Lecture 28 (Sorting 1)

Basic Sorts

CS61B, Spring 2025 @ UC Berkeley

Slides credit: Josh Hug

Goal: Sorting

Lecture 28, CS61B, Spring 2025

61B Phase 3

We are now in Phase 3 of the course:

- Algorithms and Software Engineering.

Lectures in this phase:

- Algorithms.
- 4 software engineering lectures.

Optional textbook for software engineering lectures: “A Philosophy of Software Design” by John Ousterhout.

61B Phase 3

We are now in Phase 3 of the course:

- Algorithms and Software Engineering.

Only one assignment in this phase: Project 3: Build Your Own World

- (partners required except by exception).
- Second chance to do some software engineering (after project 2B).
- Lots more design practice.
- You'll decide your own task and approach.
  - Includes "class design" (picking classes) AND data structure selection.
  - Just like project 2B, your choices will make a huge difference in code efficiency as well as ease of writing code.

Our Major Focus for Several Lectures: Sorting

For many of our remaining lectures, we'll discuss the sorting problem.

- Informally: Given items, put them in order.

This is a useful task in its own right. Examples:

- Equivalent items are adjacent, allowing rapid duplicate finding.
- Items are in increasing order, allowing binary search.
- Can be converted into various balanced data structures (e.g. BSTs, KdTrees).

Also provide interesting case studies for how to approach basic computational problems.

- Some of the solutions will involve using data structures we've studied.

The Sorting Problem

Lecture 28, CS61B, Spring 2025

Sorting - Definitions (from Donald Knuth's TAOCP)

An **ordering relation** < for keys a, b, and c has the following properties:

- Law of Trichotomy: Exactly one of a < b, a = b, b < a is true.
- Law of Transitivity: If a < b, and b < c, then a < c.

An ordering relation with the properties above is also known as a "total order".

A **sort** is a permutation (re-arrangement) of a sequence of elements that puts the keys into non-decreasing order relative to a given ordering relation.

- $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_N$

Example: String Length

Example of an ordering relation: The length of strings.

- Law of Trichotomy: Exactly one of the following is true:
  - len(a) < len(b)
  - len(a) = len(b)
  - len(b) < len(a)
- Law of Transitivity: If len(a) < len(b) and len(b) < len(c), then len(a) < len(c).

Two valid sorts for [“cows”, “get”, “going”, “the”] for the ordering relation above:

- [“the”, “get”, “cows”, “going”]
- [“get”, “the”, “cows”, “going”]

= under the relation, not the Java idea of `.equals`

Under this relation, “the” is considered = to “get”, since len(“the”) = len(“get”).

Java Note

Ordering relations are typically given in the form of `compareTo` or `compare` methods.

```java
import java.util.Comparator;

public class LengthComparator implements Comparator<String> {
    public int compare(String x, String b) {
        return x.length() - b.length();
    }
}
```

Note that with respect to the order defined by the method above “the” = “get”.

- This usage of = is not the same as the `equals` given by the String method.

Sorting: An Alternate Viewpoint

An ***inversion*** is a pair of elements that are out of order with respect to <.

Yoda

```
0 1 1 2 3 4 8 6 9 5 7
```

```
8-6 8-5 8-7 6-5 9-5 9-7
```

(6 inversions out of 55 max)

Gabriel Cramer

Another way to state the goal of sorting:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Performance Definitions

Characterizations of the runtime efficiency are sometimes called the **time complexity** of an algorithm. Example:

- Dijkstra's has time complexity O(E log V).

Each *primitive* operation (addition, array access, etc.) counts as 1 unit of time, it its time cost is independent of input size.

Characterizations of the "extra" memory usage of an algorithm is sometimes called the **space complexity** of an algorithm.

- Dijkstra's has space complexity Θ(V) (for queue, distTo, edgeTo).
  - Note that the graph takes up space Θ(V+E), but we don't count this as part of the space complexity of Dijkstra since the graph itself already exists and is an input to Dijkstra's.

Each *primitive* object (one variable, one element of a list, etc.) counts as 1 unit of space, if its size is independent of input size.

Selection Sort

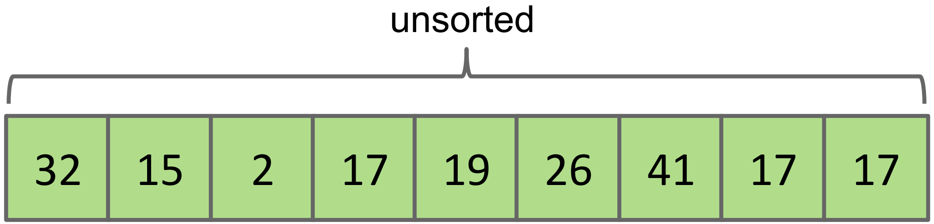
Lecture 28, CS61B, Spring 2025

Selection Sort

Selection sorting N items:

- Find the smallest item in the unsorted portion of the array.
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.

Input:

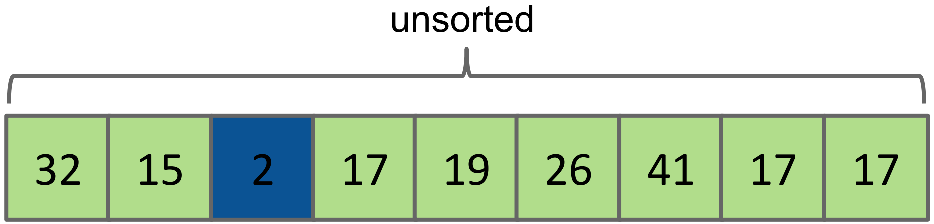


Selection Sort

Selection sorting N items:

- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.

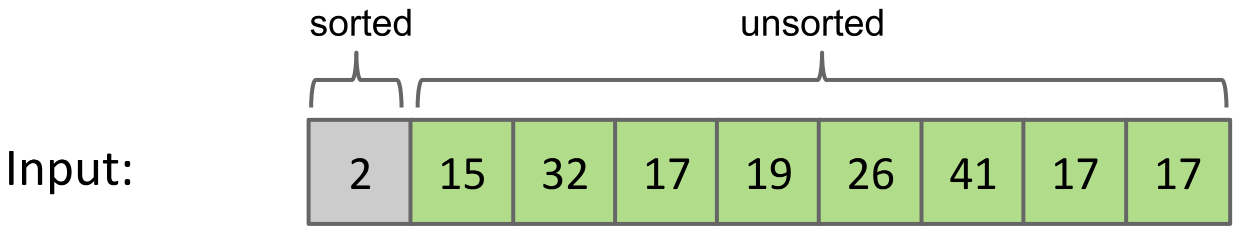
Input:



Selection Sort

Selection sorting N items:

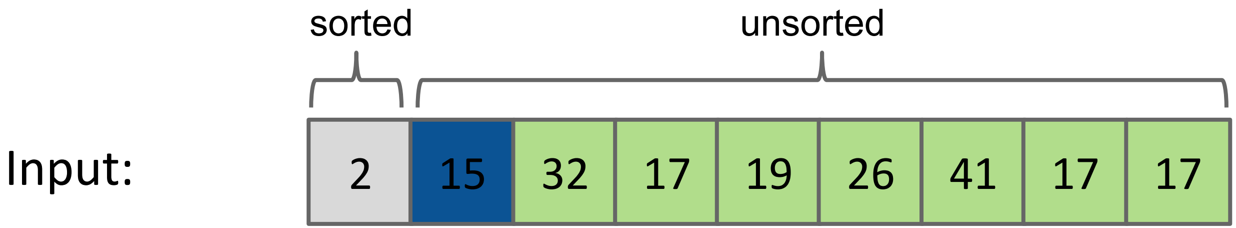
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

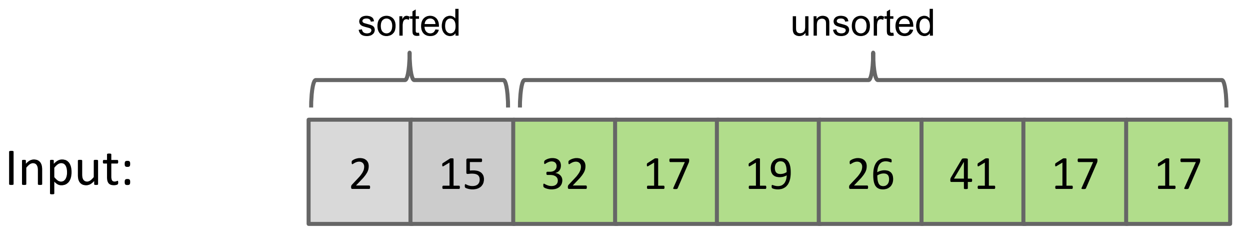
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.

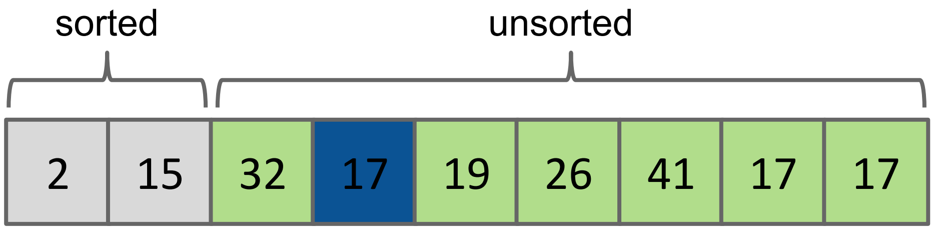


Selection Sort

Selection sorting N items:

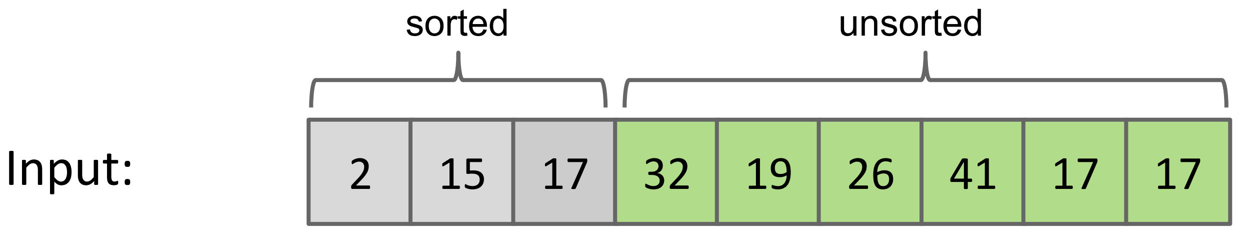
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.

Input:



Selection sorting N items:

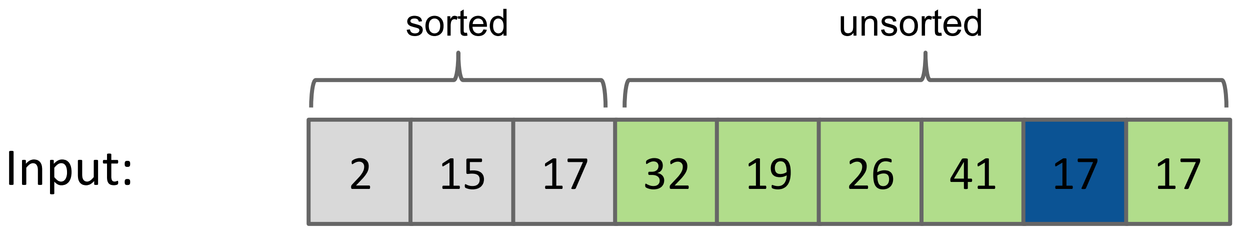
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

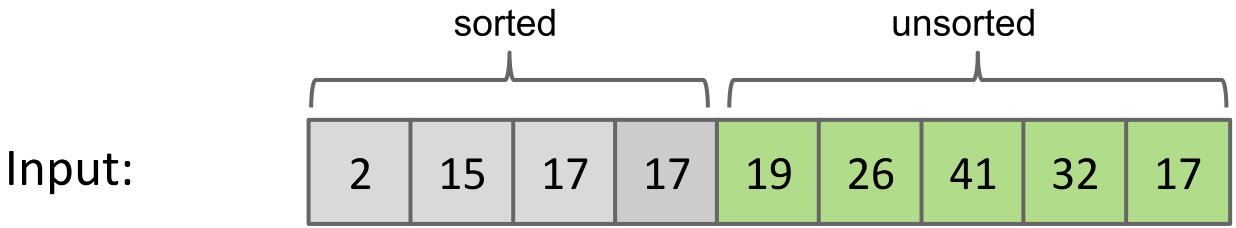
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

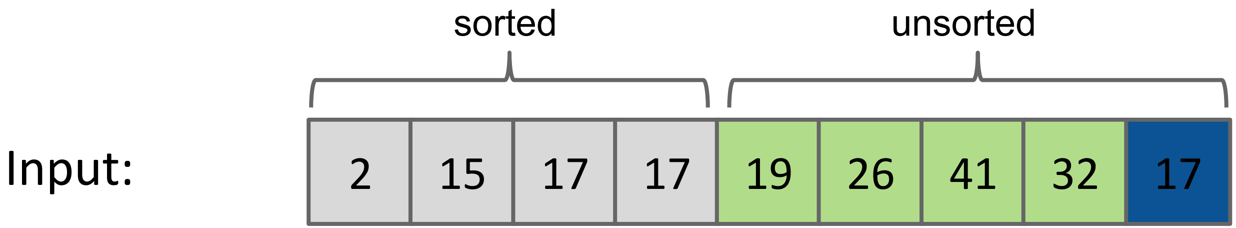
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.

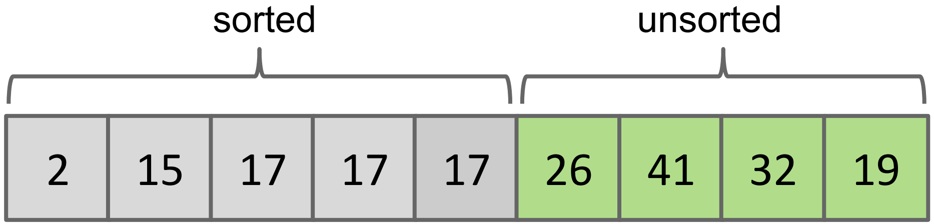


Selection Sort

Selection sorting N items:

- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.

Input:

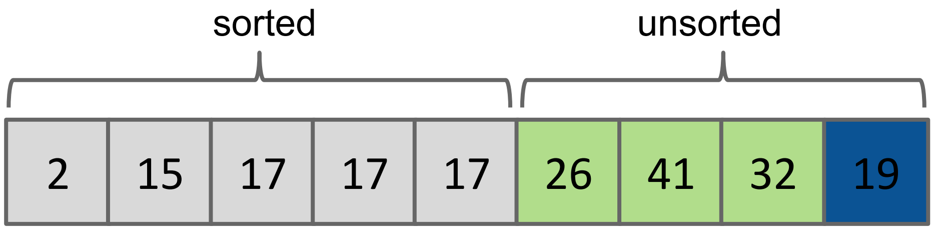


Selection Sort

Selection sorting N items:

- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.

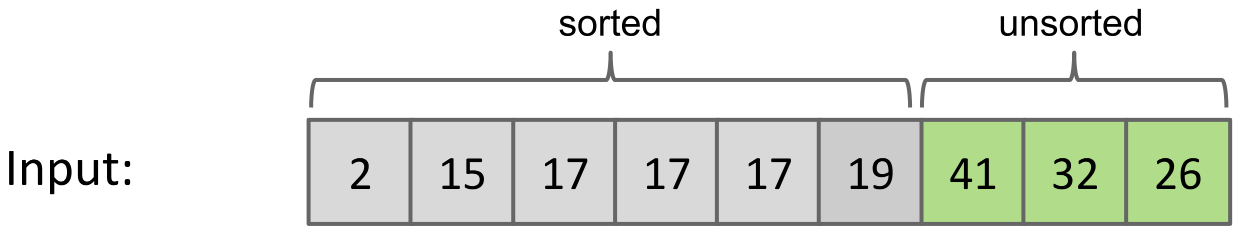
Input:



Selection Sort

Selection sorting N items:

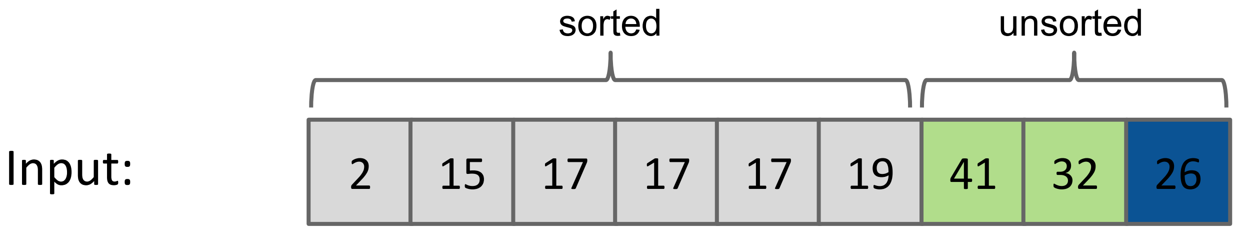
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

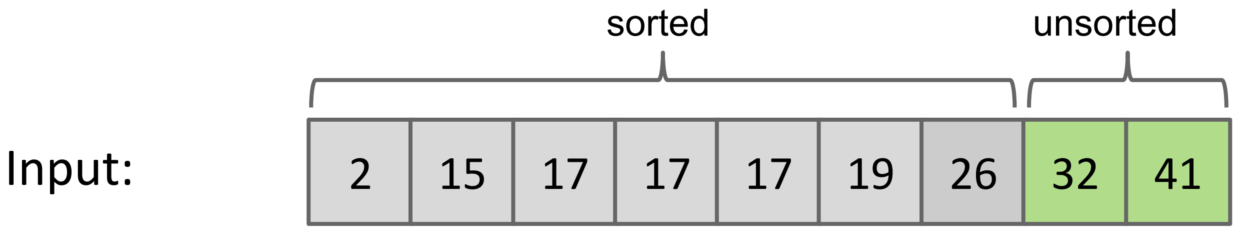
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

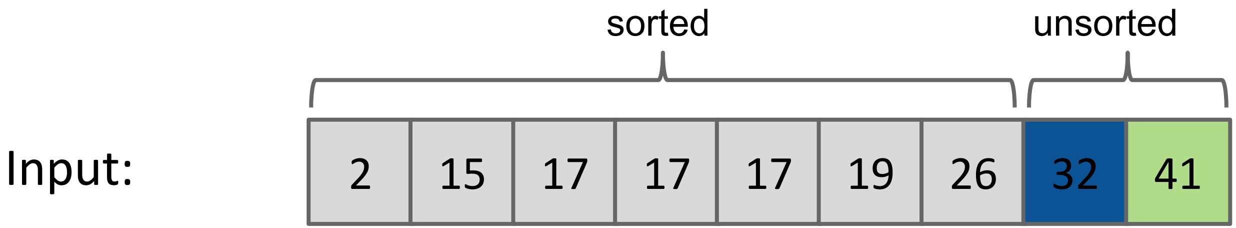
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

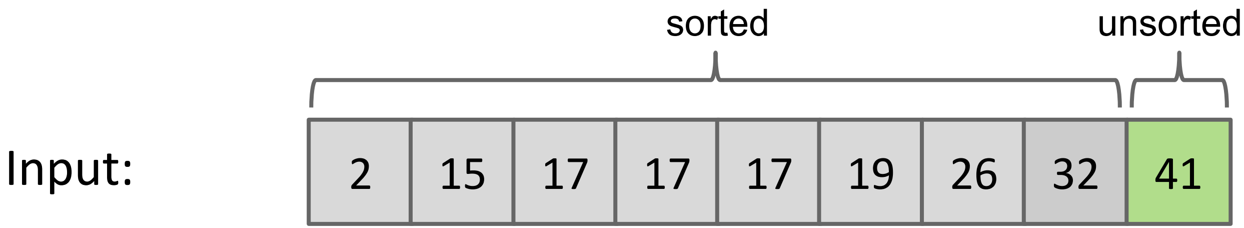
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

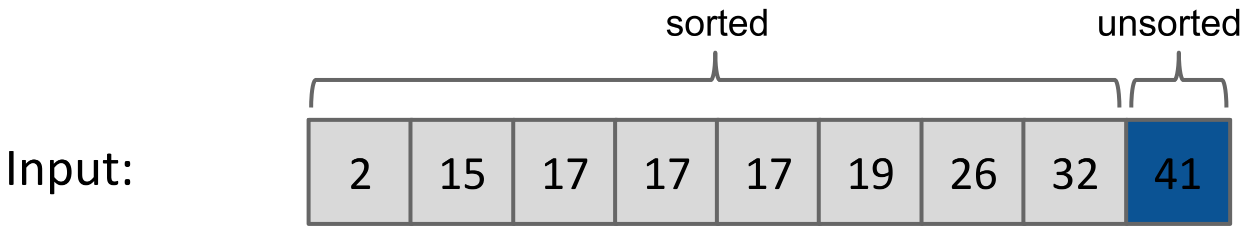
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.

sorted

Input:

2	15	17	17	17	19	26	32	41

Selection Sort

We've seen this already.

- Find smallest item.
- Swap this item to the front and 'fix' it.
- Repeat for unfixed items until all items are fixed.

Sort Properties:

- $\Theta(N^2)$ time if we use an array (or similar data structure).
- $\Theta(1)$ memory if we swap *in-place*.

Seems inefficient: We look through entire remaining array every time to find the minimum.

Naive Heapsort

Lecture 28, CS61B, Spring 2025

Naive Heapsort: Leveraging a Max-Oriented Heap

Idea: Instead of rescanning entire array looking for minimum, maintain a heap so that getting the minimum is fast!

For reasons that will become clear soon, we'll use a max-oriented heap.

A min heap would work as well, but wouldn't be able to take advantage of the fancy trick in a few slides.

Naive heapsorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put largest item at the end of the unused part of the output array.

Naive Heap Sort

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put largest item at the end of the unused part of the output array.

Input:

32	15	2	17	19	26	41	17	17

Naive Heap Sort: Phase 1: Heap Creation

Heap sorting N items:

- **Insert all items into a max heap**, and discard input array. Create output array.

Input:

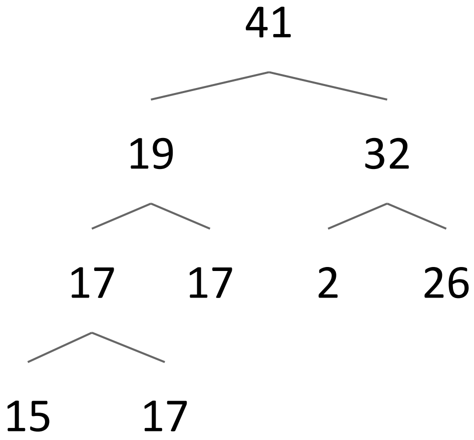
32	15	2	17	19	26	41	17	17

(Recall our heap implementation left position 0 unused)

Heap:

0	41	19	32	17	17	2	26	15	17

Size: 9



Naive Heap Sort: Phase 1: Heap Creation

Heap sorting N items:

- **Insert all items into a max heap**, and discard input array. Create output array.
- **Test your understanding: What is the runtime to complete this step?**

Input:

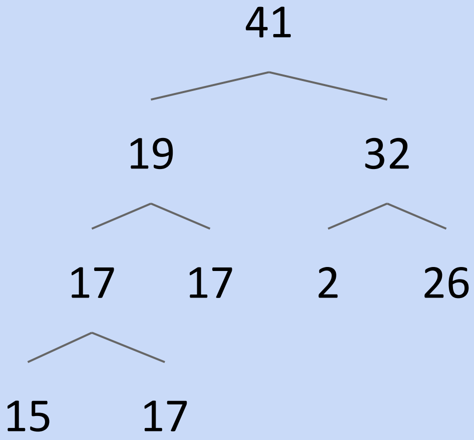
32	15	2	17	19	26	41	17	17

(Recall our heap implementation left position 0 unused)

Heap:

0	41	19	32	17	17	2	26	15	17

Size: 9



Naive Heap Sort: Phase 2: Heap Deletion

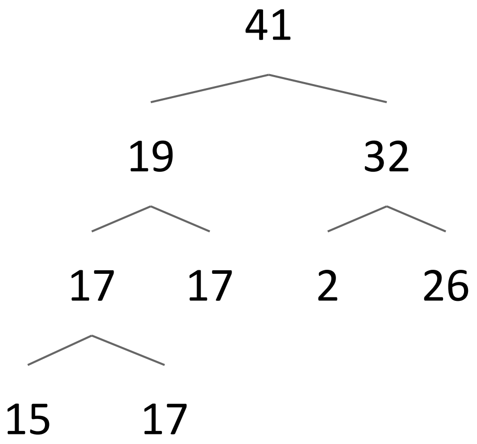
Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put largest item at the end of the unused part of the output array.

Heap:

0	41	19	32	17	17	2	26	15	17

Size: 9



Output:

0	0	0	0	0	0	0	0	0

Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**

Heap:

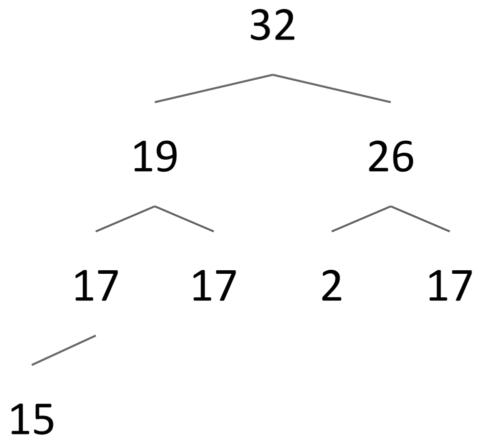
0	32	19	26	17	17	2	17	15	0

Size: 8

sorted

Output:

0	0	0	0	0	0	0	0	41



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.

Heap:

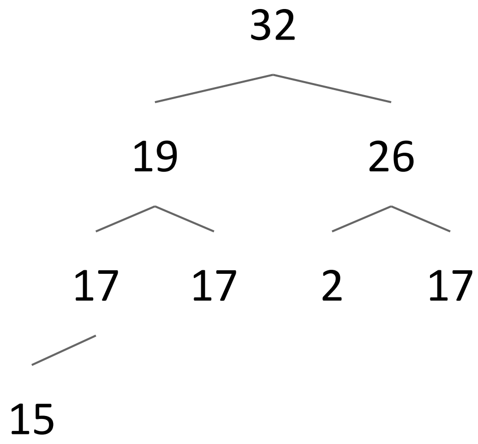
0	32	19	26	17	17	2	17	15	0

Size: 8

sorted

Output:

0	0	0	0	0	0	0	0	41



Naive Heap Sort: Phase 2: Heap Deletion

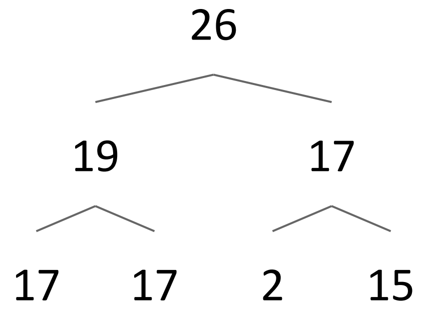
Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**

Heap:

0	26	19	17	17	17	2	15	0	0

Size: 7



sorted

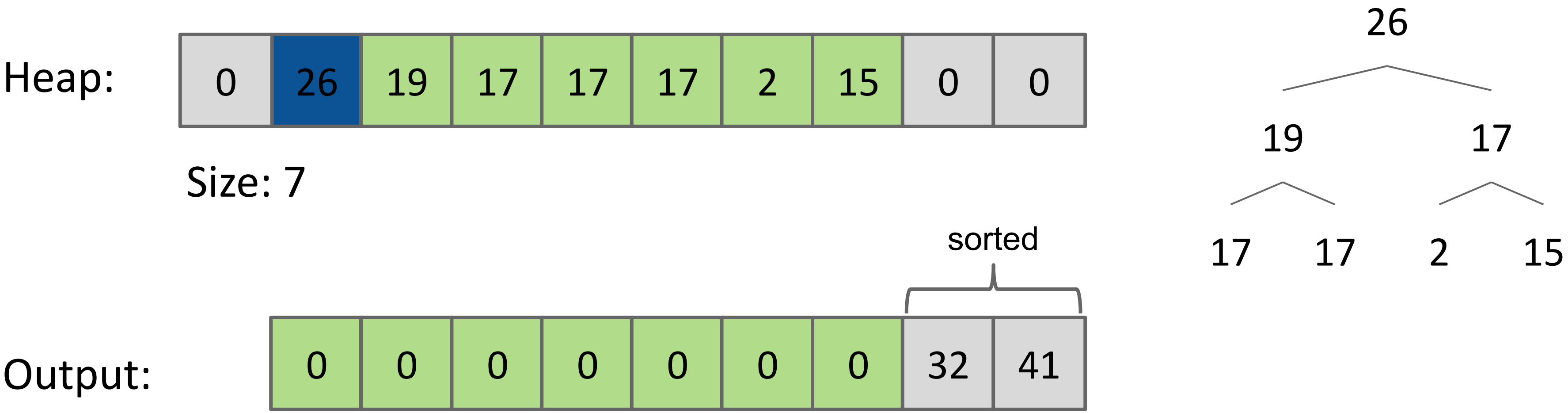
Output:

0	0	0	0	0	0	0	32	41

Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

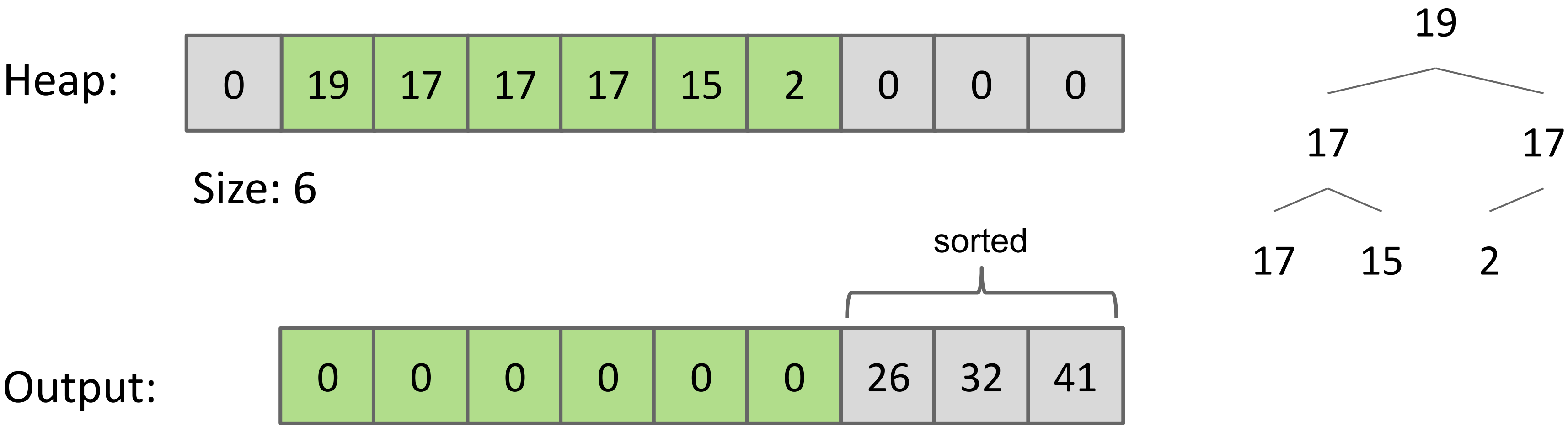
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
    - Delete largest item from the max heap.
    - Put deleted item at the end of the unused part of the output array.



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

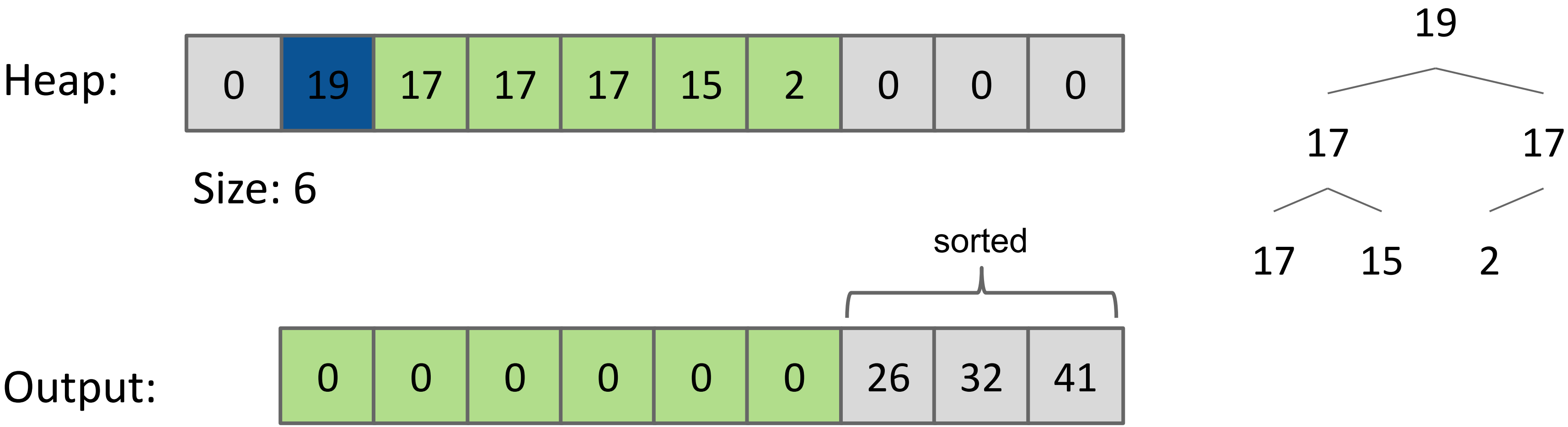
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

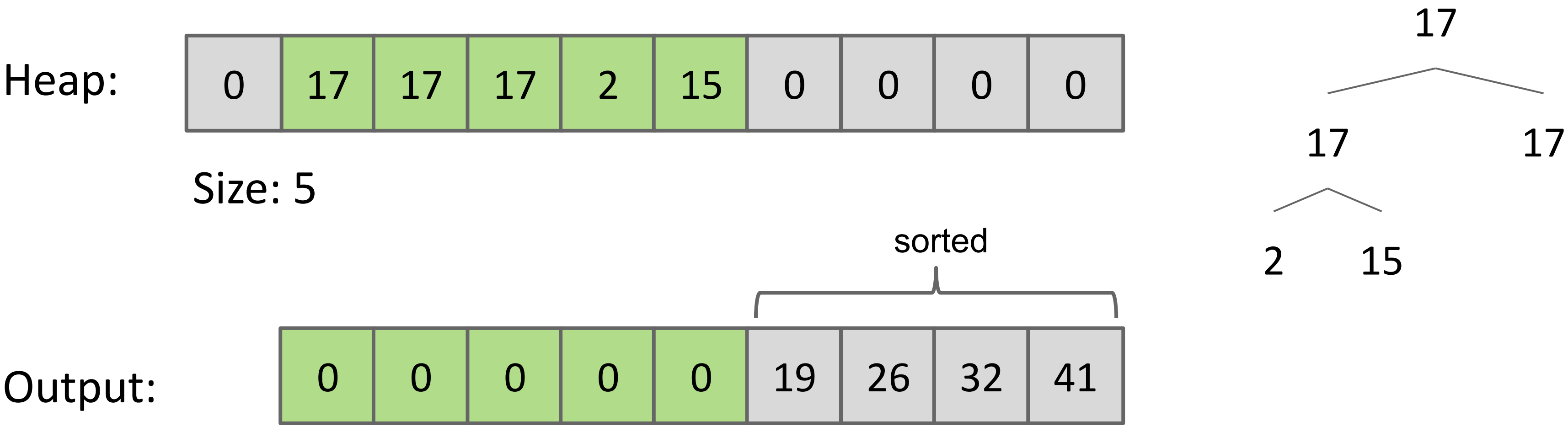
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**



Naive Heap Sort: Phase 2: Heap Deletion

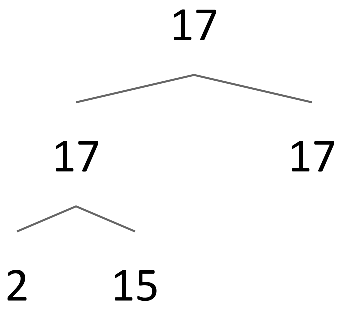
Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.

Heap:

0	17	17	17	2	15	0	0	0	0

Size: 5



sorted

Output:

0	0	0	0	0	19	26	32	41

Naive Heap Sort: Phase 2: Heap Deletion

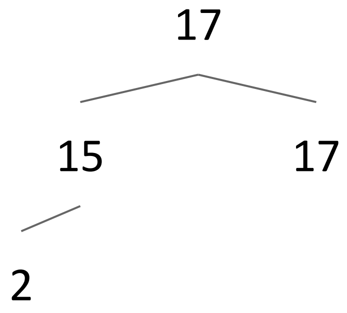
Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
    - **Delete largest item from the max heap.**
    - **Put deleted item at the end of the unused part of the output array.**

Heap:

0	17	15	17	2	0	0	0	0	0

Size: 4



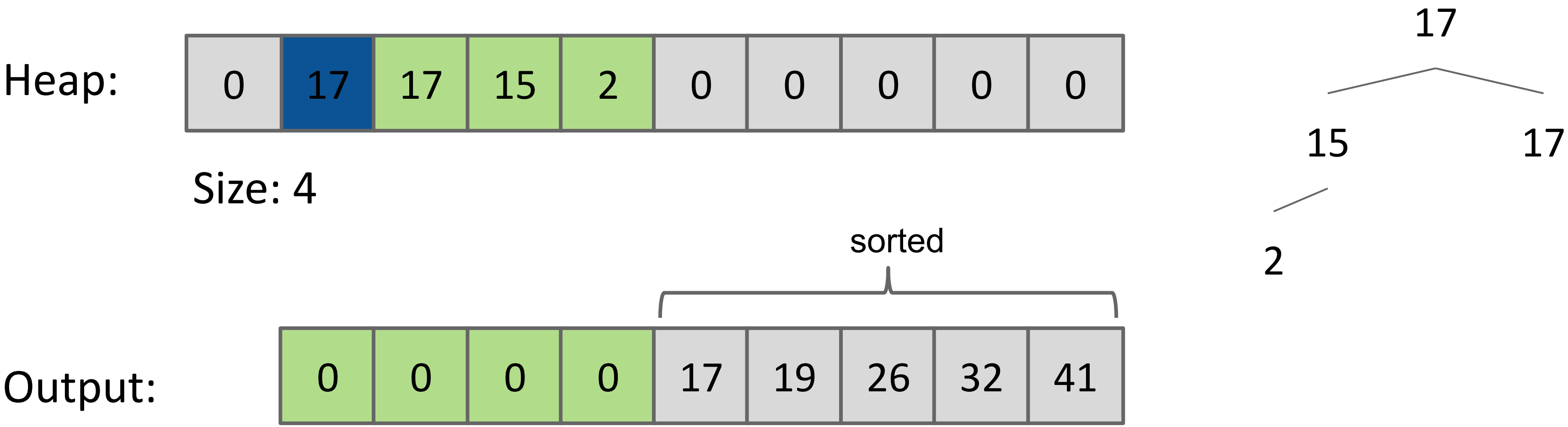
Output: $\underbrace{17 \; 19 \; 26 \; 32 \; 41}_{\text{sorted}}$

0	0	0	0	17	19	26	32	41

Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

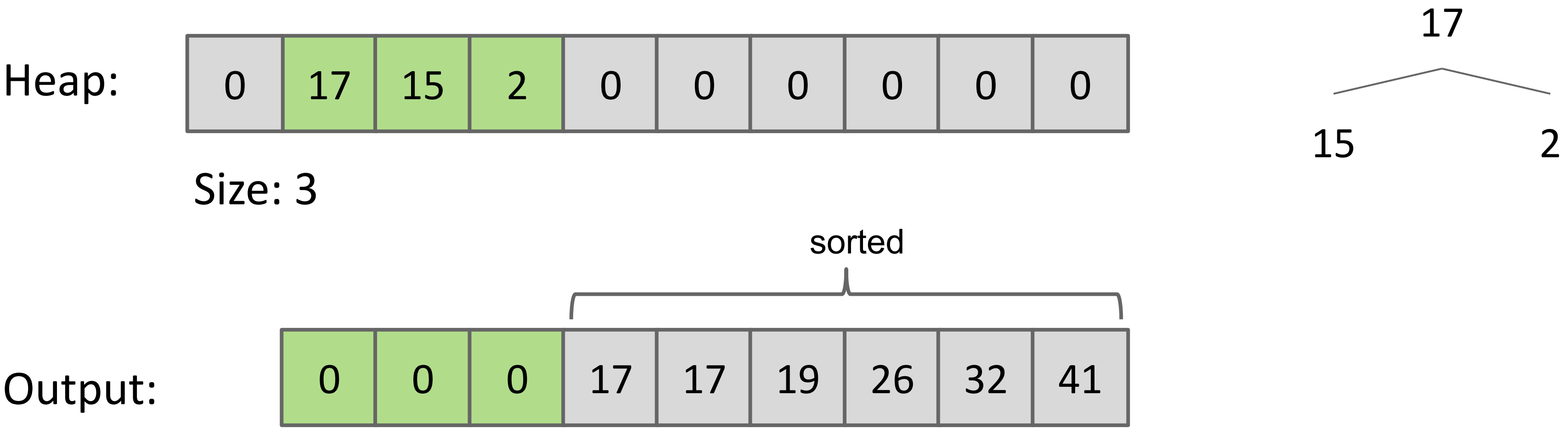
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
    - Delete largest item from the max heap.
    - Put deleted item at the end of the unused part of the output array.



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

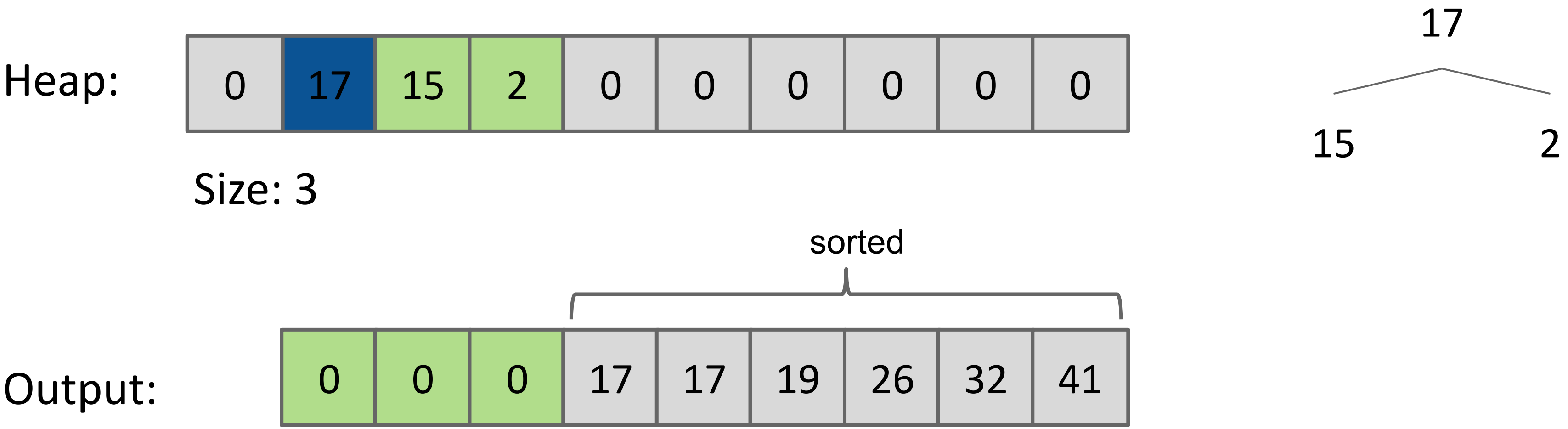
- Insert all items into a max heap, and discard input array. Create output array.
- **Repeat N times:**
    - **Delete largest item from the max heap.**
    - **Put deleted item at the end of the unused part of the output array.**



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.



Naive Heap Sort: Phase 2: Heap Deletion

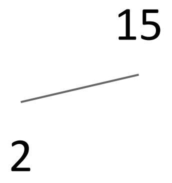
Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**

Heap:

0	15	2

Size: 2



sorted

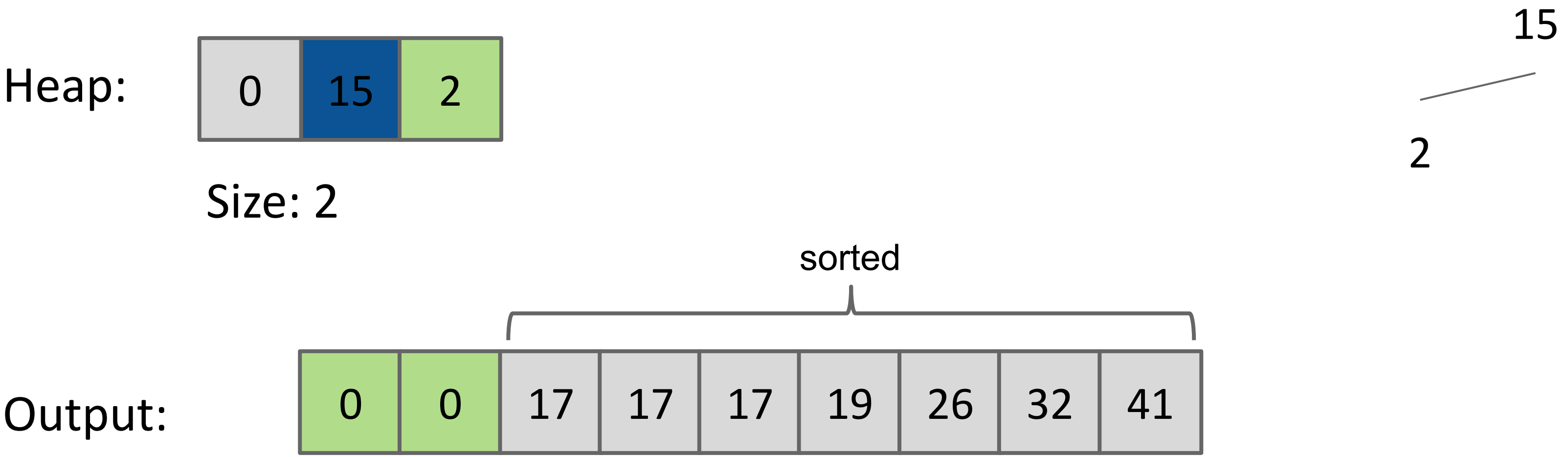
Output:

0	0	17	17	17	19	26	32	41

Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

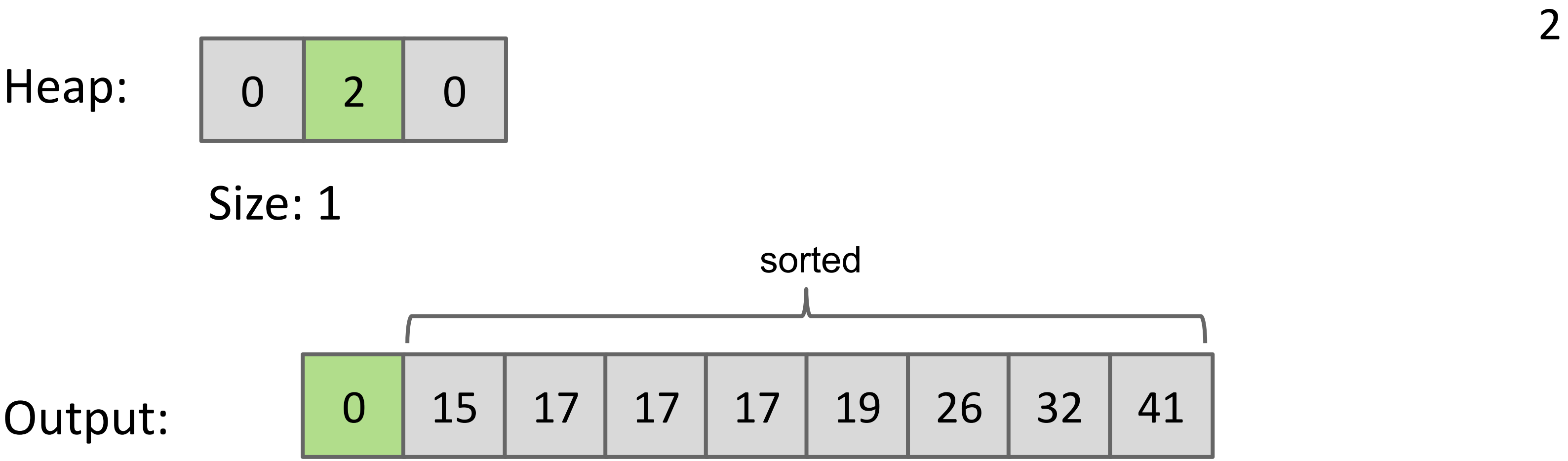
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**



Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.

2

Heap:

0	2	0

Size: 1

sorted

Output:

0	15	17	17	17	19	26	32	41

Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.

2

Heap:

0	0	0

Size: 0

sorted

Output:

2	15	17	17	17	19	26	32	41

Naive Heapsort Runtime

Heap sorting N items:

- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
    - Delete largest item from the max heap.
    - Put deleted item at the end of the unused part of the output array.

What is the TOTAL runtime of naive heapsort?

A. $\Theta(N)$

B. $\Theta(N \log N)$

C. $\Theta(N^2)$, but faster than selection sort.

Heapsort Runtime Analysis

Use the magic of the heap to sort our data.

- Getting items into the heap O(N log N) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Overall runtime is O(N log N) + Θ(N) + O(N log N) = **O(N log N)**

- Far better that selection sort!

Memory usage is Θ(N) to build the additional copy of all of our data.

- Worse than selection sort, but probably no big deal (??).
- Can eliminate this extra memory cost with same fancy trickery.

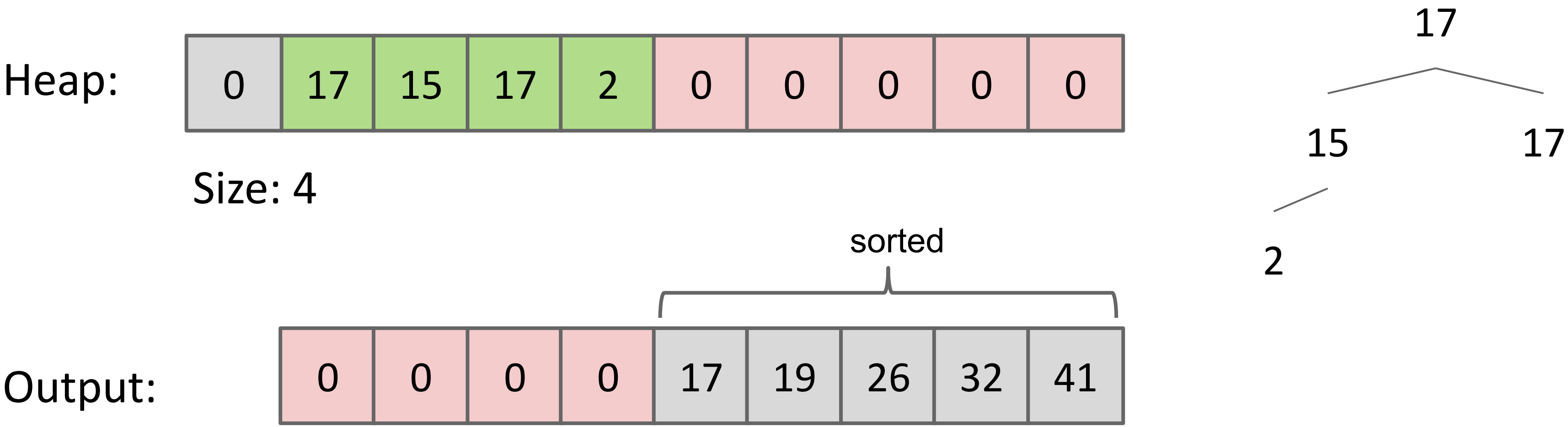
In-Place Heapsort

Lecture 28, CS61B, Spring 2025

Memory Inefficiency with Naive Heapsort

Notice here that both the heap and the output have a sequence of 0s in them

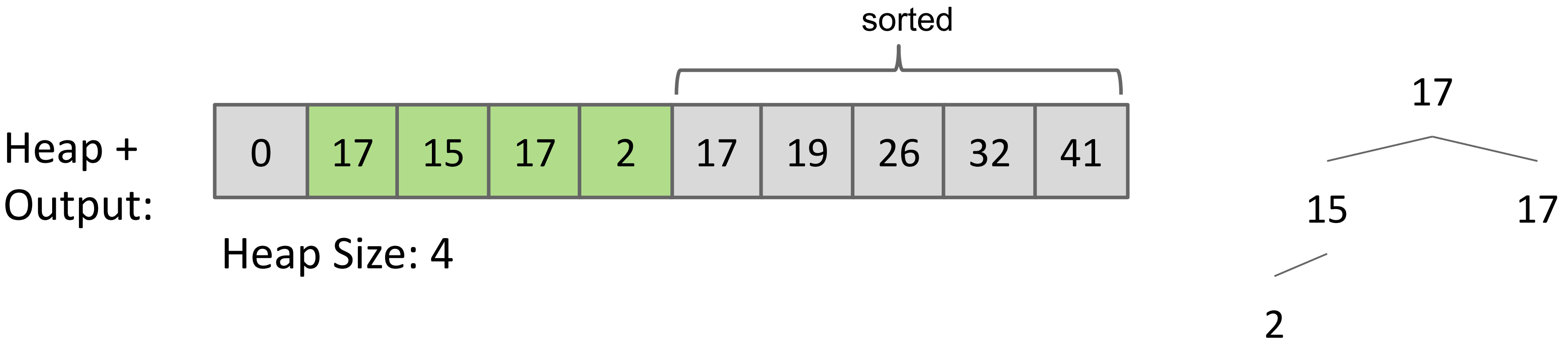
- These are kind of "filler values"; they have no meaning here, and are just placeholders
  - In theory, we don't need to store these 0s
  - Notably, at each step, the heap shrinks by 1 item, and the output grows by one item



Memory Inefficiency with Naive Heapsort

Idea: Store the heap and the output in the same array, so we save memory

1. Convert input into heap (ideally in-place so no memory is used)
2. Repeat N times:
   a. Delete largest item from the max heap, and move deleted item to vacated array slot. (Uses no extra memory!)



Heapification

Step 1: Convert input array into a heap

Two parameters we can play with here:

- Min Heap vs Max Heap
- Build the heap from the root down (top down heapification) vs build the heap from the leaves up (bottom up heapification)

In-place heap sort: [Demo]

32	15	2	17	19	26	41	17	17

heapification

41	19	32	17	15	26	2	17	17

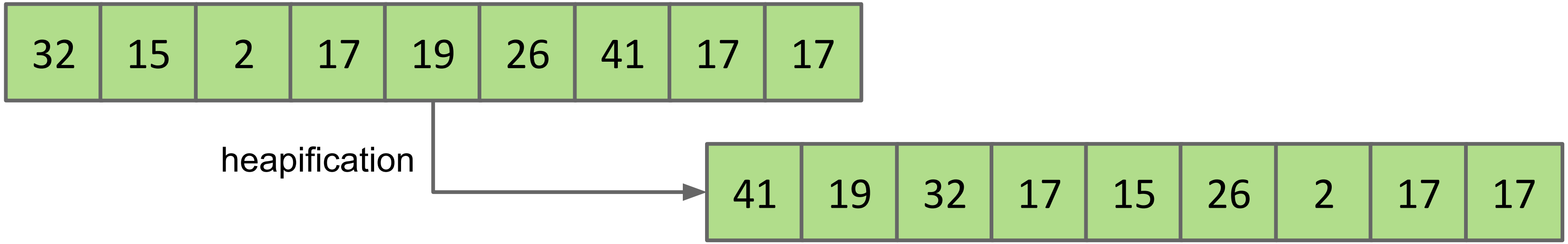
For this algorithm we don't leave spot 0 blank (since that requires making a new, larger array).

Step 1: Convert input array into a heap

Two parameters we can play with here:

- Min Heap vs **Max Heap** (since we always want to move the largest remaining item to the vacated spot)
- Top Down heapification vs **Bottom Up heapification** (Asymptotically faster)

In-place heap sort: Demo



For this algorithm we don't leave spot 0 blank (since that requires making a new array).

In-place Heap Sort

Heap sorting N items:

- Bottom-up heapify input array.
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.

Input:

32	15	2	17	19	26	41	17	17

In-place Heap Sort: Phase 1: Heapification

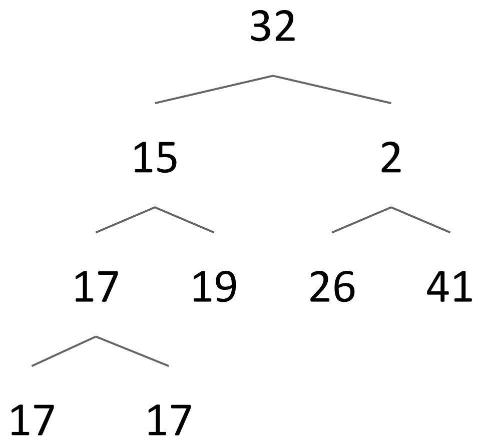
Heap sorting N items:

- **Bottom-up heapify input array:**
  - Sink nodes in reverse level order: sink(k)
  - After sinking, guaranteed that tree rooted at position k is a heap.

Note: This is not a heap yet!
That's why we're heapifying.

Input:

32	15	2	17	19	26	41	17	17



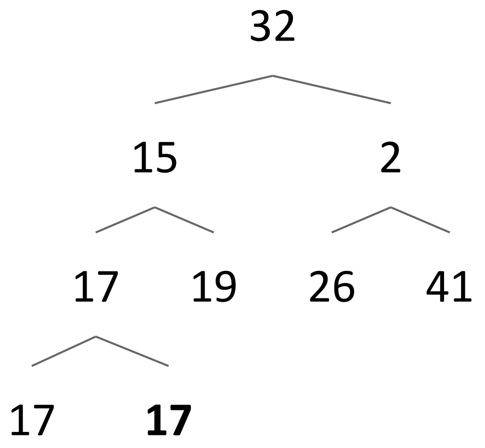
In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.

Input:

32	15	2	17	19	26	41	17	17

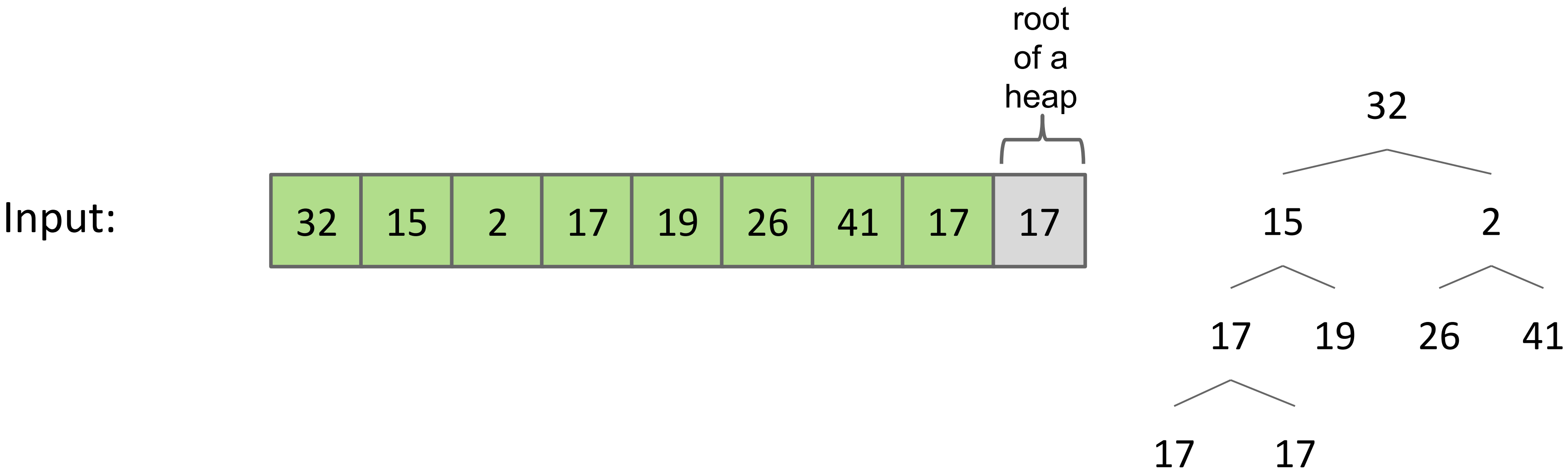


Sinking 17 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

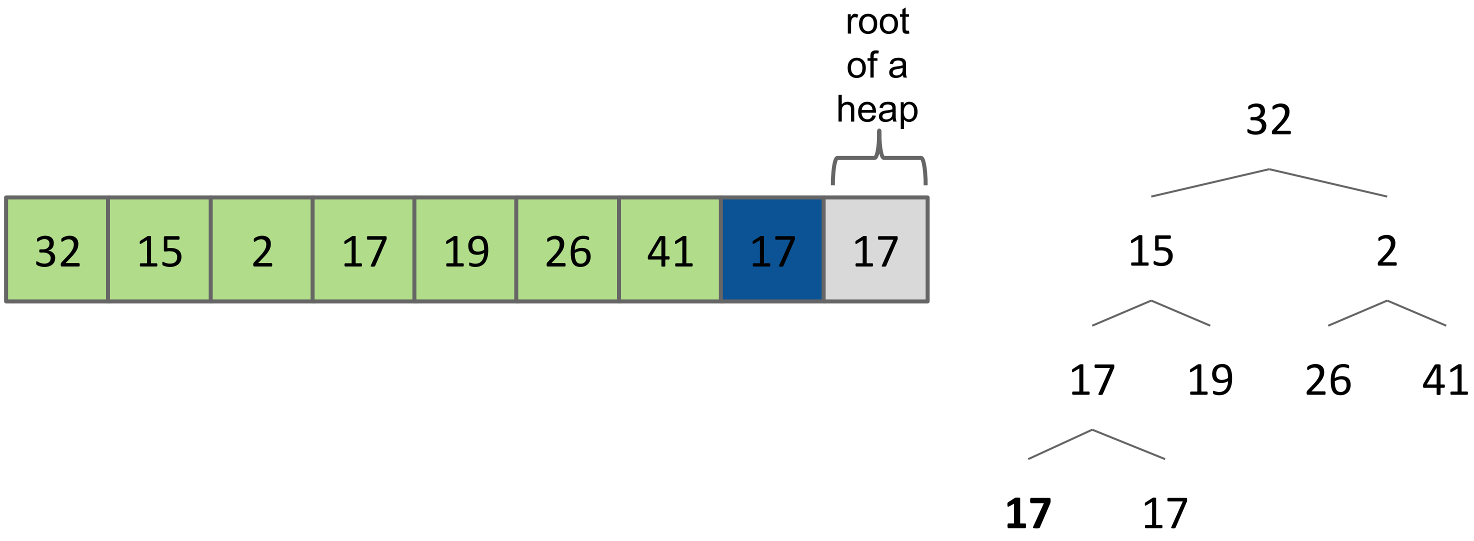


In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
    - **Sink nodes in reverse level order: sink(k)**
    - After sinking, guaranteed that tree rooted at position k is a heap.

Input:

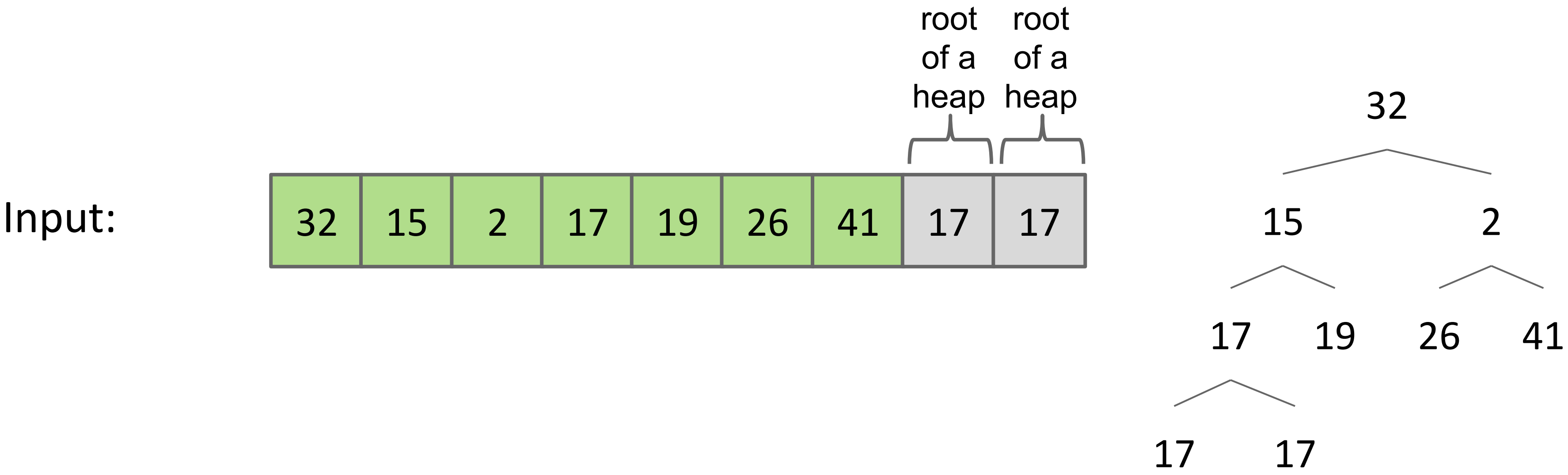


Sinking 17 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

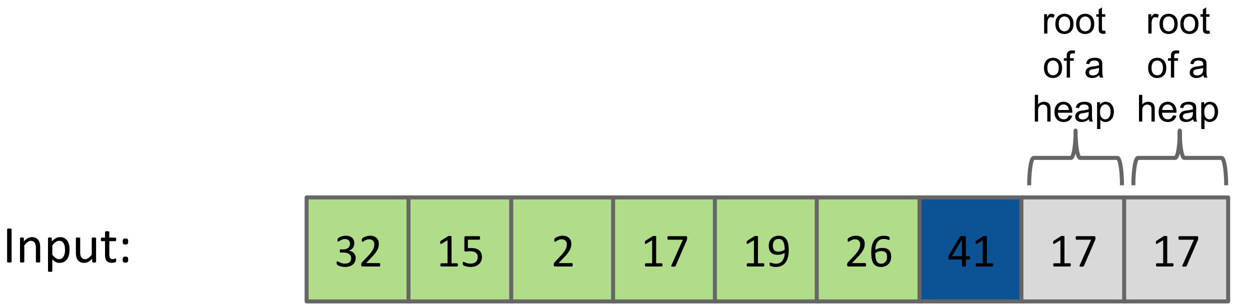
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

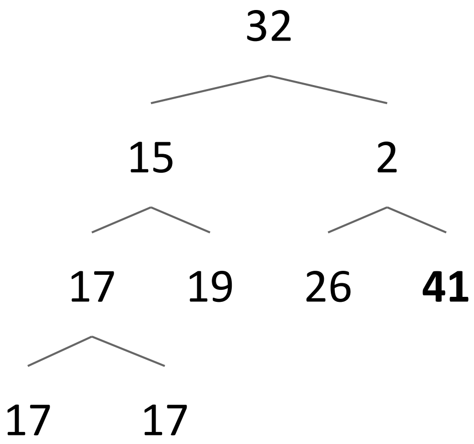


In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.



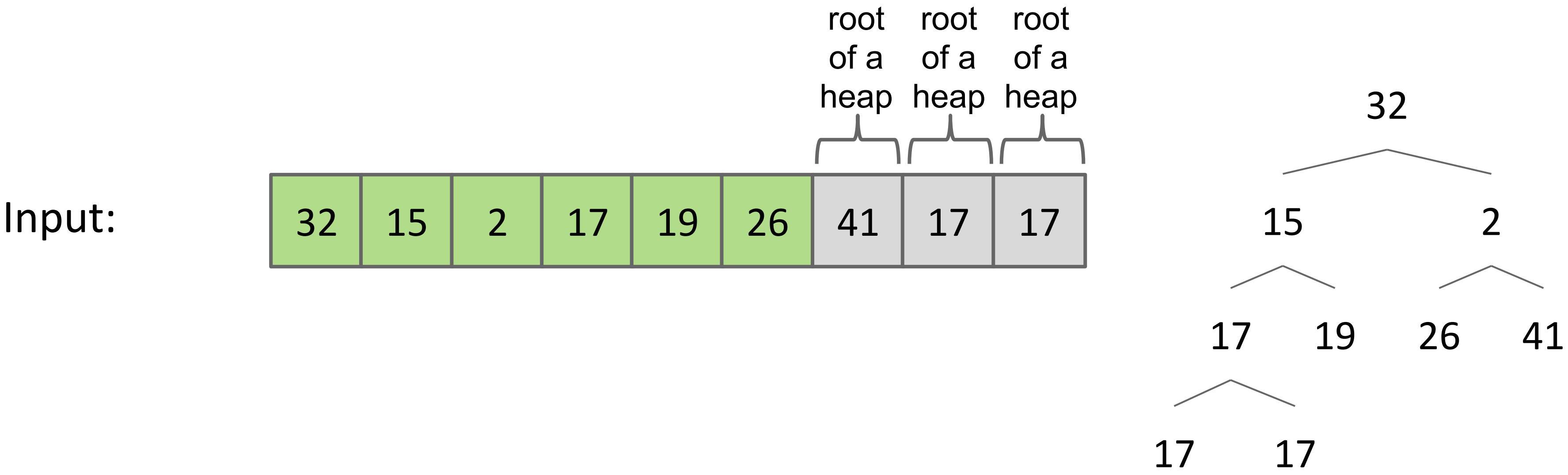


Sinking 41 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

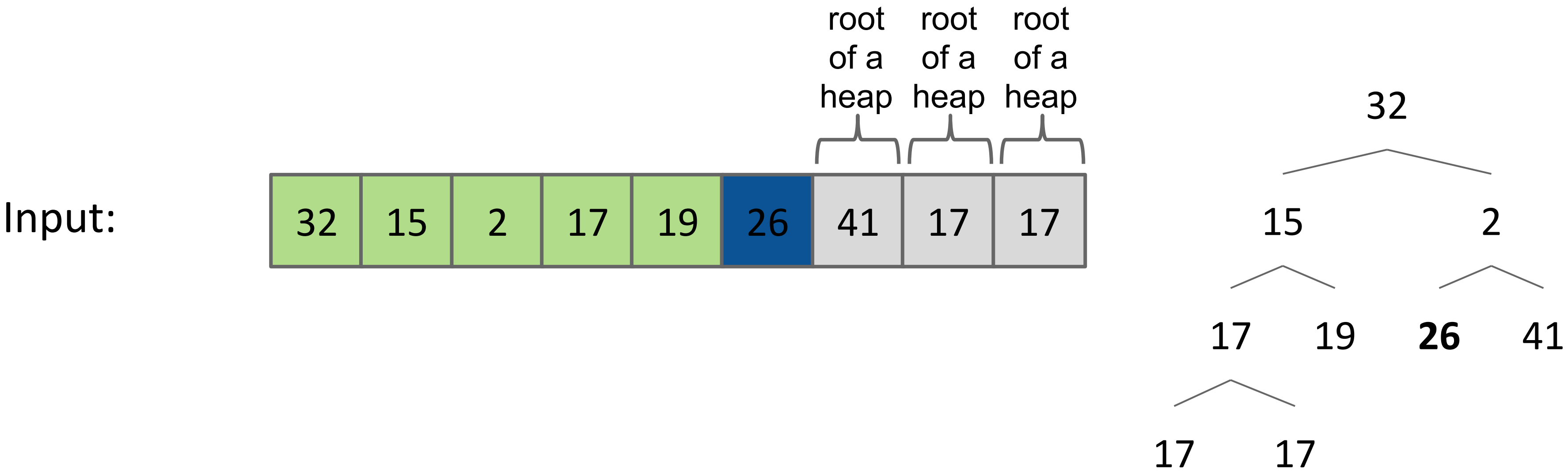
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**



In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.



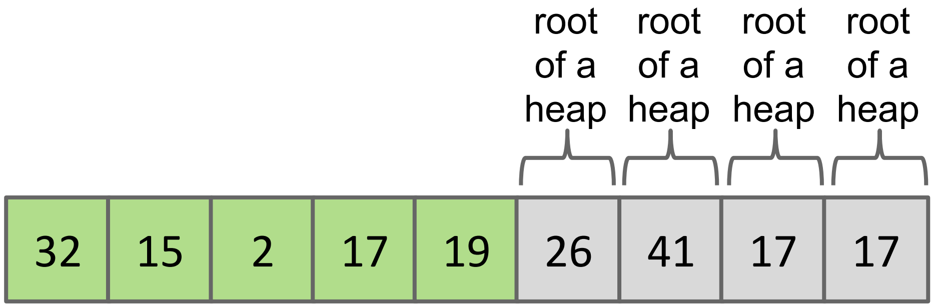
Sinking 26 has no effect.

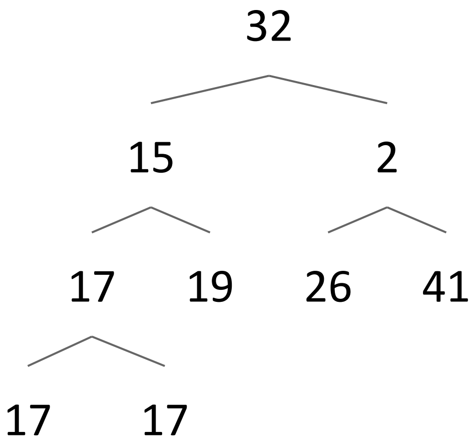
In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

Input:



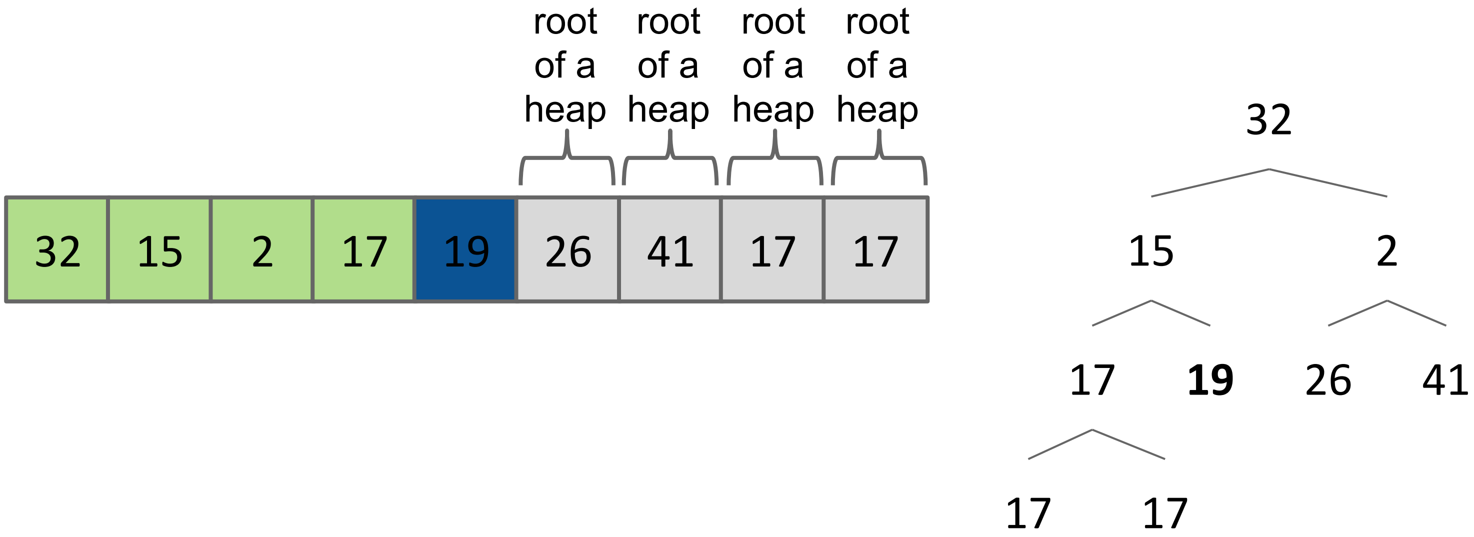


In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.

Input:



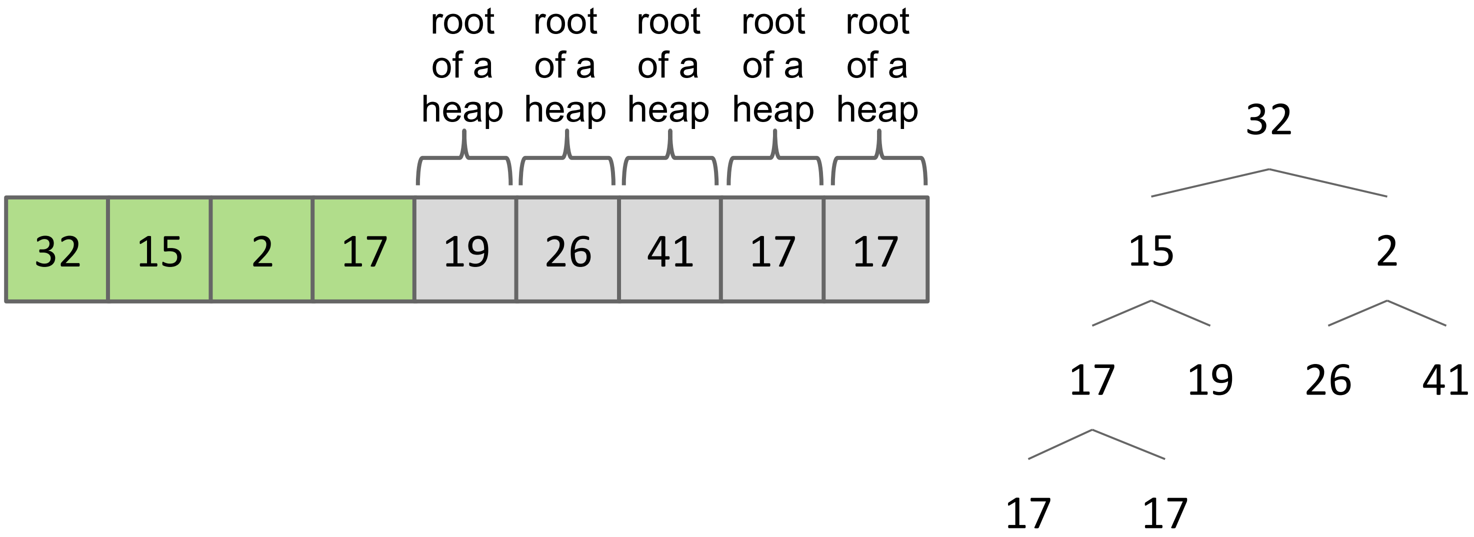
Sinking 19 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

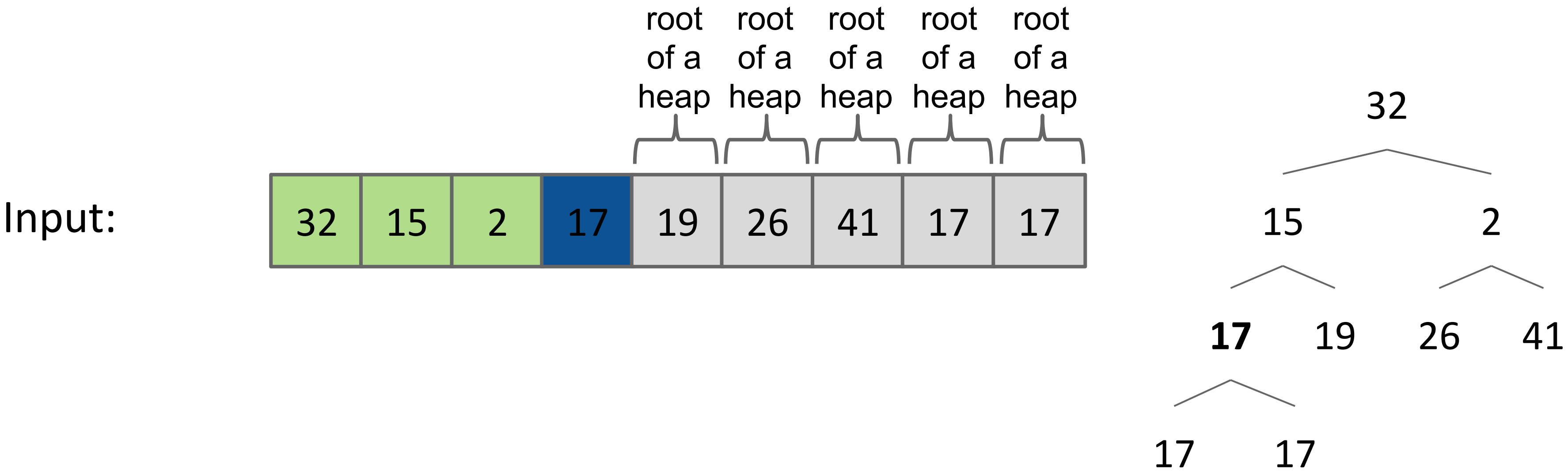
Input:



In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.



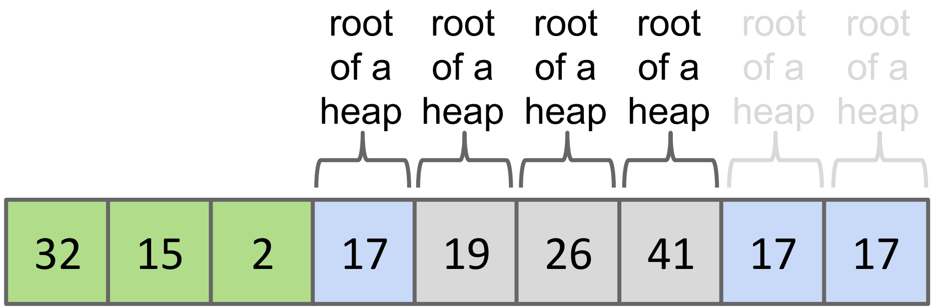
Sinking 17 has no effect.

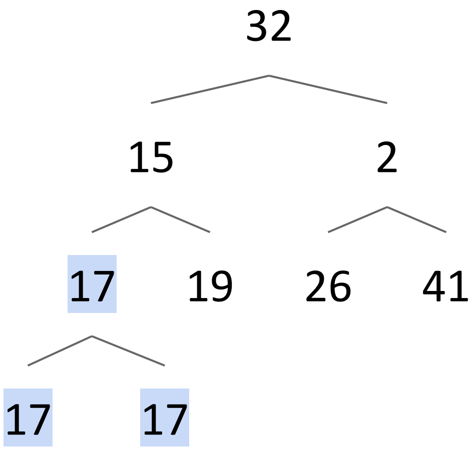
In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

Input:





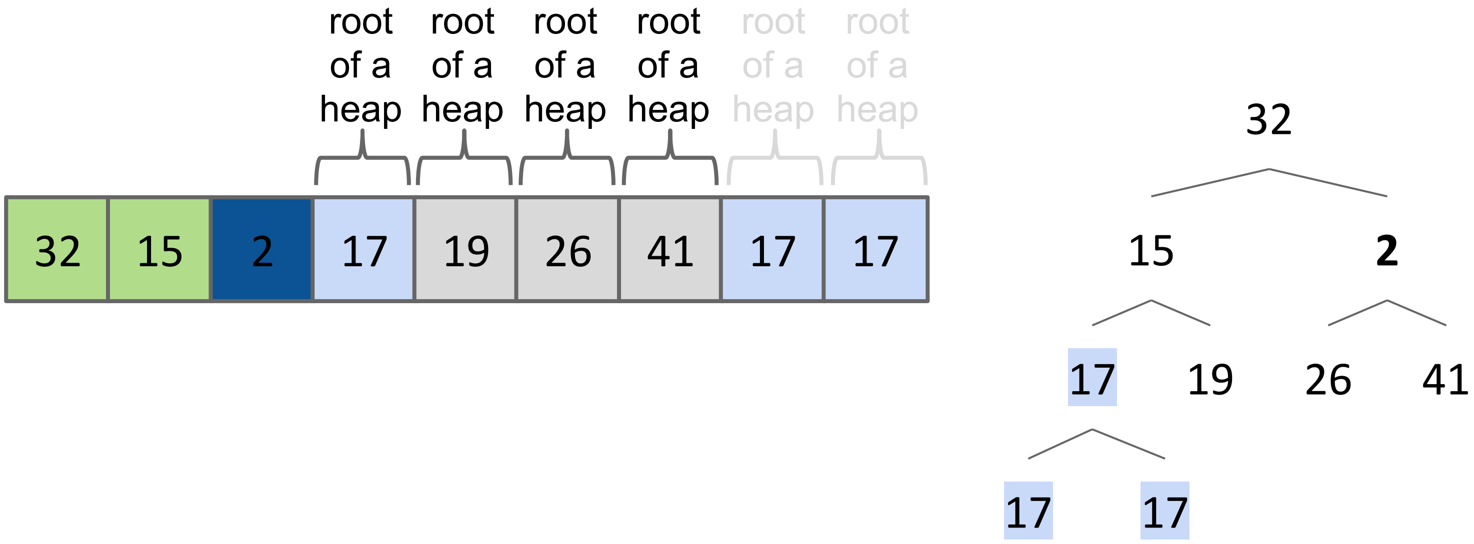
The blue coloring is to make it clear that the three 17s are all part of the same heap. I've also grayed out the "root of a heap" statement about the last two 17s since this is redundant information (all subheap nodes are also roots of that subheap).

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.

Input:

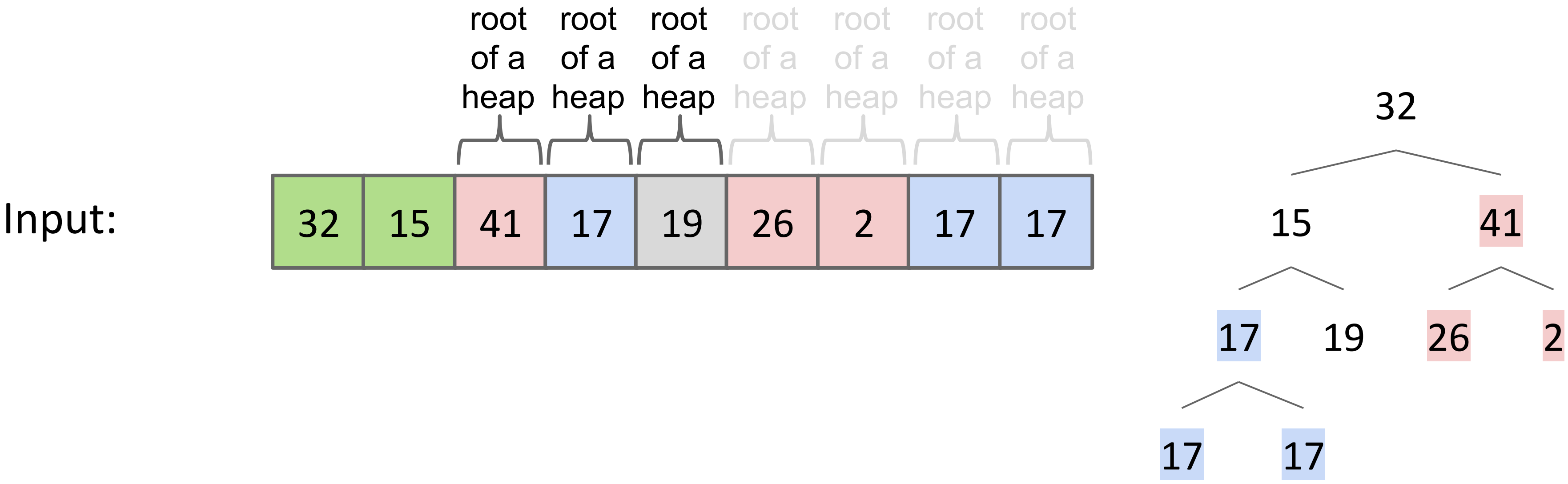


Sinking 2 does something!

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

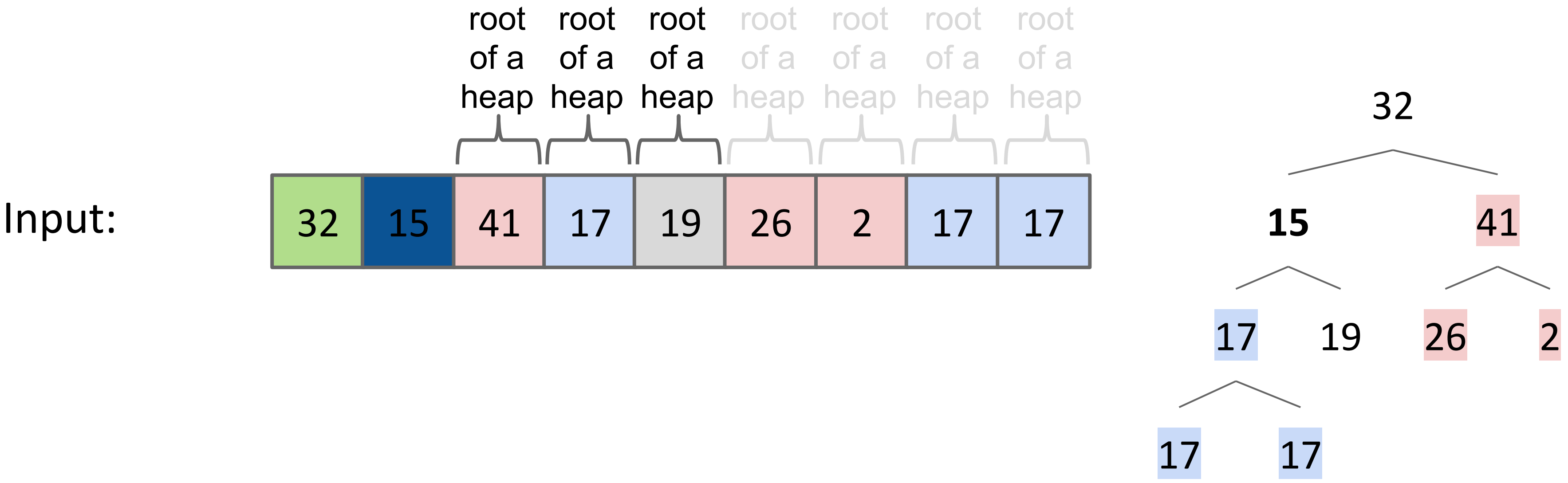
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**



In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.

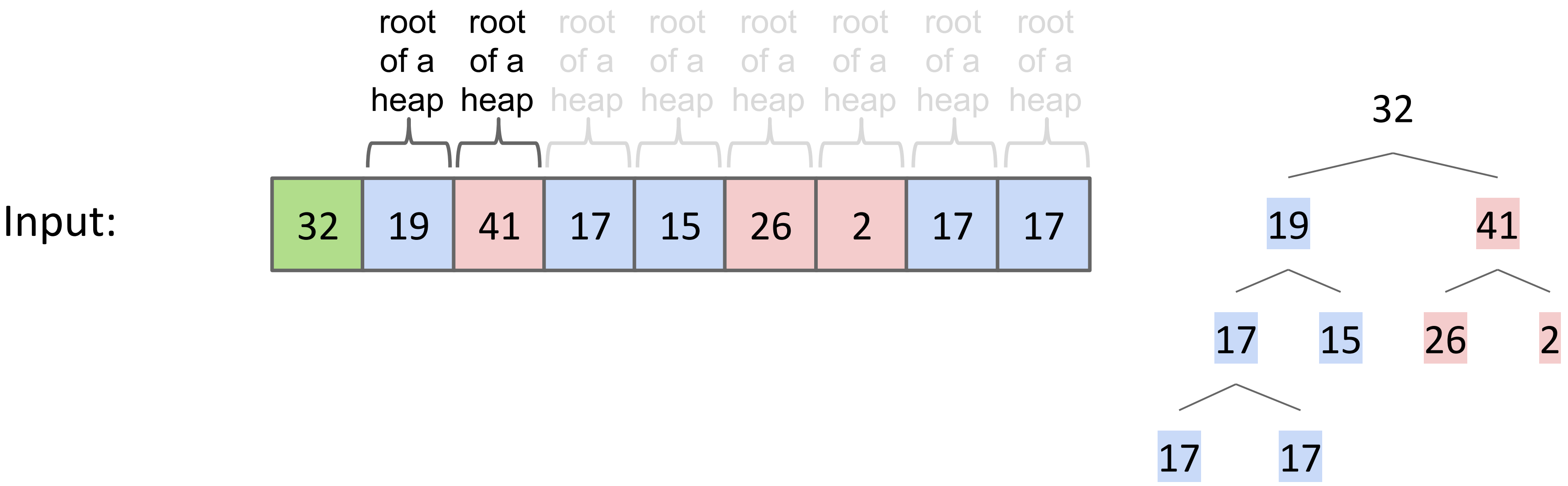


Sinking 15 does something!

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

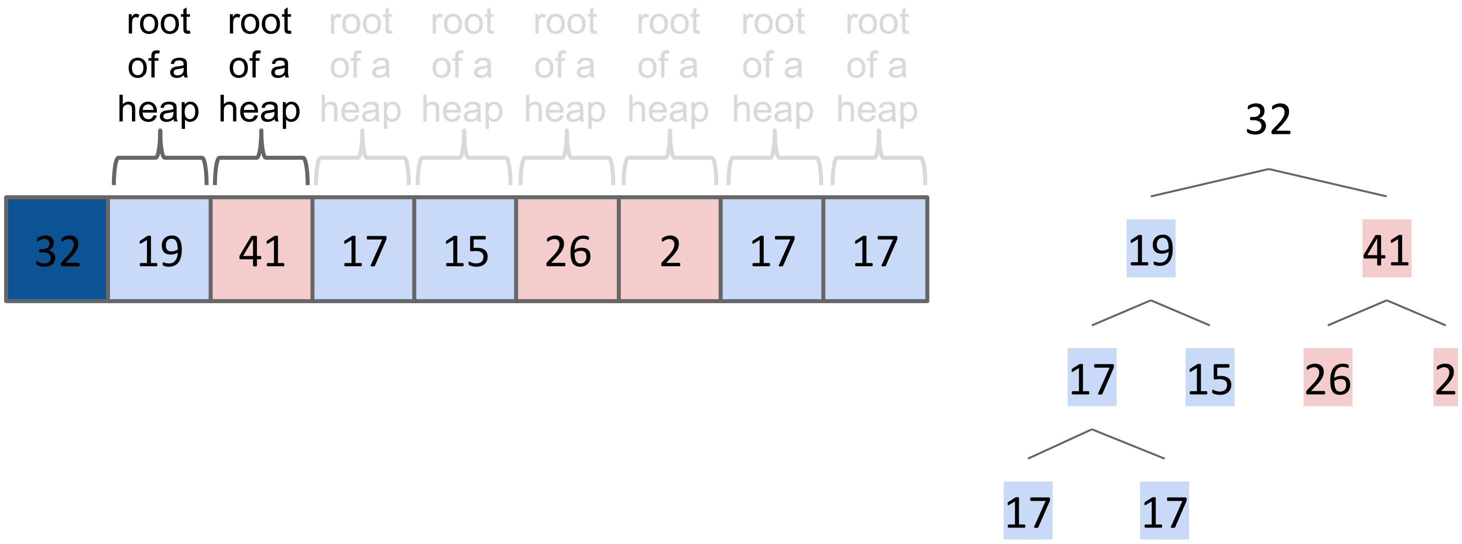


In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.

Input:



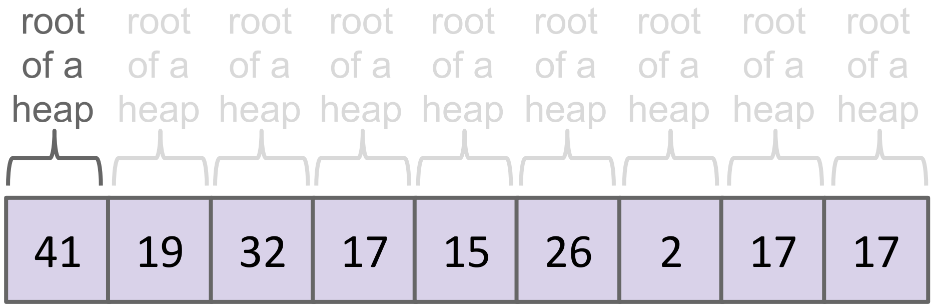
Sinking 32 does something!

In-place Heap Sort: Phase 1: Heapification

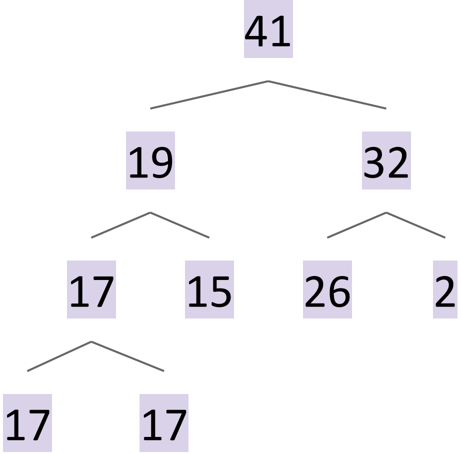
Heap sorting N items:

- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

Input:



(No room to leave an unused, spot, so we will actually use position zero for this algorithm!)



Punchline: Since tree rooted at position 0 is the root of a heap, then entire array is a heap.

In-place Heap Sort

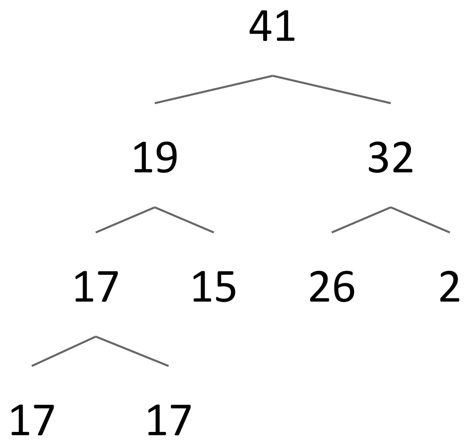
Heap sorting N items:

- **Bottom-up heapify input array (done!).**
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.

Input:

41	19	32	17	15	26	2	17	17

Size: 9



In-place Heap Sort

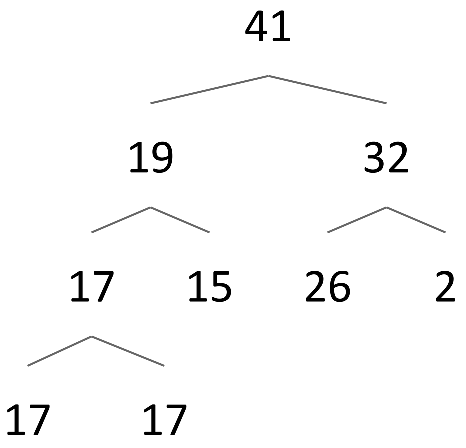
Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
    - Delete largest item from the max heap, swapping root with last item in the heap.

Input:

41	19	32	17	15	26	2	17	17

Size: 9

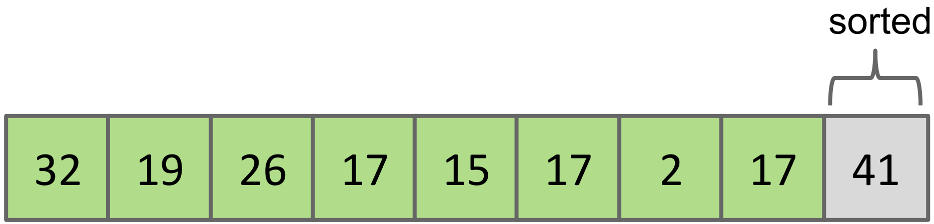


In-place Heap Sort

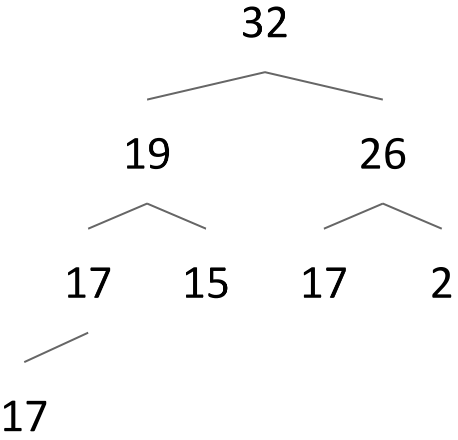
Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
  - **Delete largest item from the max heap, swapping root with last item in the heap.**

Input:



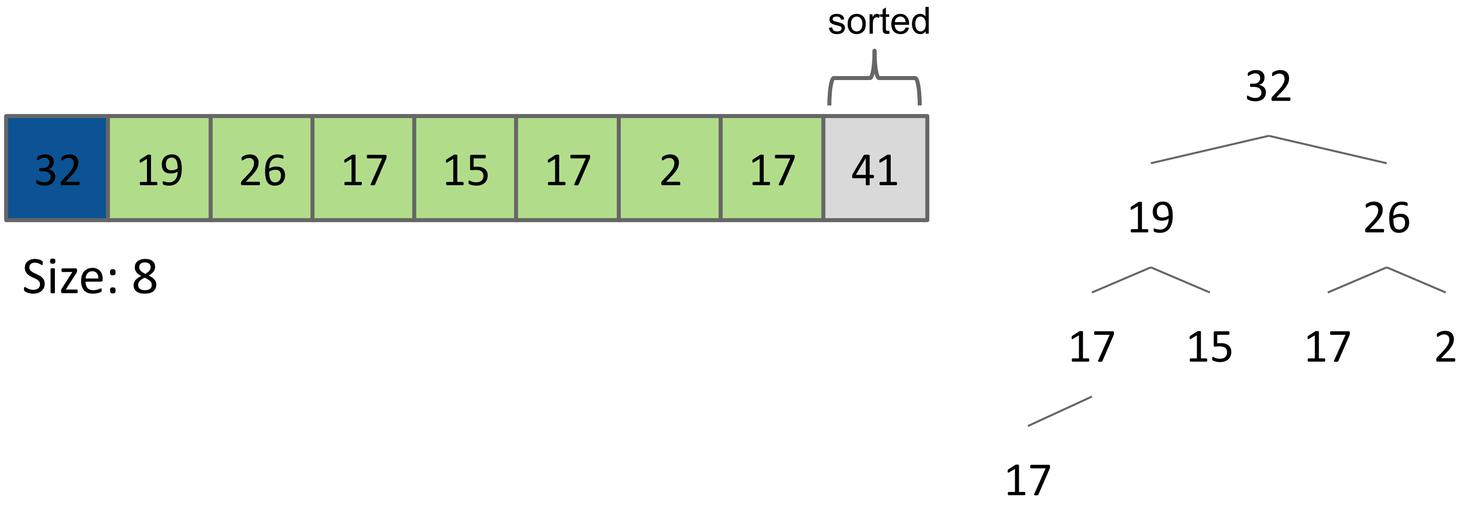
Size: 8



Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.

Input:

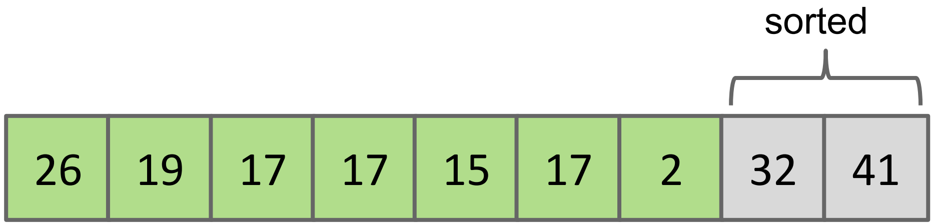


In-place Heap Sort

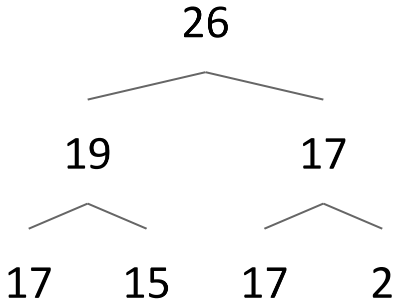
Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
    - **Delete largest item from the max heap, swapping root with last item in the heap.**

Input:



Size: 7

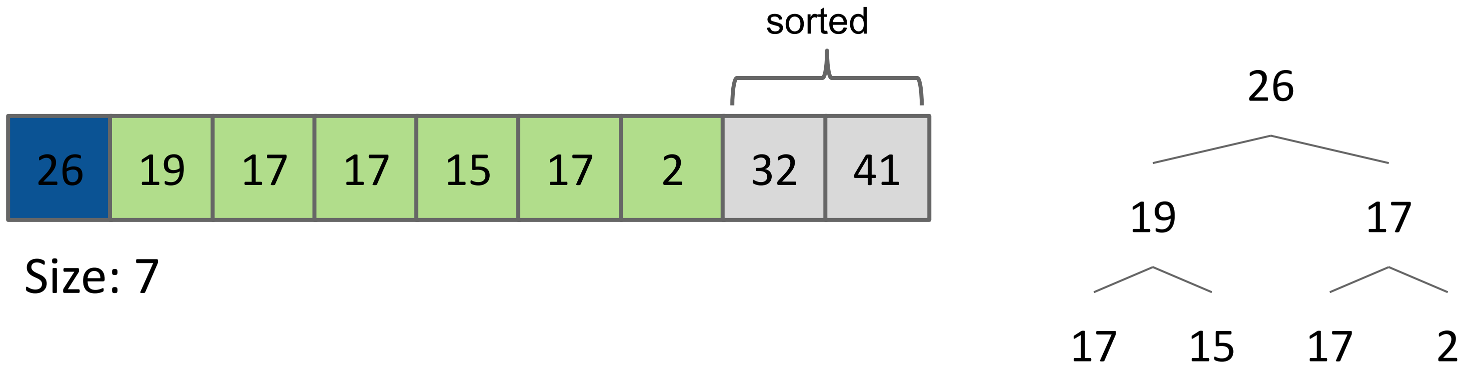


In-place Heap Sort

Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.

Input:



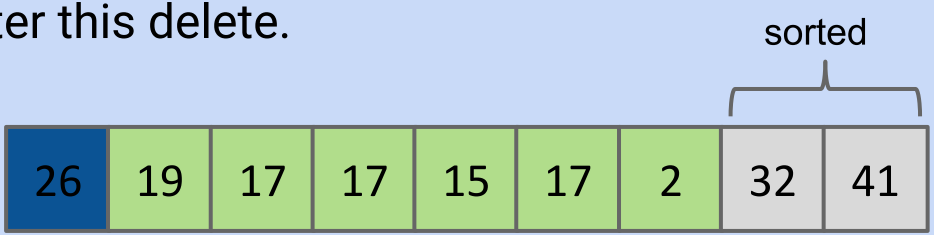
In-place Heap Sort

Heap sorting N items:

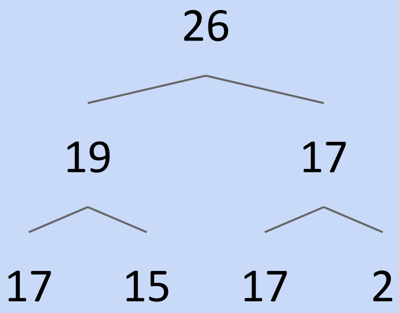
- Bottom-up heapify input array (done!).
- Repeat N times:
  - **Delete largest item from the max heap, swapping root with last item in the heap.**

Give the array after this delete.

Input:



Size: 7

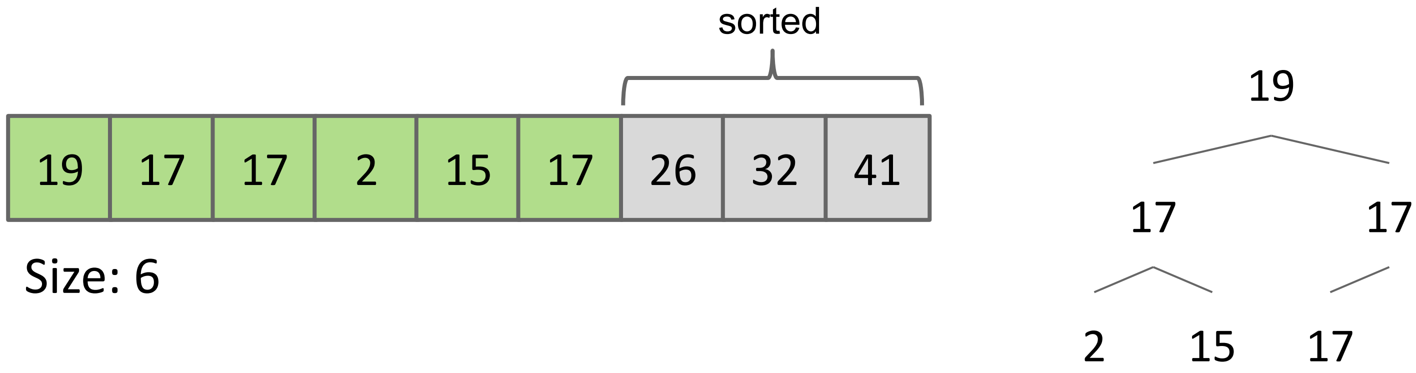


In-place Heap Sort

Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
  - Delete largest item from the max heap, swapping root with last item in the heap.

Input:



Size: 6

From here on out, the process is just the same, so verbose steps are omitted...

In-place Heap Sort

Heap sorting N items:

- Bottom-up heapify input array (done!).
- Repeat N times:
    - Delete largest item from the max heap, swapping root with last item in the heap.

sorted

Input:

2	15	15	17	17	19	26	32	41

Size: 0

Heapsort Runtime

Lecture 28, CS61B, Spring 2025

In-place Heapsort Runtime

Use the magic of the heap to sort our data.

- Bottom-up Heapification: O(???) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Give the time complexity of in-place heapsort in big O notation.

A. O(N)
B. O(N log N)
C. $O(N^2)$

In-place Heapsort Runtime

Use the magic of the heap to sort our data.

- Bottom-up Heapification: O(N log N) time.
- Selecting *largest* item: Θ(1) time.
- Removing *largest* item: O(log N) for each removal.

Give the time complexity of in-place heapsort in big O notation.

A. O(N log N)

Bottom-up heapification is N sink operations, each taking no more than O(log N) time, so overall runtime for heapification is O(N log N).

- More extra for experts, show heapsort is Θ(N log N) in the worst case.
- More extra for experts, show bottom-up Heapification is Θ(N) time.

In-place Heapsort

What is the **memory complexity** of Heapsort?

- Also called “space complexity”.

A. $\Theta(1)$

B. $\Theta(\log N)$

C. $\Theta(N)$

D. $\Theta(N \log N)$

E. $\Theta(N^2)$

In-place Heapsort

What is the **memory complexity** of Heapsort?

- Also called “space complexity”.

A. Θ(1)

B. Θ(log N)

C. Θ(N)

D. Θ(N log N)

E. $\Theta(N^2)$

The only extra memory we need is a constant number instance variables, e.g. size.

- Unimportant caveat: If we employ recursion to implement various heap operations, space complexity is Θ(log N) due to the need to track recursive calls. The difference between Θ(log N) and Θ(1) space is effectively nothing.

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$**	Link	Bad cache (61C) performance.

*: An array of all duplicates yields linear runtime for heapsort.
**: Assumes heap operations implemented iteratively, not recursively.

Mergesort

Lecture 28, CS61B, Spring 2025

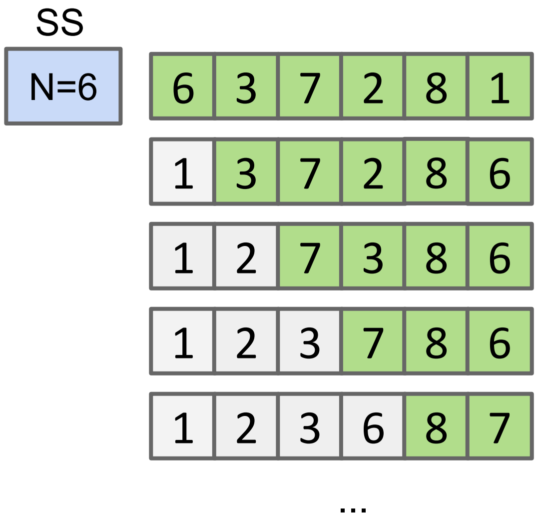
Selection Sort: A Prelude to Mergesort

Earlier we discussed a sort called selection sort:

- Find the smallest unfixed item, move it to the front, and 'fix' it.
- Sort the remaining unfixed items using selection sort.

Runtime of selection sort is $\Theta(N^2)$:

- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- …
- Look at last two unfixed items.
- Done, sum is $2+3+4+5+\ldots+N = \Theta(N^2)$



Selection Sort: A Prelude to Mergesort/Example 5

Earlier in class we discussed a sort called selection sort:

- Find the smallest unfixed item, move it to the front, and 'fix' it.
- Sort the remaining unfixed items using selection sort.

Runtime of selection sort is $\Theta(N^2)$:

- Look at all N unfixed items to find smallest.
- Then look at N-1 remaining unfixed.
- ...
- Look at last two unfixed items.
- Done, sum is $2+3+4+5+...+N = \Theta(N^2)$

SS

~36 AU N=6

SS

~4096 AU N=64

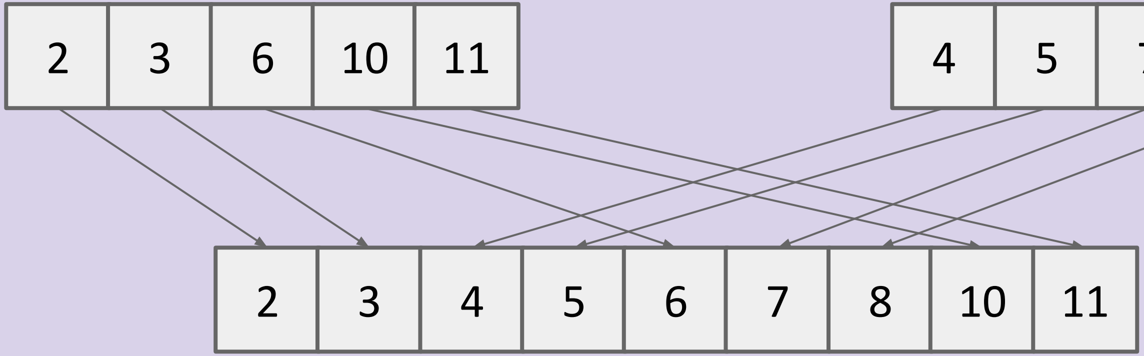
Given that runtime is quadratic, for N = 64, we might say the runtime for selection sort is 4,096 arbitrary units of time (AU).

The Merge Operation: Another Prelude to Mergesort/Example 5

Given two sorted arrays, the merge operation combines them into a single sorted array by successively copying the smallest item from the two arrays into a target array.

Merging Demo (Link)

Merge Runtime



How does the runtime of merge grow with N, the total number of items?

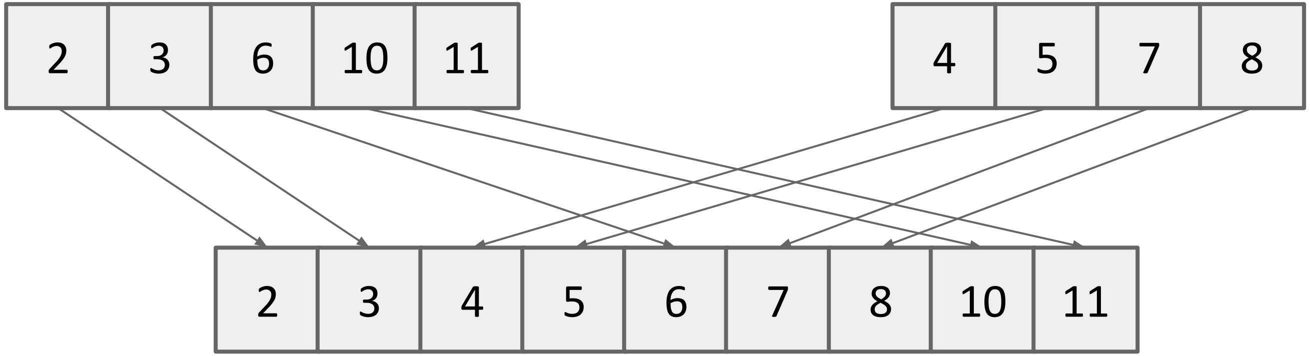
A. Θ(1)

B. Θ(log N)

C. Θ(N)

D. $\Theta(N^2)$

Merge Runtime



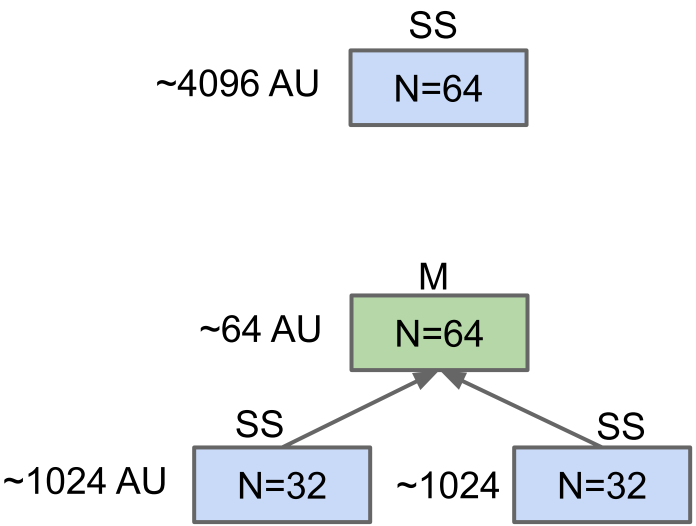
How does the runtime of merge grow with N, the total number of items?

C. Θ(N). Why? Use array writes as cost model, merge does exactly N writes.

Using Merge to Speed Up the Sorting Process

Merging can give us an improvement over vanilla selection sort:

- Selection sort the left half: $\Theta(N^2)$.
- Selection sort the right half: $\Theta(N^2)$.
- Merge the results: $\Theta(N)$.



N=64: ~2112 AU.

- Merge: ~64 AU.
- Selection sort: ~2*1024 = ~2048 AU.

Still $\Theta(N^2)$, but faster since $N+2*(N/2)^2 < N^2$

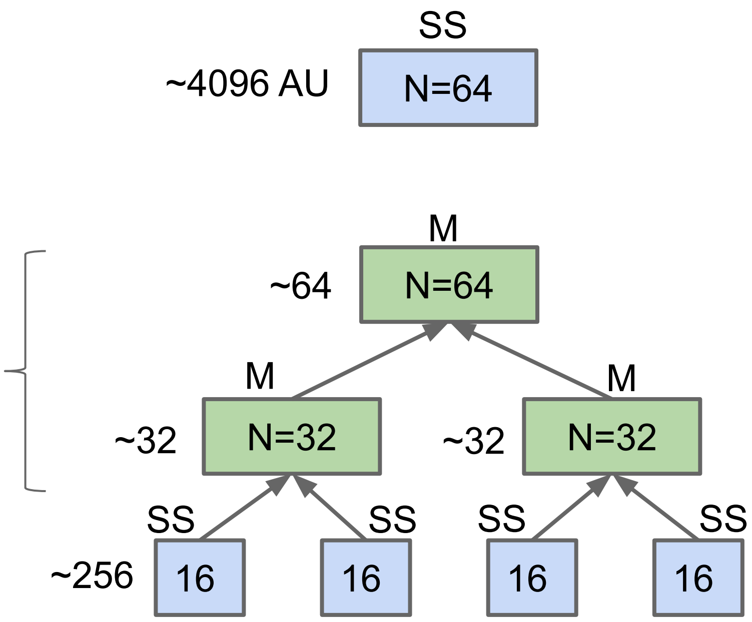
- ~2112 vs. ~4096 AU for N=64.

Two Merge Layers

Can do even better by adding a second layer of merges.

Runtime for each sort:

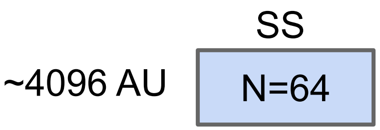
- Selection sort only: ~4096 AU.
- One layer of merges: ~2112 AU.
- Two layers of merges: ~1152 AU.
    - Merge: ~64 AU + 2*~32 AU.
    - Selection sort: 4*~256.



Example 5: Mergesort

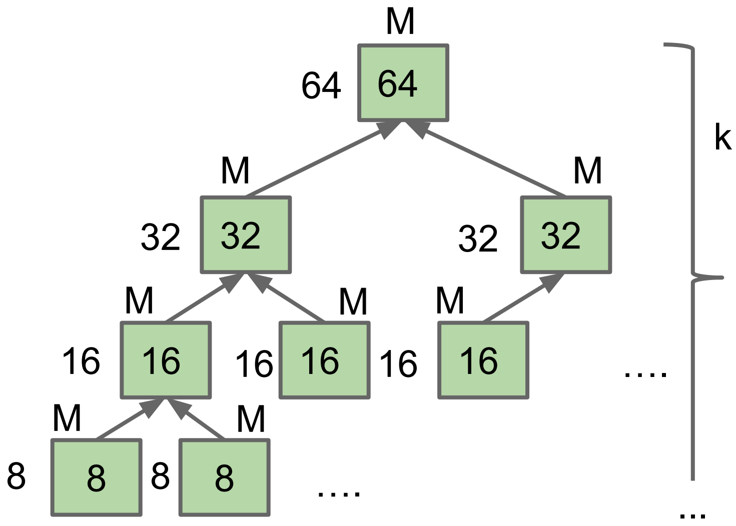
Mergesort does merges all the way down (no selection sort):

- If array is of size 1, return.
- Mergesort the left half: Θ(??).
- Mergesort the right half: Θ(??).
- Merge the results: Θ(N).



Total runtime to merge all the way down: ~384 AU

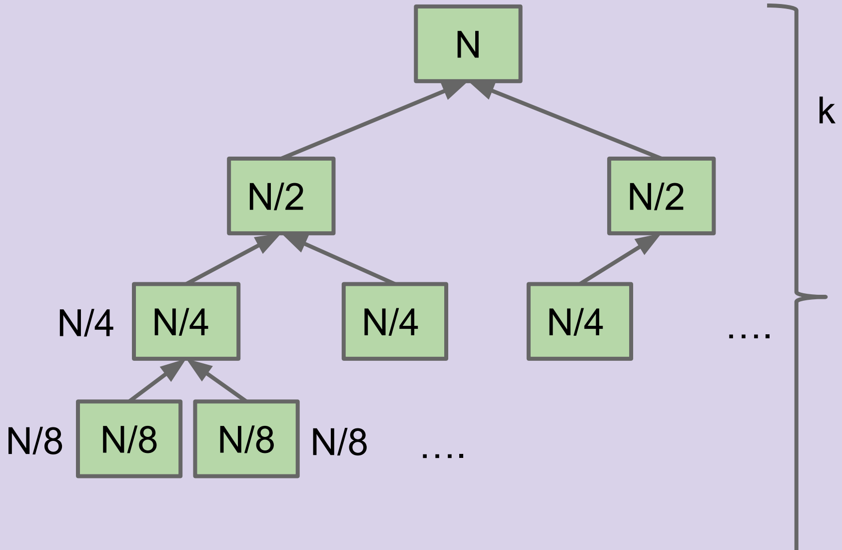
- Top layer: ~64 = 64 AU
- Second layer: ~32*2 = 64 AU
- Third layer: ~16*4 = 64 AU
- Overall runtime in AU is ~64k, where k is the number of layers.
- $k = \log_2(64) = 6$, so ~384 total AU.



Example 5: Mergesort Order of Growth

For an array of size N, what is the worst case runtime of Mergesort?

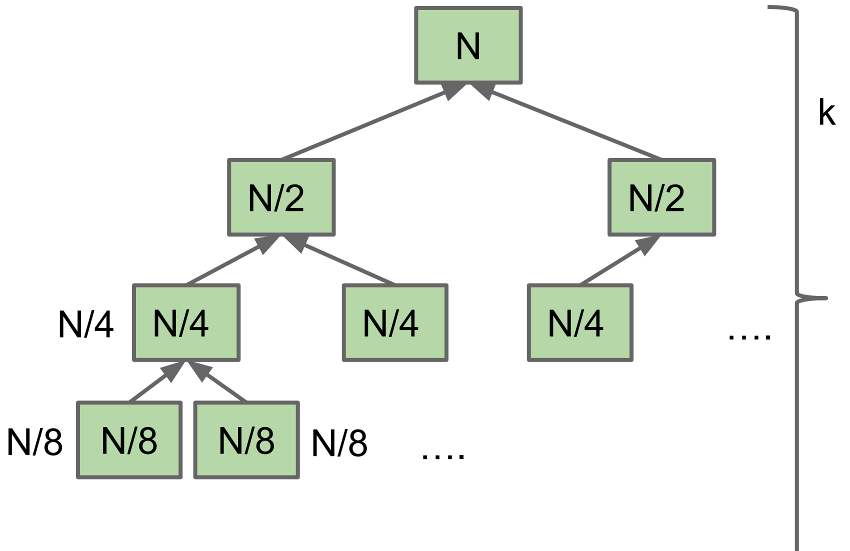
A. $\Theta(1)$
B. $\Theta(\log N)$
C. $\Theta(N)$
D. $\Theta(N \log N)$
E. $\Theta(N^2)$



Example 5: Mergesort Order of Growth

Mergesort has worst case runtime = $\Theta(N \log N)$.

- Every level takes ~N AU.
  - Top level takes ~N AU.
  - Next level takes ~N/2 + ~N/2 = ~N.
  - One more level down: ~N/4 + ~N/4 + ~N/4 + ~N/4 = ~N.
- Thus, total runtime is ~Nk, where k is the number of levels.
  - How many levels? Goes until we get to size 1.
  - $k = \log_2(N)$.
- Overall runtime is $\Theta(N \log N)$.



Exact count explanation is tedious.

- Omitted here. See textbook exercises.

Mergesort

We've seen this one before as well.

Mergesort:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.

Time complexity, analysis from asymptotics lecture: Θ(N log N runtime)

- Space complexity with aux array: Costs Θ(N) memory.

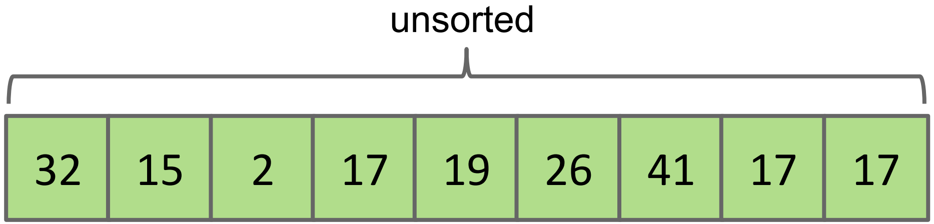
Also possible to do in-place merge sort, but algorithm is very complicated, and runtime performance suffers by a significant constant factor.

Top-Down Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half.
- Merge the two sorted halves to form the final result.

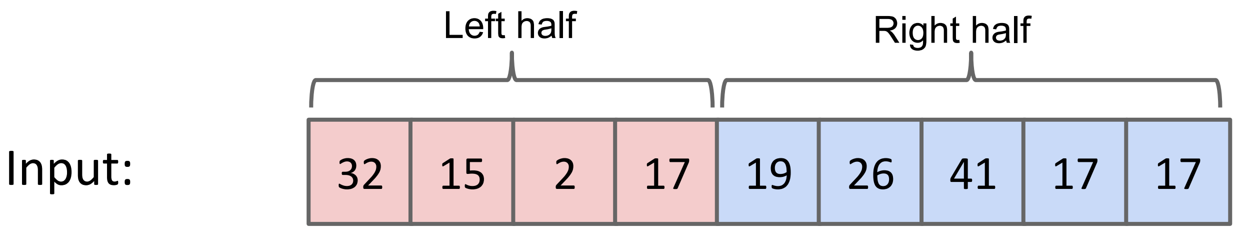
Input:



Top-Down Merge Sort

Top-Down merge sorting N items:

- **Split items into 2 roughly even pieces.**
- Mergesort each half.
- Merge the two sorted halves to form the final result.



Merge Sort

Top-Down merge sorting N items:

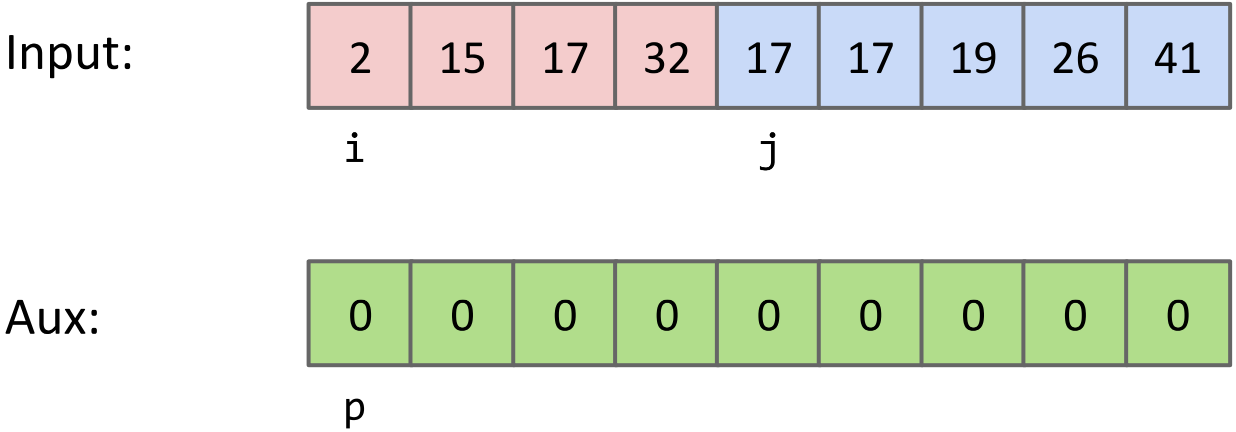
- Split items into 2 roughly even pieces.
- **Mergesort each half (steps not shown, this is a recursive algorithm!)**
- Merge the two sorted halves to form the final result.

	sorted					sorted			
Input:	2	15	17	32	17	17	19	26	41

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- **Merge the two sorted halves to form the final result.**
    - Compare input[i] < input[j].
    - Copy smaller item and increment p and i or j.



Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.

Input:	**2**	15	17	32	**17**	17	19	26	41
	i				j				

Aux:	0	0	0	0	0	0	0	0	0
	p								

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

Input:	2	15	17	32	17	17	19	26	41
		i			j				

Aux:	2	0	0	0	0	0	0	0	0
		p							

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.

Input:	2	**15**	17	32	**17**	17	19	26	41
		i			j				

Aux:	2	0	0	0	0	0	0	0	0
		p							

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - Compare input[i] < input[j] (if necessary).
    - **Copy smaller item and increment p and i or j.**

Input:

2	15	17	32	17	17	19	26	41
		i		j				

Aux:

2	15	0	0	0	0	0	0	0
		p						

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

Input:

2	15	**17**	32	**17**	17	19	26	41
		i		j				

Aux:

2	15	0	0	0	0	0	0	0
		p						

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

Input:

2	15	17	32	17	17	19	26	41
			i	j				

Aux:

2	15	17	0	0	0	0	0	0
			p					

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.

Input:	2	15	17	**32**	**17**	17	19	26	41
				i	j				
Aux:	2	15	17	0	0	0	0	0	0
				p					

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

Input:

2	15	17	32	17	17	19	26	41
			i		j			

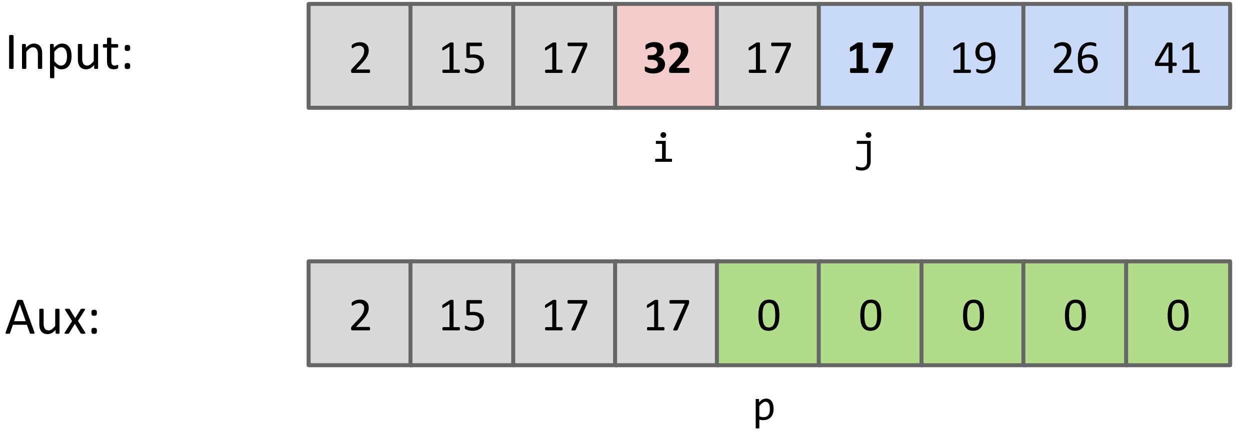
Aux:

2	15	17	17	0	0	0	0	0
				p				

Merge Sort

Top-Down merge sorting N items:

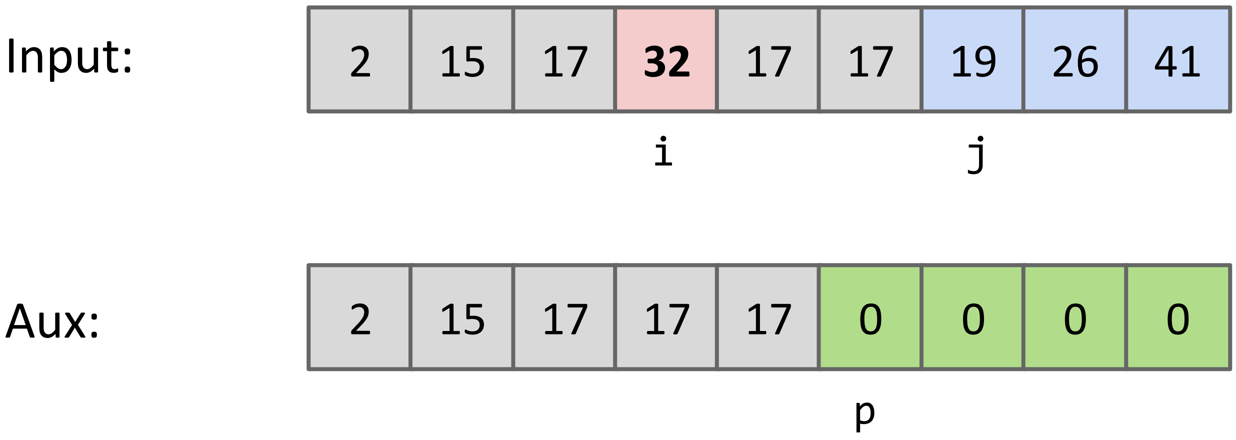
- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.



Merge Sort

Top-Down merge sorting N items:

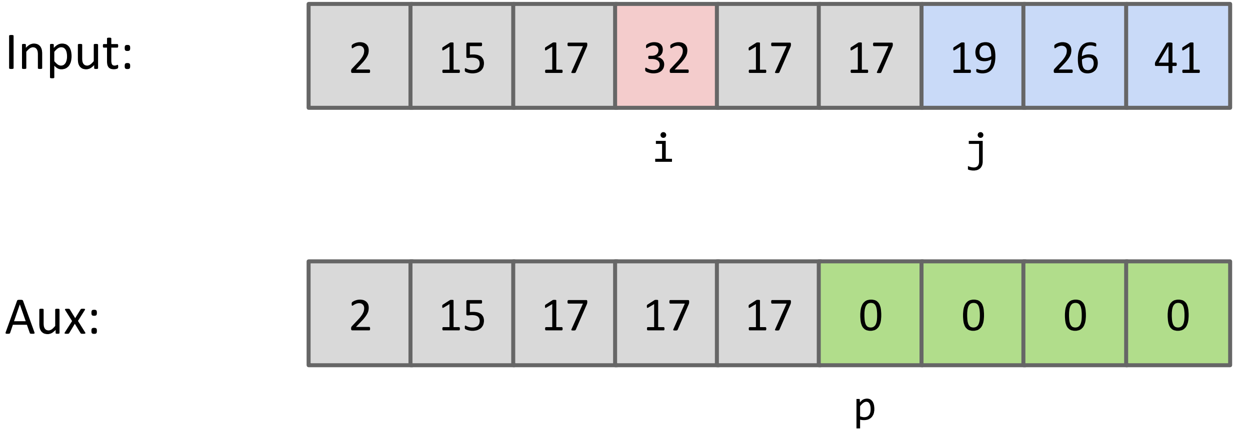
- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**



Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**



Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

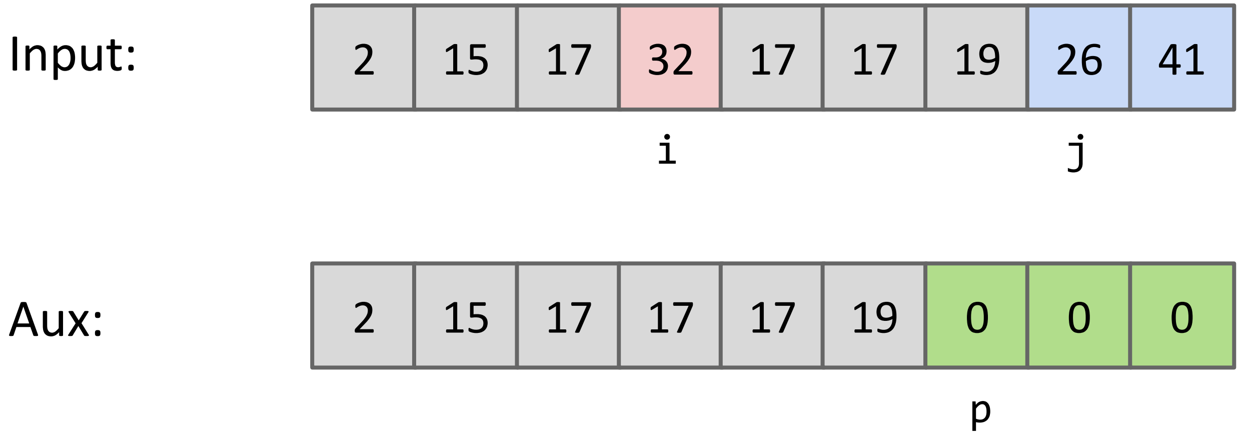
Input:	2	15	17	**32**	17	17	**19**	26	41
				i			j		

Aux:	2	15	17	17	17	0	0	0	0
						p			

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**



Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

Input:	2	15	17	**32**	17	17	19	**26**	41
				i				j	

Aux:	2	15	17	17	17	19	0	0	0
							p		

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - Compare input[i] < input[j] (if necessary).
    - **Copy smaller item and increment p and i or j.**

Input:	2	15	17	32	17	17	19	26	41
				i					j

Aux:	2	15	17	17	17	19	26	0	0
								p	

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
    - **Compare input[i] < input[j] (if necessary).**
    - Copy smaller item and increment p and i or j.

Input:	2	15	17	**32**	17	17	19	26	**41**
				i					j
Aux:	2	15	17	17	17	19	26	0	0
								p	

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - **Compare input[i] < input[j] (if necessary).**
  - Copy smaller item and increment p and i or j.

No comparison is made this time, since the left side has run out of items!

Input:

2	15	17	32	17	17	19	26	**41**
				i				j

Aux:

2	15	17	17	17	19	26	32	0
								p

Merge Sort

Top-Down merge sorting N items:

- Split items into 2 roughly even pieces.
- Mergesort each half (steps not shown, this is a recursive algorithm!)
- Merge the two sorted halves to form the final result.
  - Compare input[i] < input[j] (if necessary).
  - **Copy smaller item and increment p and i or j.**

Input:

2	15	17	32	17	17	19	26	41

i j

Aux:

2	15	17	17	17	19	26	32	41

p

Sorts So Far

	Best Case Runtime	Worst Case Runtime	Space	Demo	Notes
Selection Sort	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	Link	
Heapsort (in place)	$\Theta(N)$*	$\Theta(N \log N)$	$\Theta(1)$**	Link	Bad cache (61C) performance.
Mergesort	$\Theta(N \log N)$	$\Theta(N \log N)$	$\Theta(N)$	Link	Faster than heap sort.

*: An array of all duplicates yields linear runtime for heapsort.
**: Assumes heap operations implemented iteratively, not recursively.