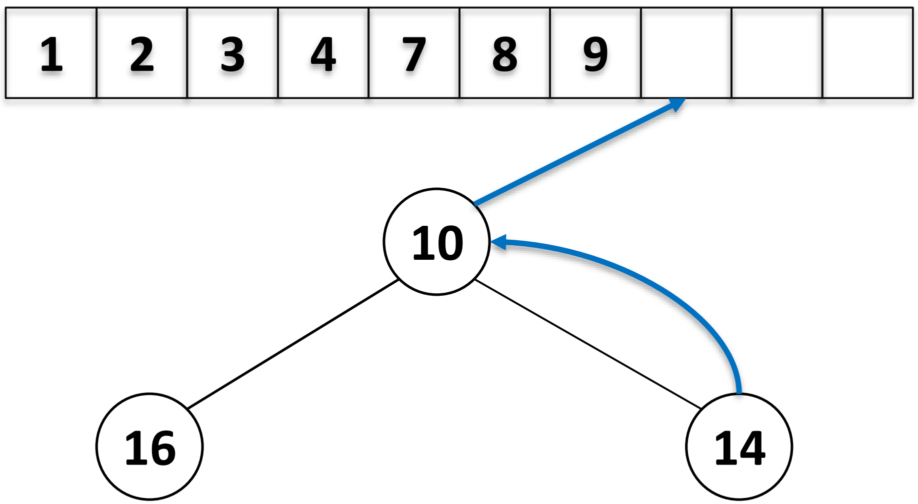
- Perform n Extract-Min operations

1	2	3	4	7	8	9			



Heapsort - Example

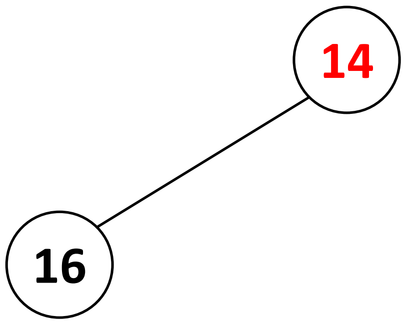
- Perform n Extract-Min operations



Heapsort - Example

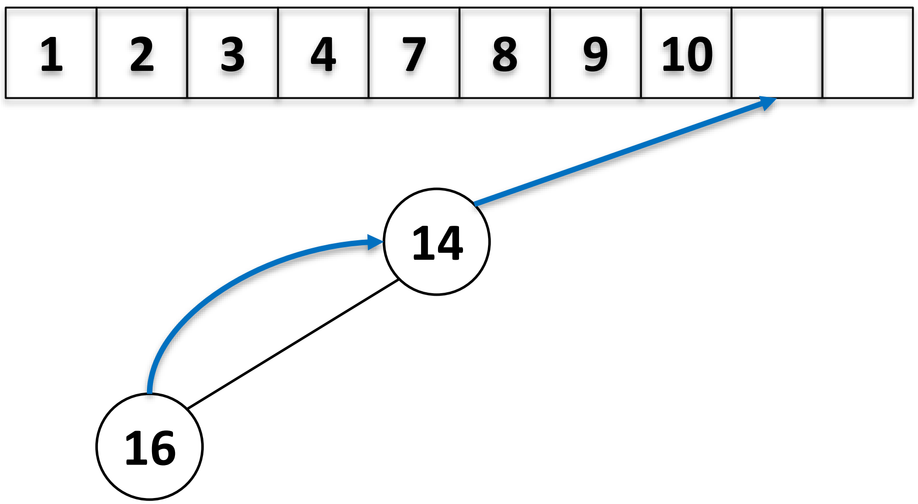
- Perform n Extract-Min operations

1	2	3	4	7	8	9	10		



Heapsort - Example

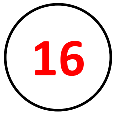
- Perform n Extract-Min operations



Heapsort - Example

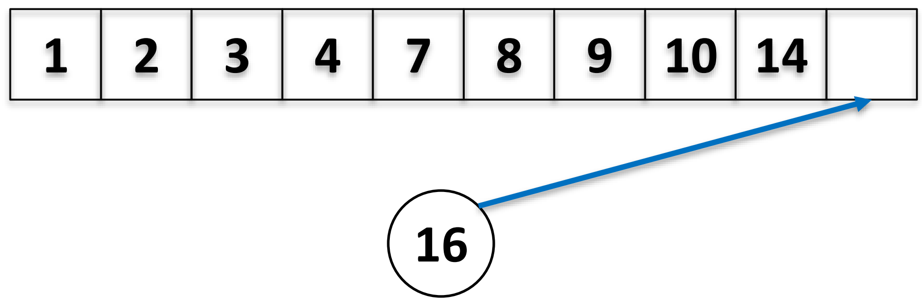
- Perform n Extract-Min operations

1	2	3	4	7	8	9	10	14	



Heapsort - Example

- Perform n Extract-Min operations



Heapsort - Example

- Perform n Extract-Min operations

1	**2**	**3**	**4**	**7**	**8**	**9**	**10**	**14**	**16**

Heapsort - Example

- Perform n Extract-Min operations

1	**2**	**3**	**4**	**7**	**8**	**9**	**10**	**14**	**16**

Summary

- Priority queue is an abstract data structure that supports two operations: Insert and Extract-Min.

Summary

- Priority queue is an abstract data structure that supports two operations: Insert and Extract-Min.

- If priority queues are implemented using heaps, then these two operations are supported in O(log n) time.

Summary

- Priority queue is an abstract data structure that supports two operations: Insert and Extract-Min.

- If priority queues are implemented using heaps, then these two operations are supported in O(log n) time.

- Heapsort takes O(n log n) time, which is as efficient as merge sort and quicksort.

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

Objective

- All sorting algorithms seen so far are based on comparing elements
  - E.g., insertion sort, merge sort, and heapsort

Objective

- All sorting algorithms seen so far are based on comparing elements
  - E.g., insertion sort, merge sort, and heapsort
- Insertion sort has worst-case running time $\Theta(n^2)$, while the others have worst-case running time $\Theta(n \log n)$

Objective

- All sorting algorithms seen so far are based on comparing elements
  - E.g., insertion sort, merge sort, and heapsort
- Insertion sort has worst-case running time $\Theta(n^2)$, while the others have worst-case running time $\Theta(n \log n)$

Question

Can we do better?

Objective

- All sorting algorithms seen so far are based on comparing elements
  - E.g., insertion sort, merge sort, and heapsort
- Insertion sort has worst-case running time $\Theta(n^2)$, while the others have worst-case running time $\Theta(n \log n)$

Question

Can we do better?

Goal

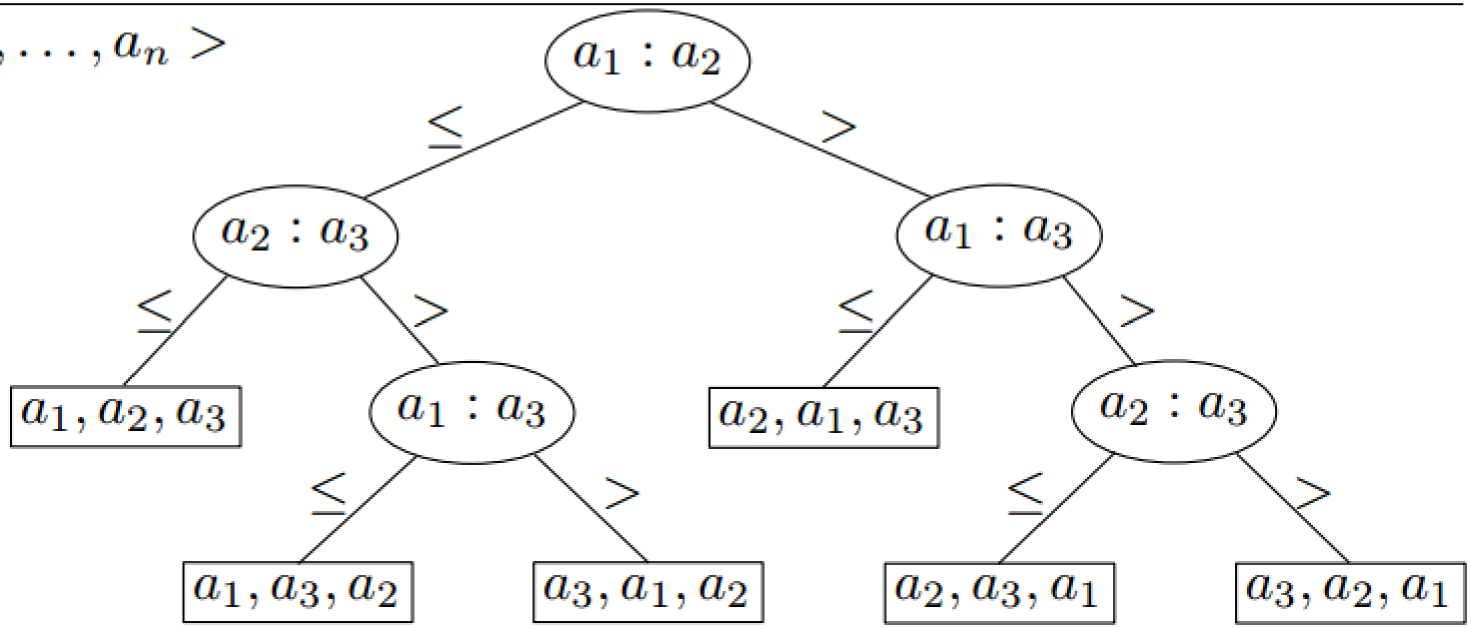
We will prove that any comparison-based sorting algorithm has a worst-case running time $\Omega(n \log n)$.

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

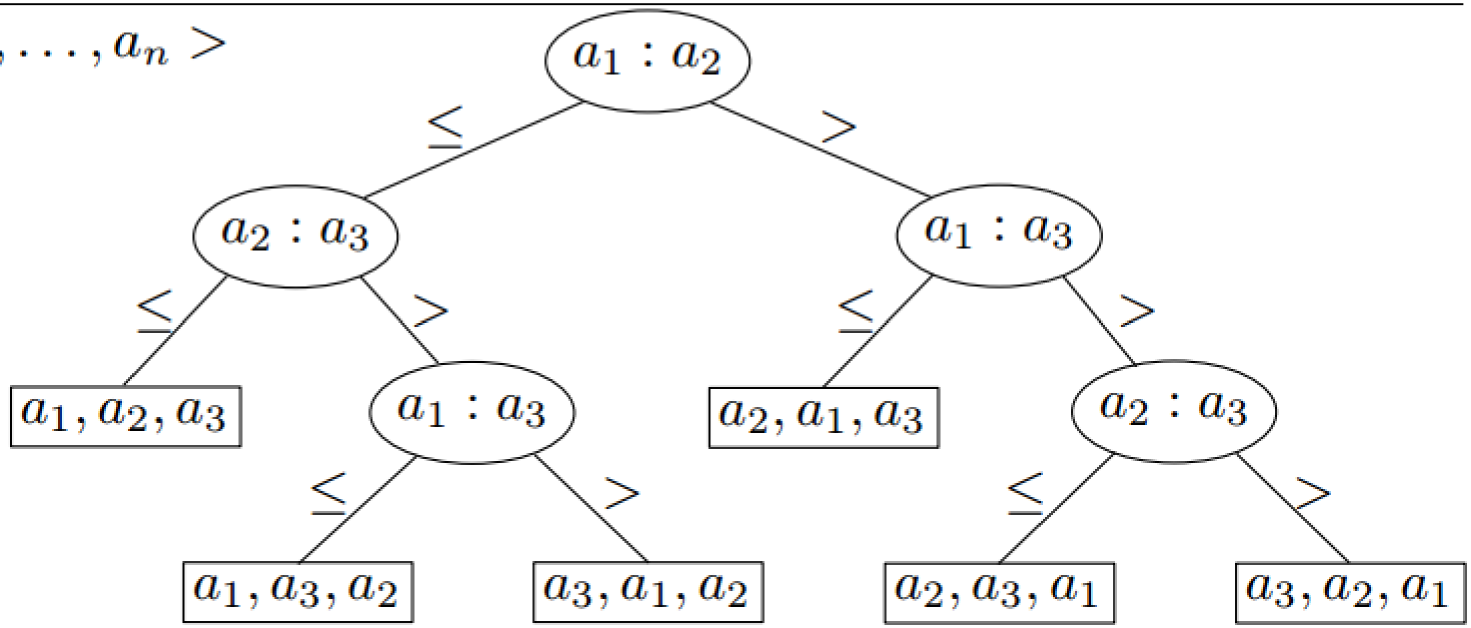
Decision-tree Model

Sort $< a_1, a_2, \ldots, a_n >$



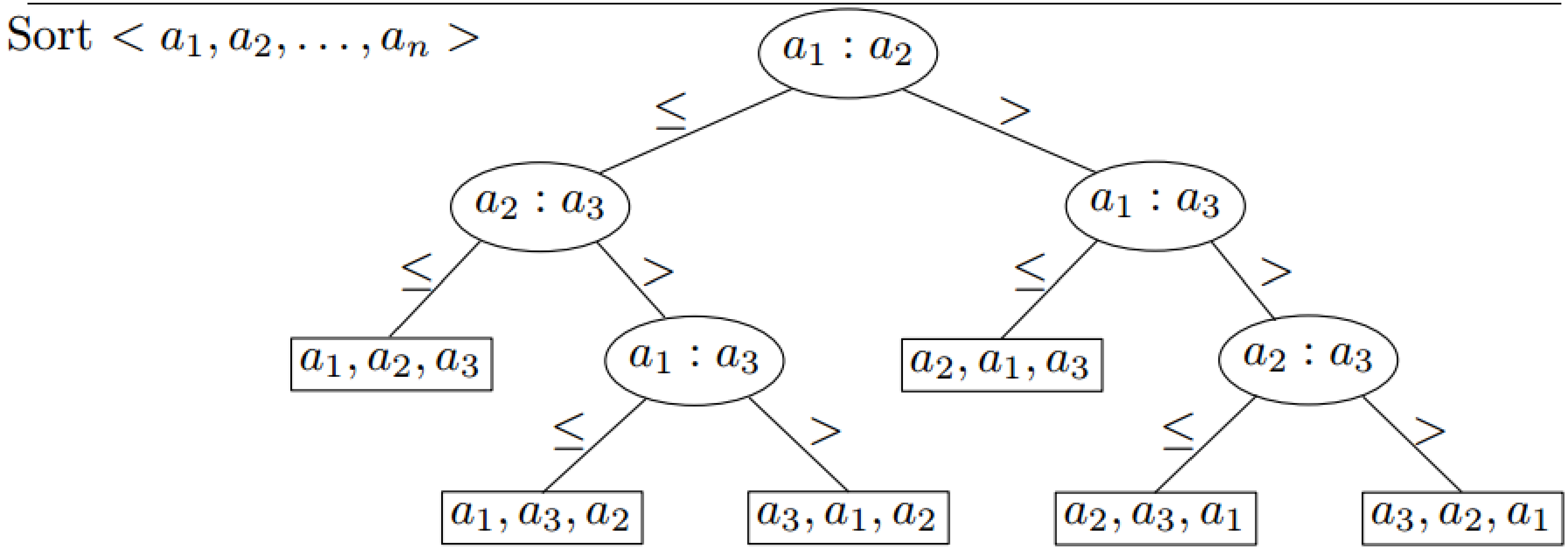
Decision-tree Model

Sort $< a_1, a_2, \ldots, a_n >$



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}

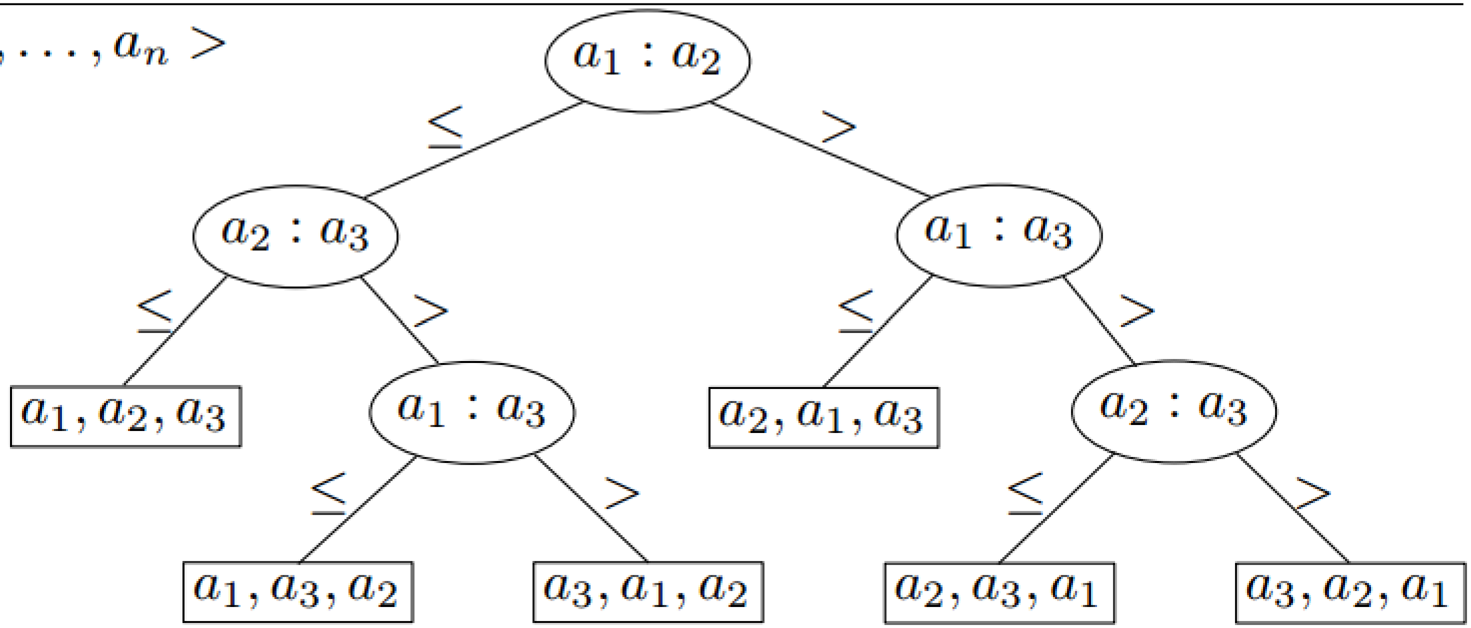
Decision-tree Model



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$

Decision-tree Model

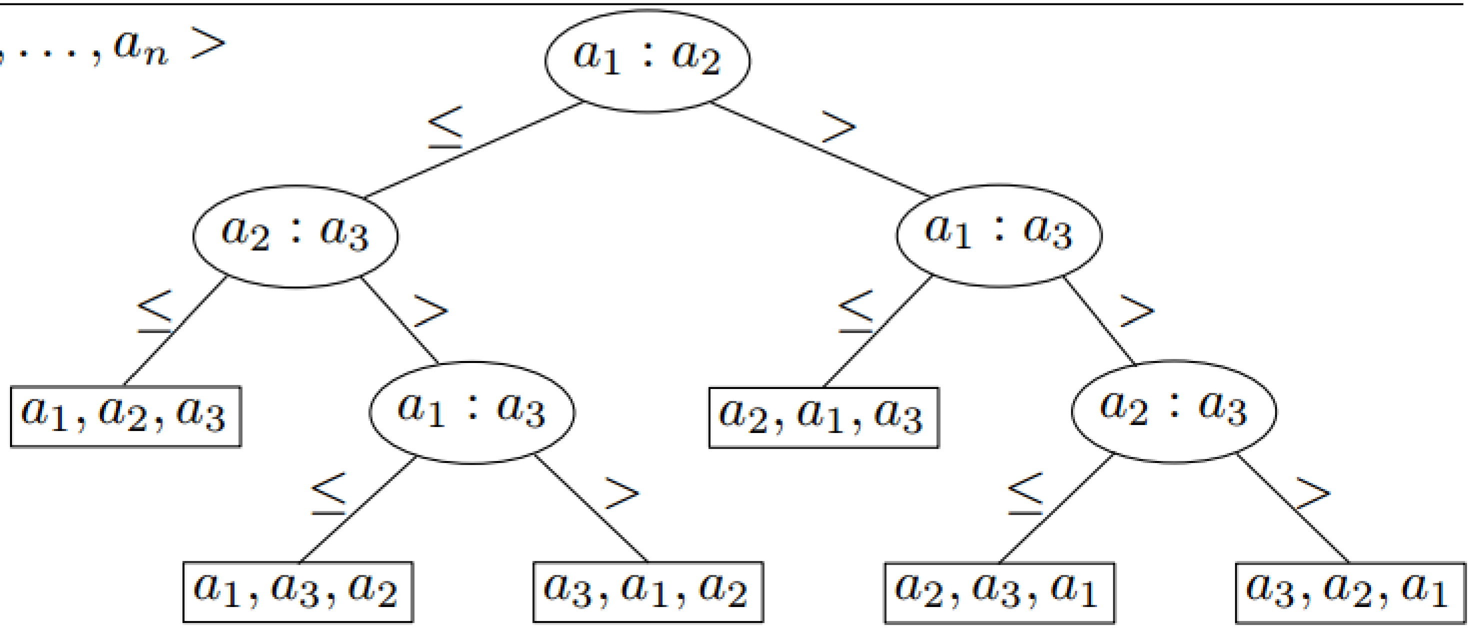
Sort $< a_1, a_2, \ldots, a_n >$



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$

Decision-tree Model

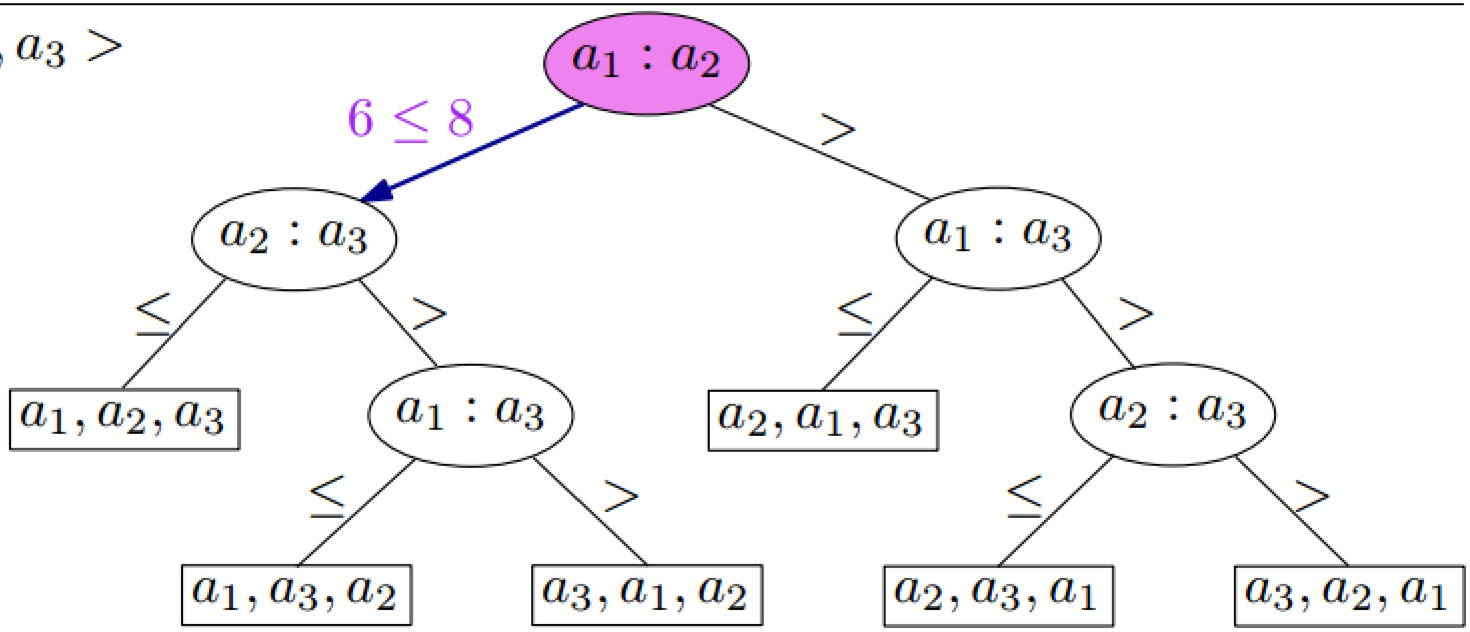
Sort $< a_1, a_2, \ldots, a_n >$



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Model

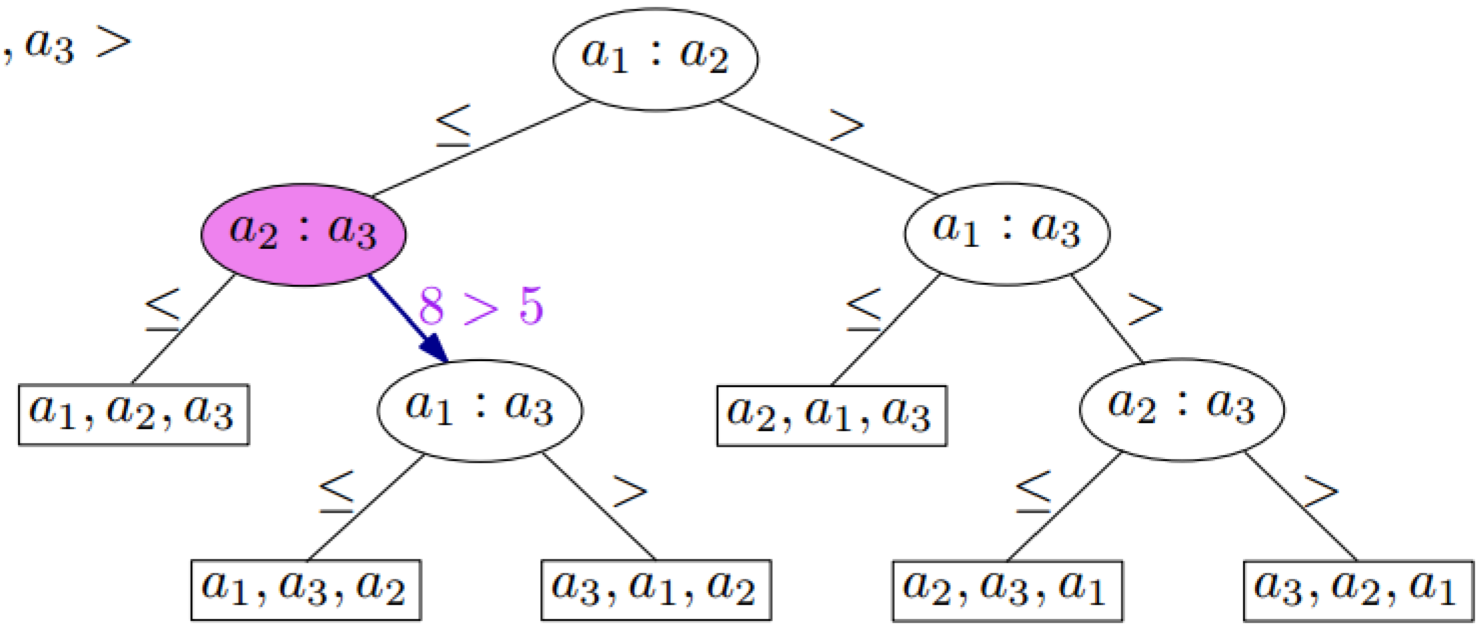
Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Model

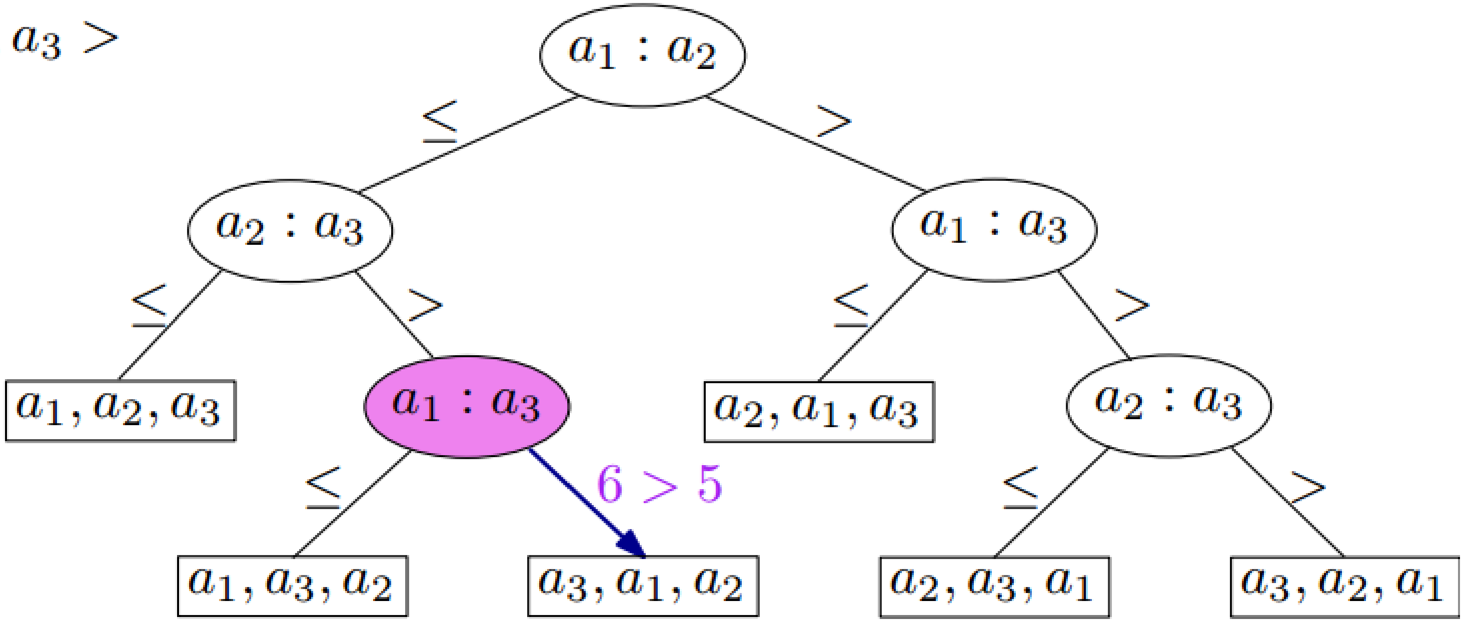
Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Model

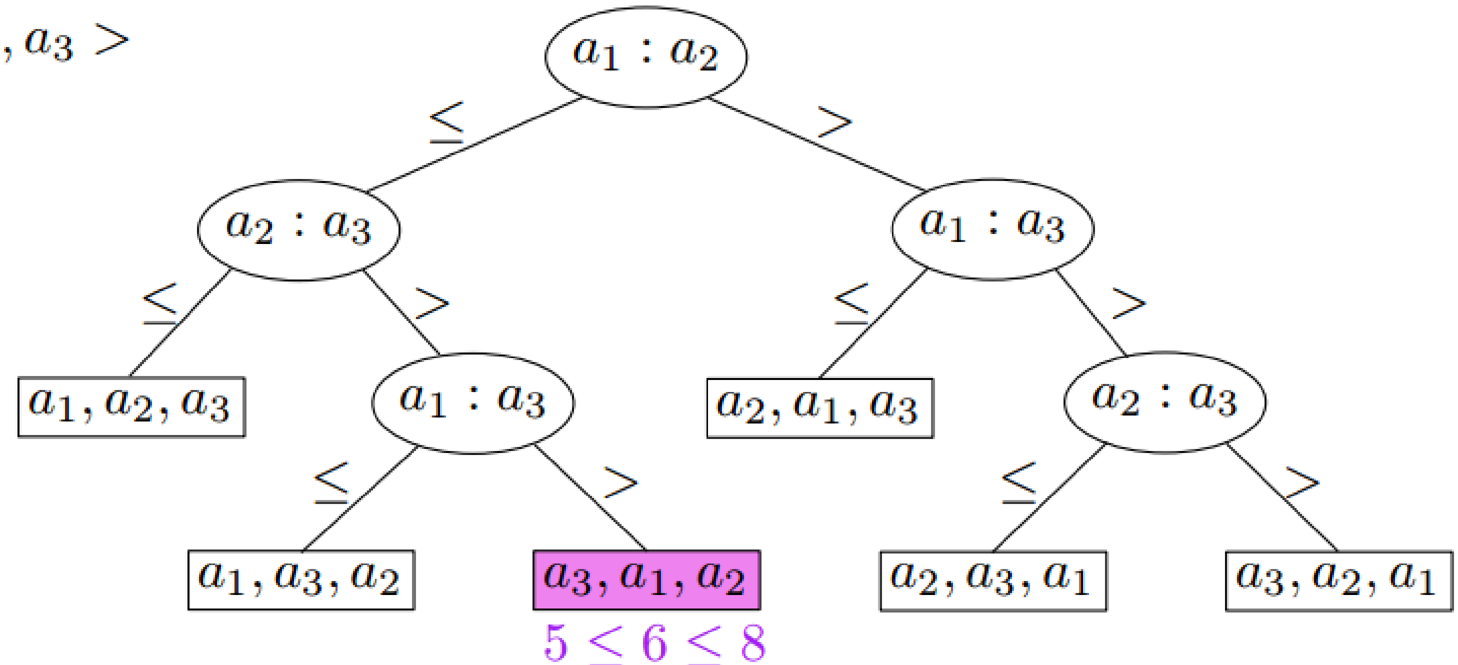
Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Model

Sort $< a_1, a_2, a_3 >$
$=< 6, 8, 5 >$:



- Each internal node is labeled $a_i : a_j$ for {1,2,…,n}
  - The left subtree shows subsequent comparisons if $a_i \leq a_j$
  - The right subtree shows subsequent comparisons if $a_i > a_j$
- Each leaf corresponds to an input ordering

Decision-tree Model

A decision tree can model the execution of any comparison-based sorting algorithm

Decision-tree Model

A decision tree can model the execution of any comparison-based sorting algorithm

- One tree for each input size n

Decision-tree Model

A decision tree can model the execution of any comparison-based sorting algorithm

- One tree for each input size n
- Worst-case running time = height of tree

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ *comparisons.*

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ *comparisons.*

Proof.

- A decision tree to sort n elements must have at least $n!$ leaves, since there are $n!$ possible orderings.

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof.

- A decision tree to sort n elements must have at least $n!$ leaves, since there are $n!$ possible orderings.
- A binary tree of height h has at most 2^h leaves

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof.

- A decision tree to sort n elements must have at least $n!$ leaves, since there are $n!$ possible orderings.
- A binary tree of height h has at most 2^h leaves
- Thus, $n! \leq 2^h$
  $\Rightarrow h \geq \log n! = \Omega(n \log n)$ (proved in previous lecture)

$\square$

Lower Bound for Sorting

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons.

Proof.

- A decision tree to sort n elements must have at least $n!$ leaves, since there are $n!$ possible orderings.
- A binary tree of height h has at most 2^h leaves
- Thus, $n! \leq 2^h$
  $\Rightarrow h \geq \log n! = \Omega(n \log n)$ (proved in previous lecture)

$\square$

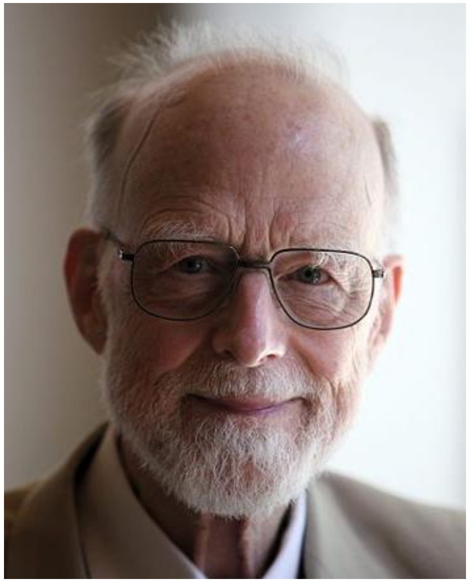
Corollary

Heapsort and merge sort are asymptotically optimal comparison-based sorting algorithms.

Summary

John von Neumann

Merge Sort Algorithm was invented in 1945



Tony Hoare

Quicksort Algorithm was invented in 1959

J. W. J. Williams

Heapsort Algorithm was invented in 1964

Can we do better for the sorting problem?

Outline

- Review to Divide-and-Conquer Paradigm
- Heapsort
  - Priority Queues
  - (Binary) Heap
  - Heapsort
- Lower Bound for Comparison-based Sorting
  - Objective
  - Decision Tree Model
- Sorting in Linear Time
  - Counting Sort

Main Ideas

- Counting sort determines, for each input element *x*, the number of elements less than *x*.

Main Ideas

- Counting sort determines, for each input element *x*, the number of elements less than *x*.

- It uses this information to place element *x* directly into its position in the output array.

Main Ideas

- Counting sort determines, for each input element *x*, the number of elements less than *x*.

- It uses this information to place element *x* directly into its position in the output array.
    - For example, if 17 elements are less than *x*, then *x* belongs in output position 18.

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0$;
end
for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
| $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
| $B[C[A[j]]] \leftarrow A[j]$;
| $C[A[j]] \leftarrow C[A[j]] - 1$;
end
return B;

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C				

B					

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0$;
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1;//C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1];//C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j]$;
  $C[A[j]] \leftarrow C[A[j]] - 1$;
end
return B;

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	0

B					

for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;$
end

Counting Sort

Counting-Sort(A,B,k)

```
Input: A[1...n] where A[j] ∈ {1, 2, ..., k}
Output: B[1...n],sorted
let C[1...k] be a new array;
for i ← 1 to k do
 |  C[i] ← 0;
end
for j ← 1 to n do
 |  C[A[j]] ← C[A[j]] + 1;//C[i] = |{key = i}|
end
for i ← 2 to k do
 |  C[i] ← C[i] + C[i − 1];//C[i] = |{key ≤ i}|
end
for j ← n to 1 do
 |  B[C[A[j]]] ← A[j];
 |  C[A[j]] ← C[A[j]] − 1;
end
return B;
```

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	0	0	1

B					

for $j \leftarrow 1$ *to* n **do**
$\quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	0	1	0	1

B					

for $j \leftarrow 1$ *to* n **do**

$\quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	1

B					

for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	1	0	2

B					

for $j \leftarrow 1$ *to* n **do**

$\quad C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$

end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end

Counting Sort

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1,2,...,k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
  $C[i] \leftarrow 0;$
end
for $j \leftarrow 1$ *to* n **do**
  $C[A[j]] \leftarrow C[A[j]] + 1; // C[i] = |\{key = i\}|$
end
for $i \leftarrow 2$ *to* k **do**
  $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end
for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;$
end
return B;

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	0	2

for $i \leftarrow 2$ *to* k **do**
$\quad C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$
end

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	3	2

$$
\begin{array}{l}
\textbf{for } i \leftarrow 2 \textit{ to } k \textbf{ do} \\
\mid \quad C[i] \leftarrow C[i] + C[i-1]; {\color{blue}//C[i] = |\{key \leq i\}|} \\
\textbf{end}
\end{array}
$$

Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2

	1	2	3	4
C	1	2	0	2

B					

C'	1	3	3	5

for $i \leftarrow 2$ *to* k **do**

| $C[i] \leftarrow C[i] + C[i-1]; // C[i] = |\{key \leq i\}|$

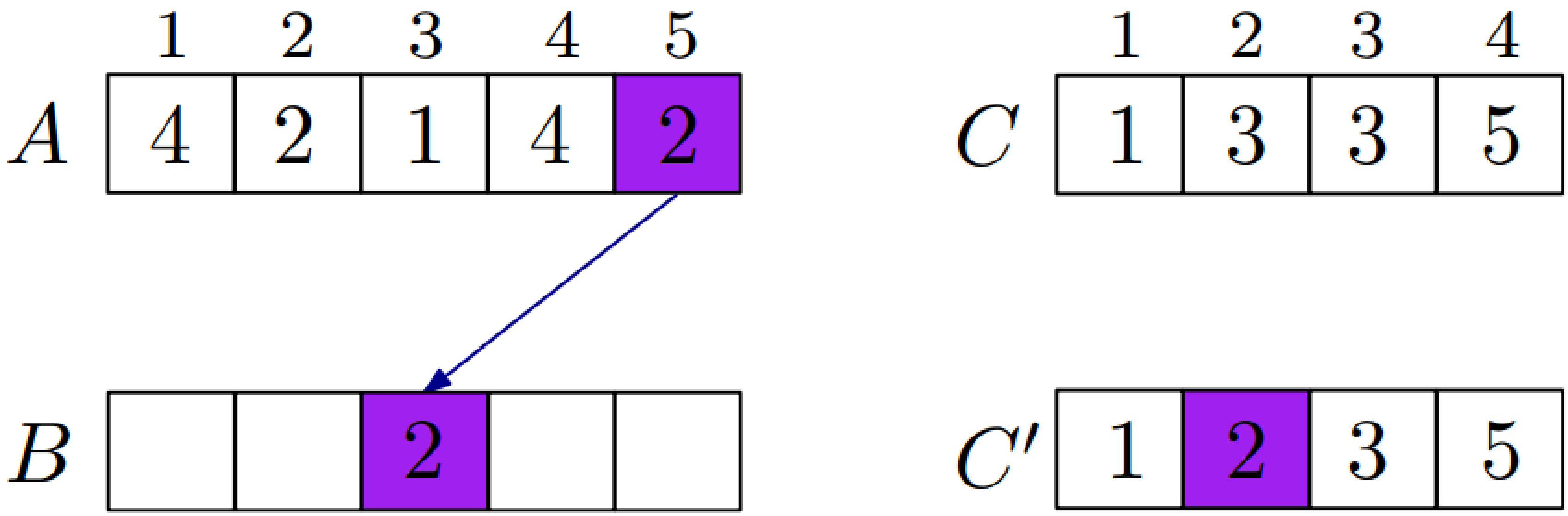
end

Counting Sort

Counting-Sort(A,B,k)

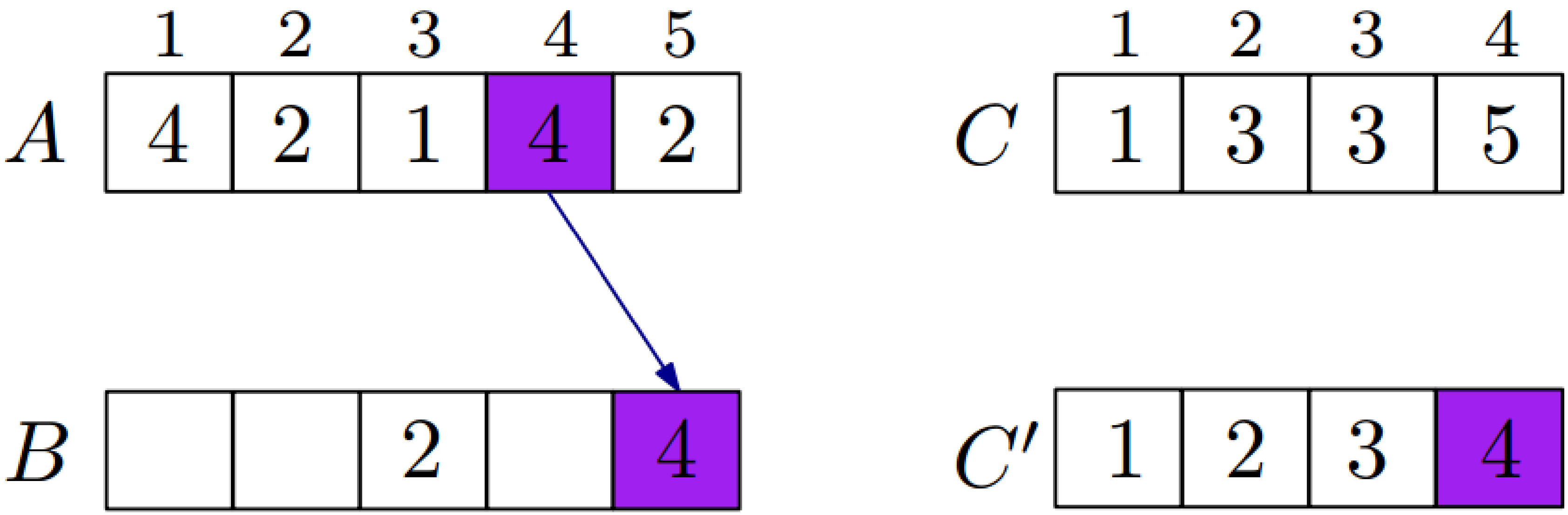
```
Input: A[1...n] where A[j] ∈ {1, 2, ..., k}
Output: B[1...n],sorted
let C[1...k] be a new array;
for i ← 1 to k do
 |  C[i] ← 0;
end
for j ← 1 to n do
 |  C[A[j]] ← C[A[j]] + 1;//C[i] = |{key = i}|
end
for i ← 2 to k do
 |  C[i] ← C[i] + C[i − 1];//C[i] = |{key ≤ i}|
end
for j ← n to 1 do
 |  B[C[A[j]]] ← A[j];
 |  C[A[j]] ← C[A[j]] − 1;
end
return B;
```

Example: Counting Sort



$$
\begin{array}{l}
\textbf{for } j \leftarrow n \textit{ to } 1 \textbf{ do} \\
\quad B[C[A[j]]] \leftarrow A[j]; \\
\quad C[A[j]] \leftarrow C[A[j]] - 1; \\
\textbf{end}
\end{array}
$$

Example: Counting Sort



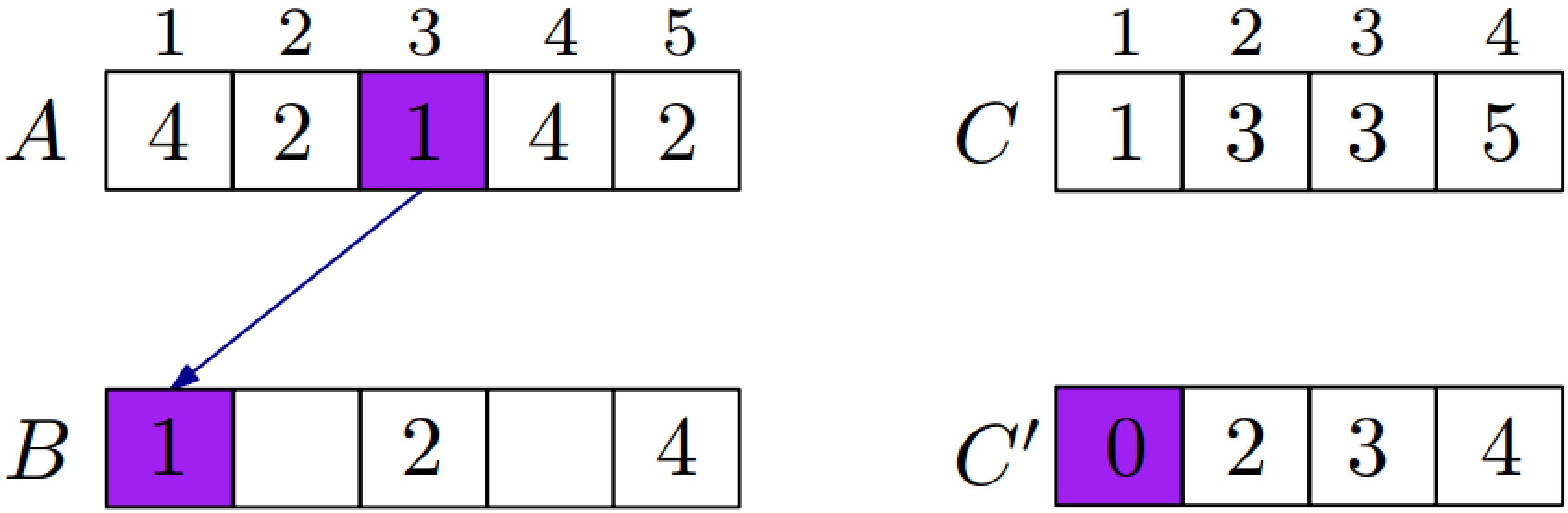
for $j \leftarrow n$ *to* 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

Example: Counting Sort



for $j \leftarrow n$ *to* 1 **do**

$\quad B[C[A[j]]] \leftarrow A[j];$

$\quad C[A[j]] \leftarrow C[A[j]] - 1;$

end

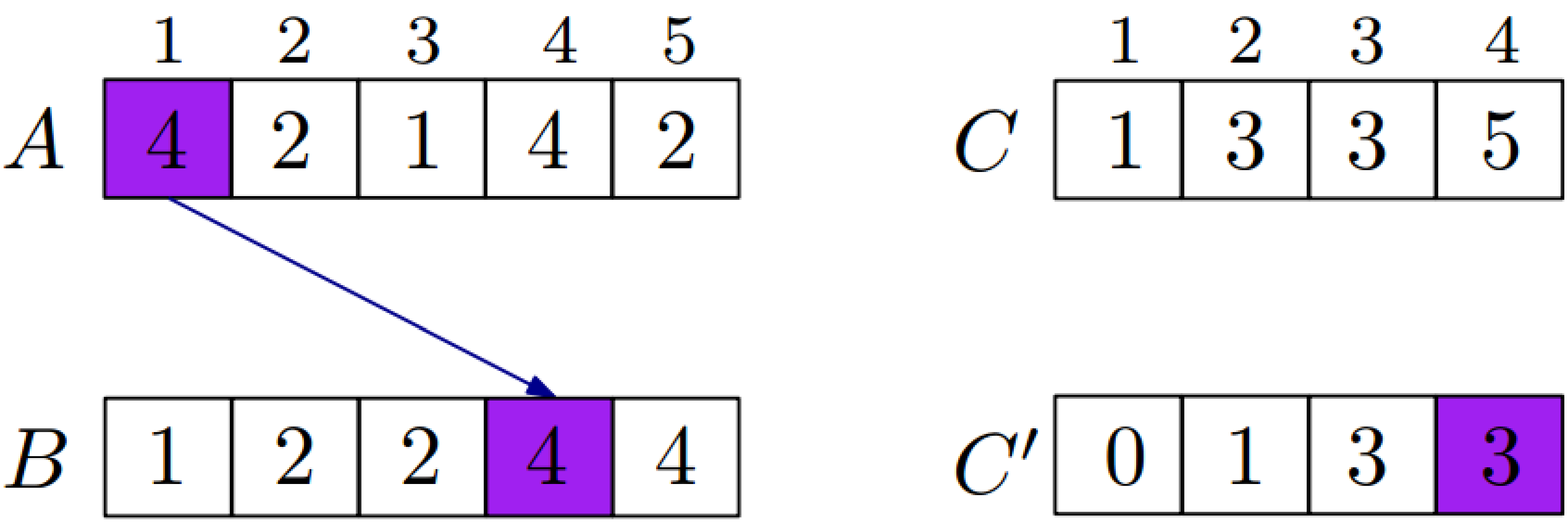
Example: Counting Sort

	1	2	3	4	5
A	4	2	1	4	2
B	1	2	2		4

	1	2	3	4
C	1	3	3	5
C'	0	1	3	4

for $j \leftarrow n$ *to* 1 **do**
  $B[C[A[j]]] \leftarrow A[j];$
  $C[A[j]] \leftarrow C[A[j]] - 1;$
end

Example: Counting Sort



$$
\begin{array}{l}
\textbf{for } j \leftarrow n \ \textit{to}\ 1 \textbf{ do} \\
\quad B[C[A[j]]] \leftarrow A[j]; \\
\quad C[A[j]] \leftarrow C[A[j]] - 1; \\
\textbf{end}
\end{array}
$$

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0;//O(k)$
end

return B;

Analysis

Counting-Sort(A,B,k)

Input: $\boldsymbol{A}[1...n]$ where $A[j] \in \{1, 2, ..., k\}$
Output: $\boldsymbol{B}[1...n]$,sorted
let $C[1...k]$ be a new array;
for $i \leftarrow 1$ *to* k **do**
| $C[i] \leftarrow 0;//O(k)$
end
for $j \leftarrow 1$ *to* n **do**
| $C[A[j]] \leftarrow C[A[j]] + 1;//O(n)$
end

return B;

Analysis

Counting-Sort(A,B,k)

```
Input: A[1...n] where A[j] ∈ {1, 2, ..., k}
Output: B[1...n],sorted
let C[1...k] be a new array;
for i ← 1 to k do
 |  C[i] ← 0;//O(k)
end
for j ← 1 to n do
 |  C[A[j]] ← C[A[j]] + 1;//O(n)
end
for i ← 2 to k do
 |  C[i] ← C[i] + C[i − 1];//O(k)
end



return B;
```

Analysis

Counting-Sort(A,B,k)

```
Input: A[1...n] where A[j] ∈ {1, 2, ..., k}
Output: B[1...n],sorted
let C[1...k] be a new array;
for i ← 1 to k do
 |  C[i] ← 0;//O(k)
end
for j ← 1 to n do
 |  C[A[j]] ← C[A[j]] + 1;//O(n)
end
for i ← 2 to k do
 |  C[i] ← C[i] + C[i − 1];//O(k)
end
for j ← n to 1 do
 |  B[C[A[j]]] ← A[j];
 |  C[A[j]] ← C[A[j]] − 1;//O(n)
end
return B;
```

Analysis

Counting-Sort(A,B,k)

```
Input: A[1...n] where A[j] ∈ {1, 2, ..., k}
Output: B[1...n],sorted
let C[1...k] be a new array;
for i ← 1 to k do
 |  C[i] ← 0;//O(k)
end
for j ← 1 to n do
 |  C[A[j]] ← C[A[j]] + 1;//O(n)
end
for i ← 2 to k do
 |  C[i] ← C[i] + C[i − 1];//O(k)
end
for j ← n to 1 do
 |  B[C[A[j]]] ← A[j];
 |  C[A[j]] ← C[A[j]] − 1;//O(n)
end
return B;
```

Total: $O(n+k)$

Running Time

If $k = O(n)$, then counting sort takes $O(n)$ time.

- But didn't we prove that sorting must take $\Omega(n \log n)$ time?

Running Time

If k = O(n), then counting sort takes O(n) time.

- But didn't we prove that sorting must take Ω(n log n) time?
- No, actually we proved that any comparison-based sorting algorithm takes Ω(n log n) time.

Running Time

If k = O(n), then counting sort takes O(n) time.

- But didn't we prove that sorting must take Ω(n log n) time?
- No, actually we proved that any comparison-based sorting algorithm takes Ω(n log n) time.
- Note that counting sort is not a comparison-based sorting algorithm.

Running Time

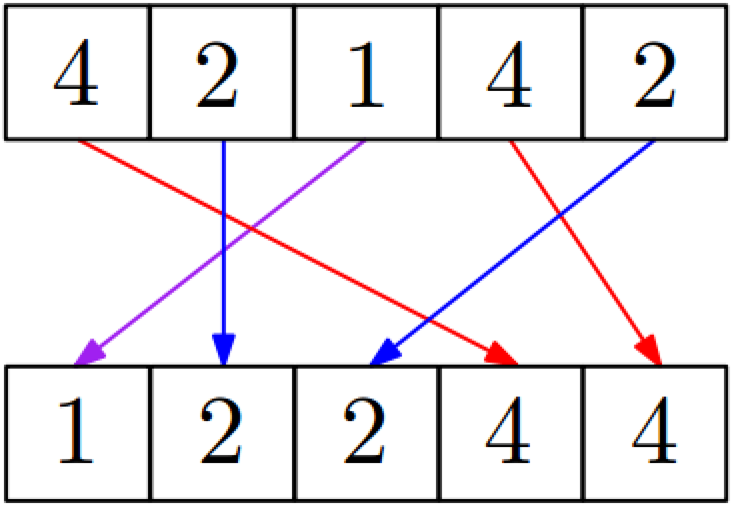
If k = O(n), then counting sort takes O(n) time.

- But didn't we prove that sorting must take Ω(n log n) time?
- No, actually we proved that any comparison-based sorting algorithm takes Ω(n log n) time.
- Note that counting sort is not a comparison-based sorting algorithm.
- In fact, it makes no comparison at all!

Stable Sorting

Counting sort is a stable sort

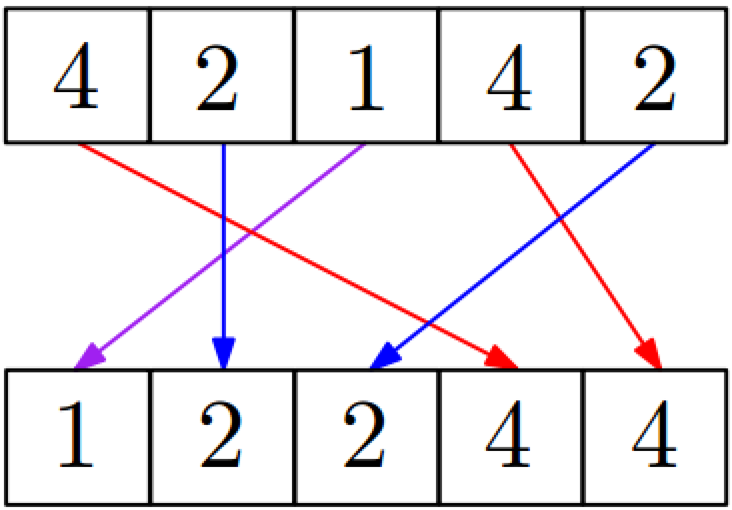
- it preserves the input order among equal elements.



Stable Sorting

Counting sort is a stable sort

- it preserves the input order among equal elements.



Exercise

What other sorts have this property?

谢谢

BDA
Beihang University