Lecture 29 (Sorting 1)

Basic Sorts

CS61B, Spring 2024 @ UC Berkeley

Slides credit: Josh Hug

Goal: Sorting

Lecture 29, CS61B, Spring 2024

61B Phase 3

We are now in Phase 3 of the course:

- Algorithms and Software Engineering.

Lectures in this phase:

- Algorithms.
- 3 software engineering lectures (we already did #1).

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 29, CS61B, Spring 2024

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”).

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

```
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).

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.

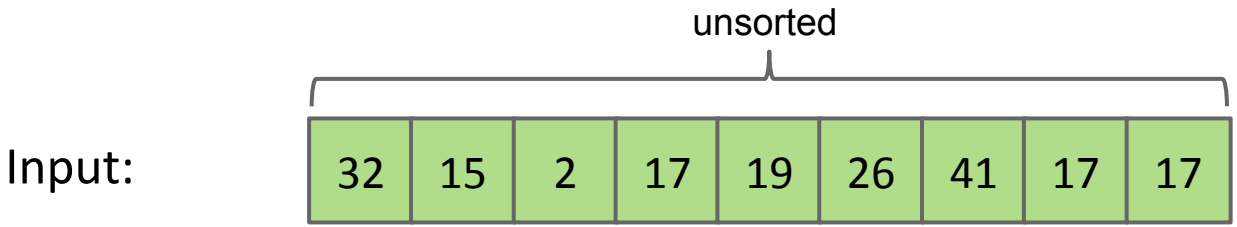
Selection Sort

Lecture 29, CS61B, Spring 2024

Selection Sort

Selection sorting N items:

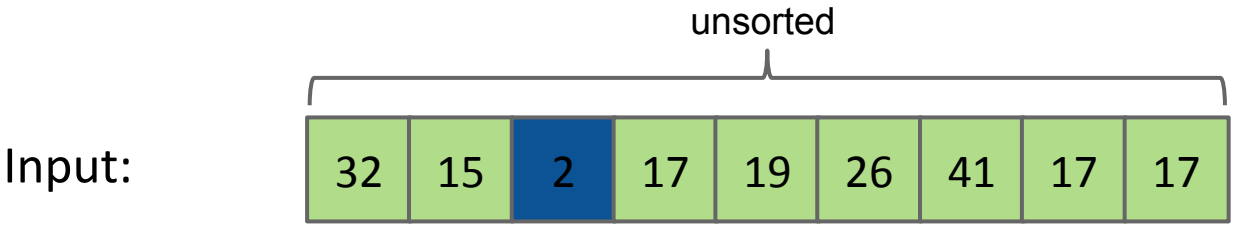
- Find the smallest item in the unsorted portion of the array.
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

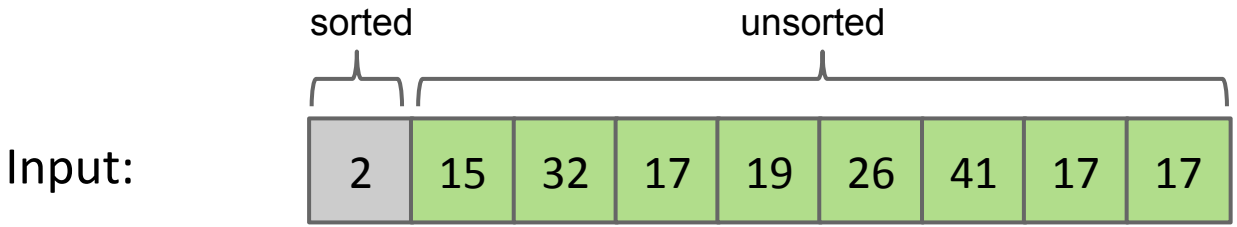
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

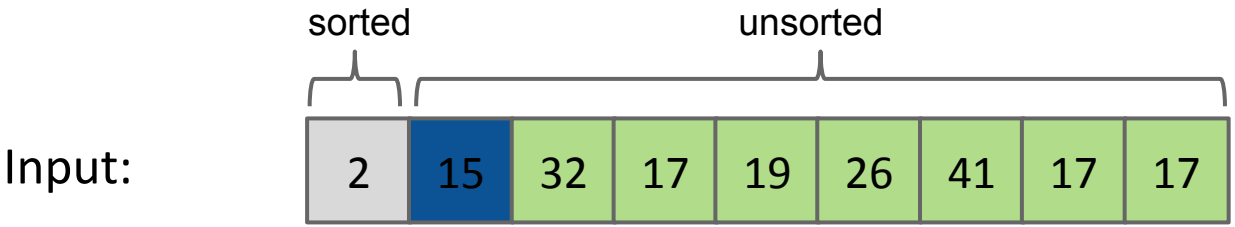
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

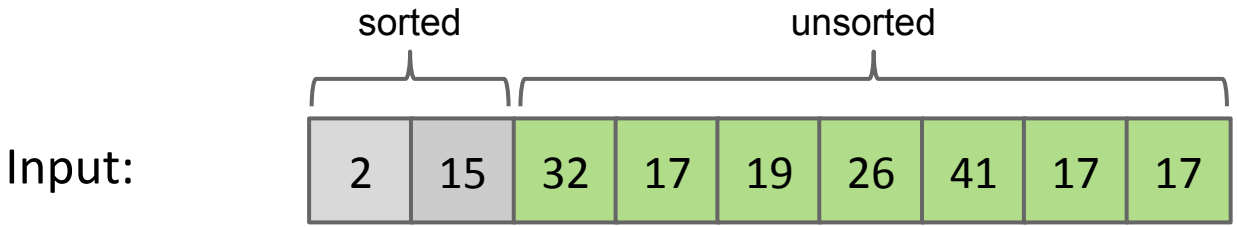
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

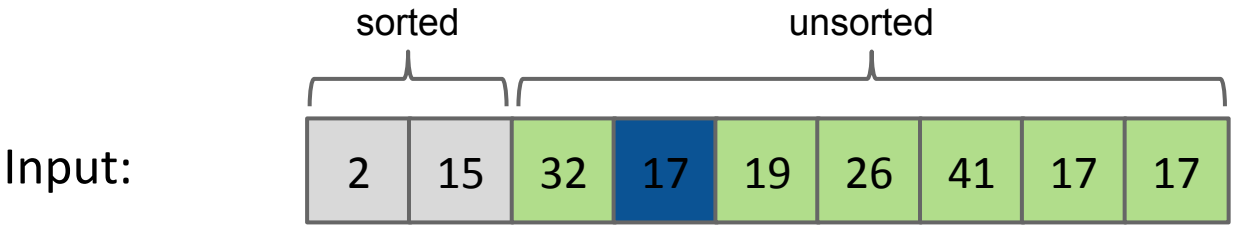
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

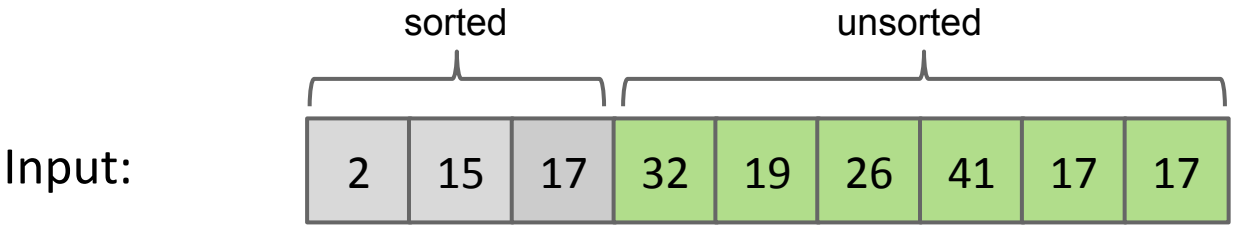
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

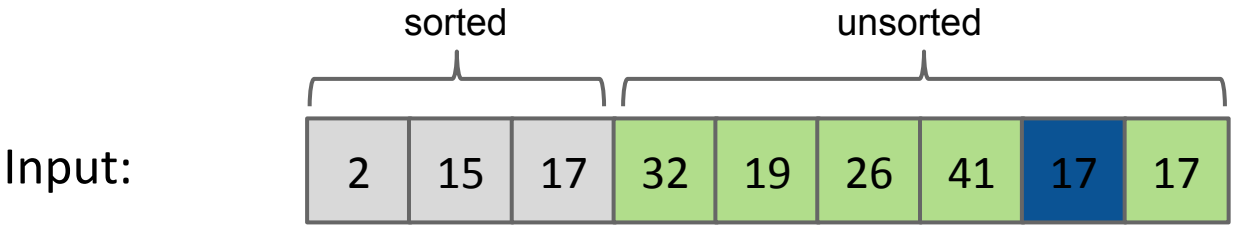
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

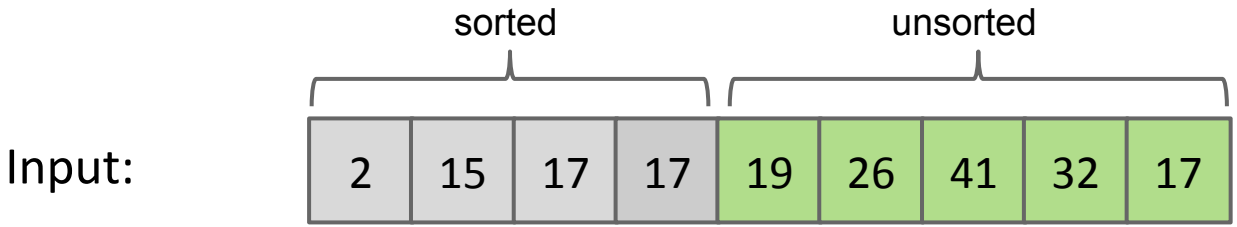
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

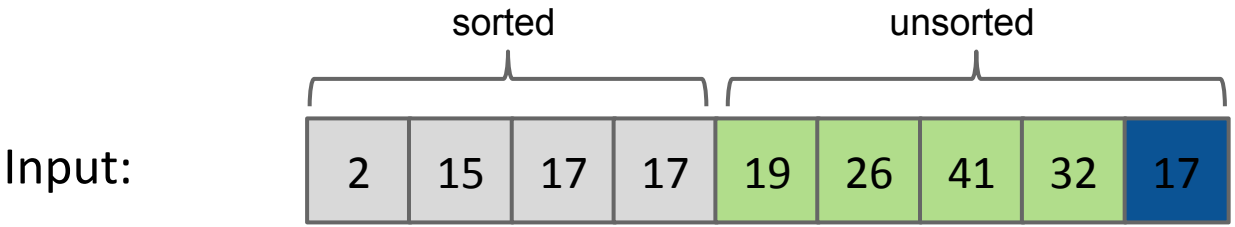
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

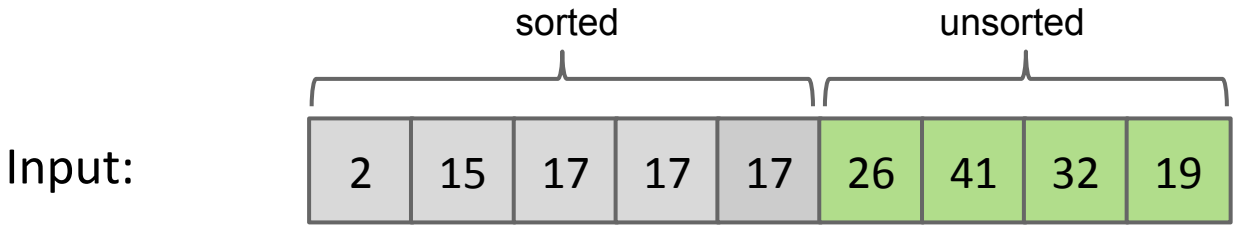
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

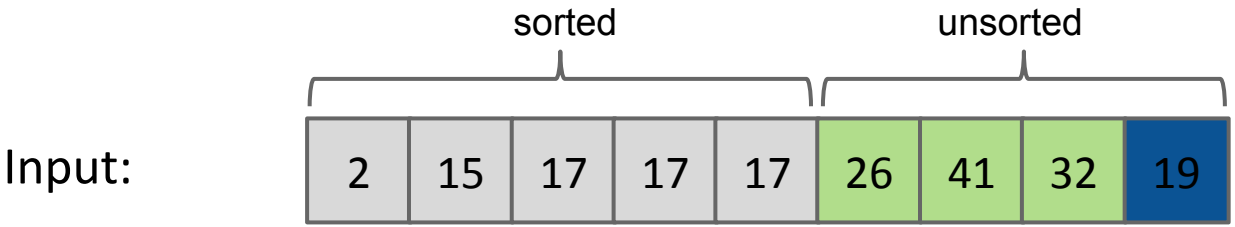
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

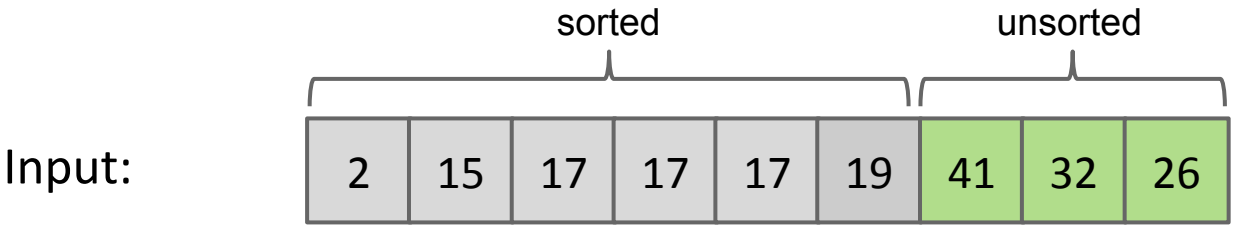
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

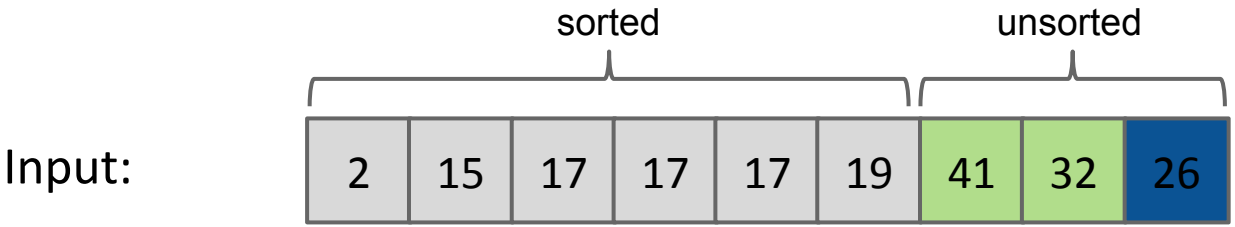
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

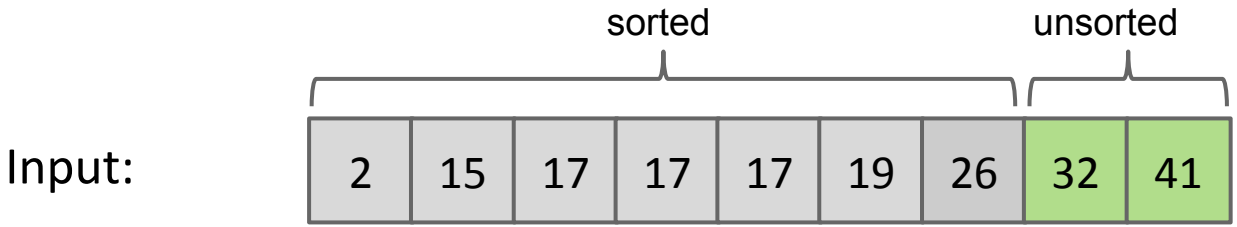
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

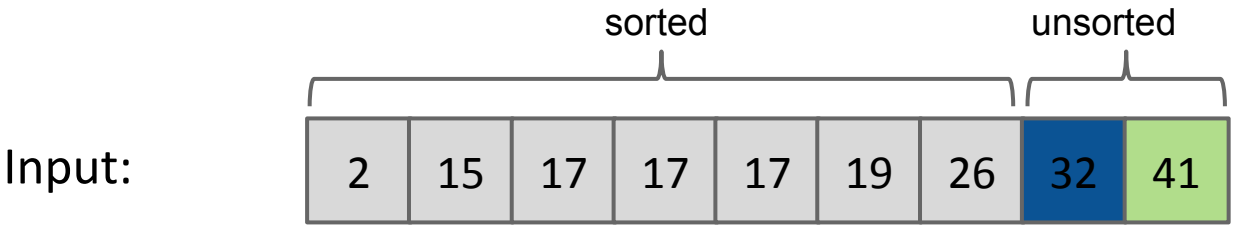
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

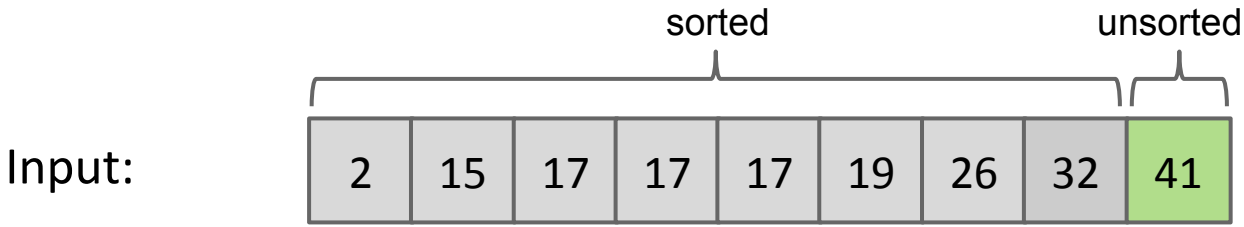
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

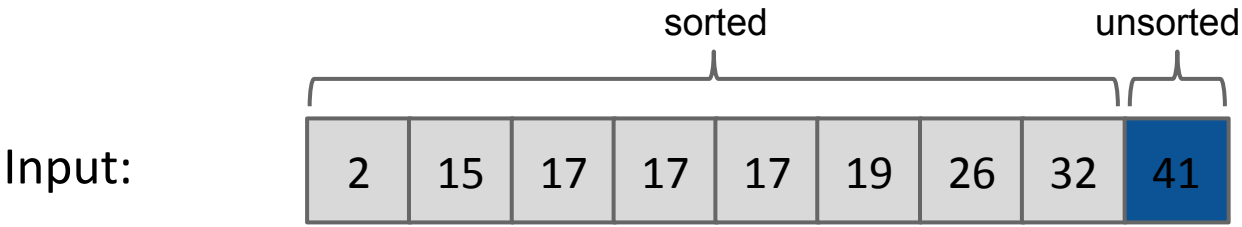
- Find the smallest item in the unsorted portion of the array.
- **Move it to the end of the sorted portion of the array.**
- Selection sort the remaining unsorted items.



Selection Sort

Selection sorting N items:

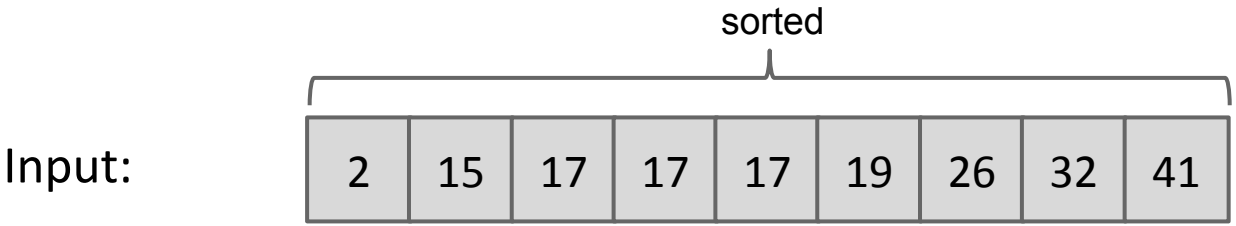
- **Find the smallest item in the unsorted portion of the array.**
- Move it to the end of the sorted portion of the array.
- Selection sort the remaining unsorted items.



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

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).

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

Naive Heapsort

Lecture 29, CS61B, Spring 2024

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.

Naive heapsorting N items:

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

- 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:

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:

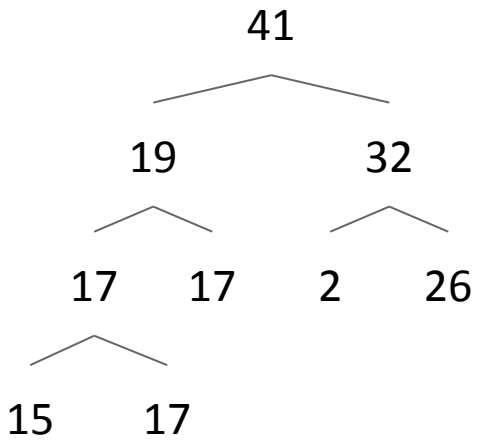
32	15	2	17	19	26	41	17	17

(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:

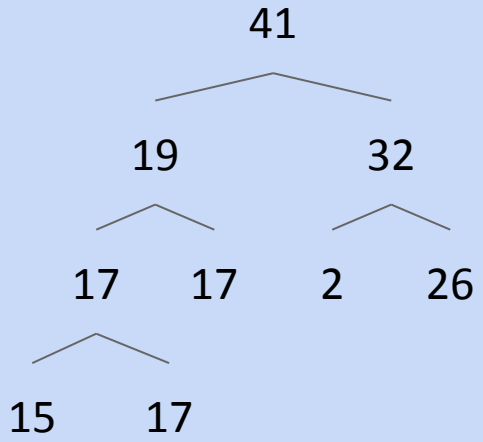
32	15	2	17	19	26	41	17	17

(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

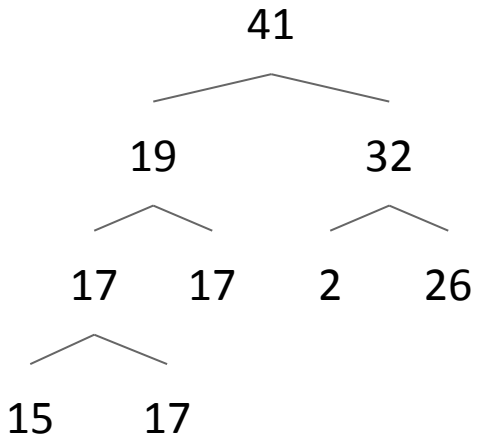
Heap sorting N items:

- 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

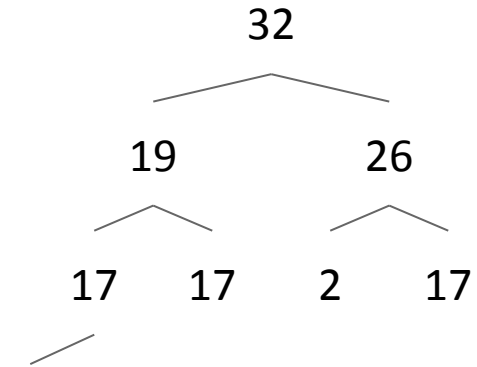
Heap sorting N items:

- 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

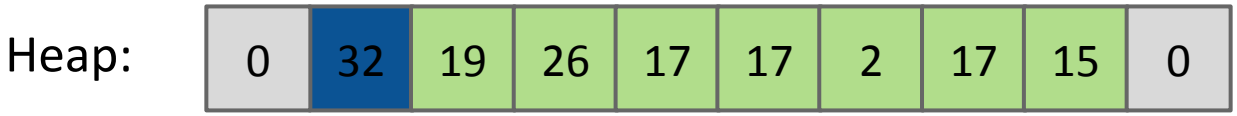
Output:

0	0	0	0	0	0	0	0	41

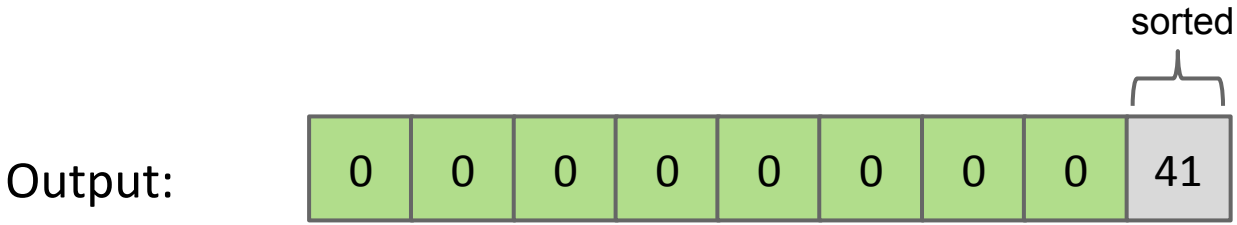
Naive Heap Sort: Phase 2: Heap Deletion

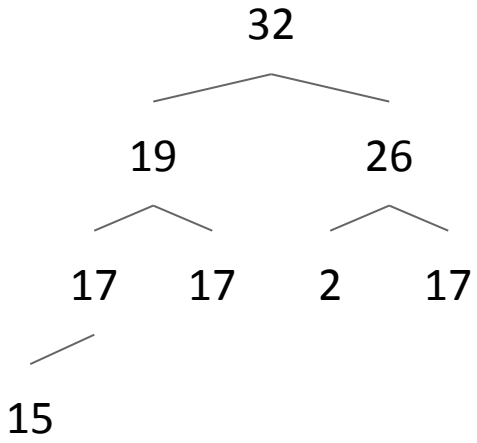
Heap sorting N items:

- 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.



Size: 8





Naive Heap Sort: Phase 2: Heap Deletion

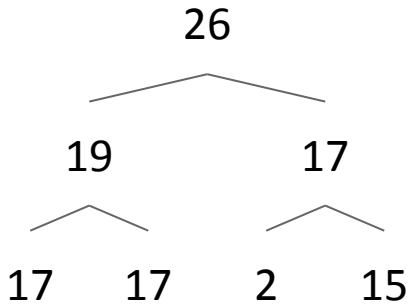
Heap sorting N items:

- 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



Output:

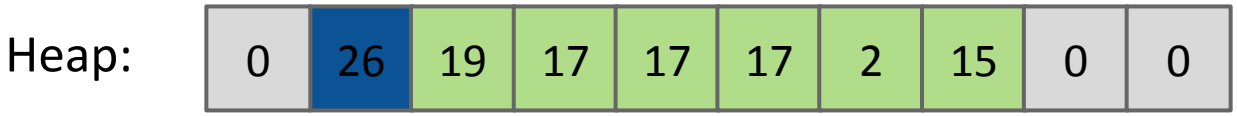
0	0	0	0	0	0	0	32	41

sorted

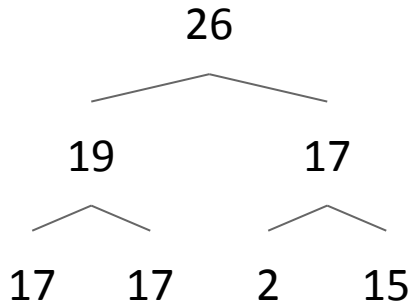
Naive Heap Sort: Phase 2: Heap Deletion

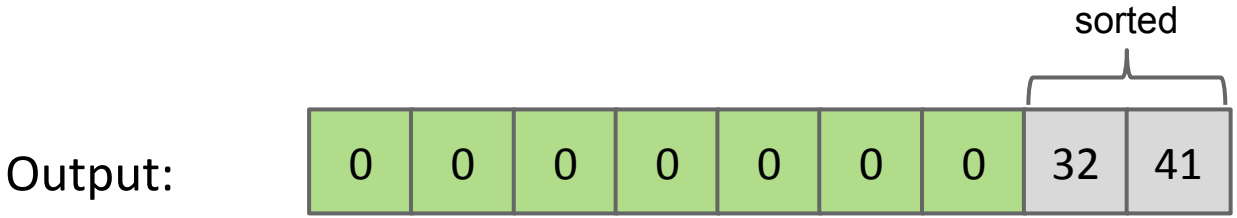
Heap sorting N items:

- 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.



Size: 7





Naive Heap Sort: Phase 2: Heap Deletion

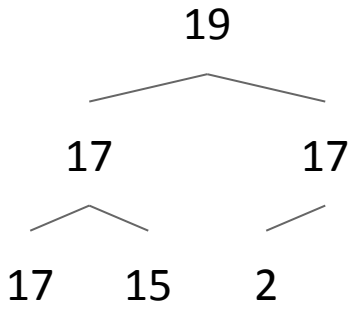
Heap sorting N items:

- 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	19	17	17	17	15	2	0	0	0

Size: 6



Output:

sorted

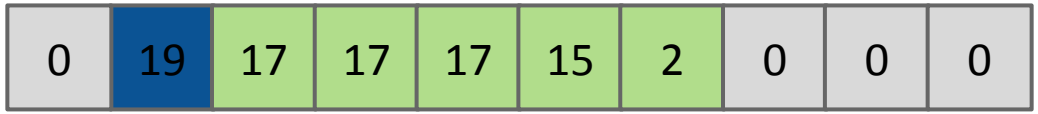
0	0	0	0	0	0	26	32	41

Naive Heap Sort: Phase 2: Heap Deletion

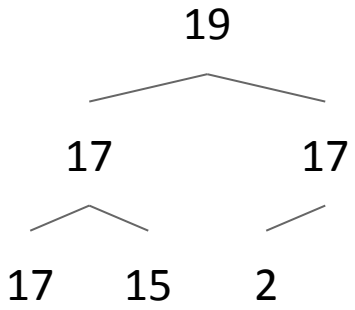
Heap sorting N items:

- 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.

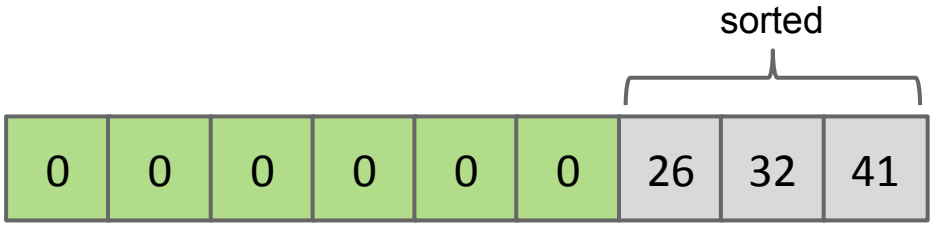
Heap:



Size: 6



Output:



Naive Heap Sort: Phase 2: Heap Deletion

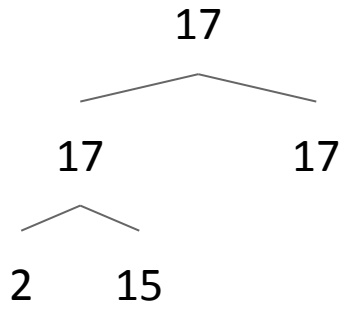
Heap sorting N items:

- 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



Output:

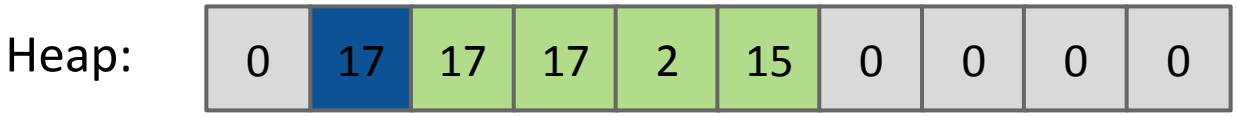
0	0	0	0	0	19	26	32	41

sorted

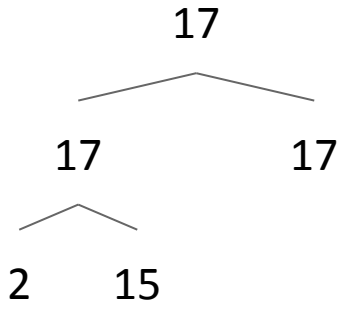
Naive Heap Sort: Phase 2: Heap Deletion

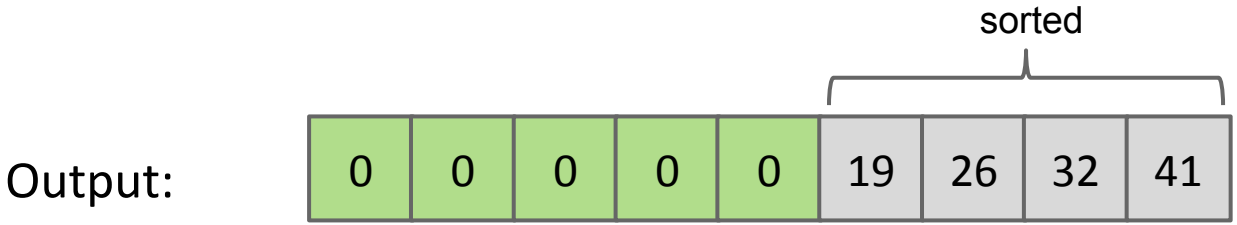
Heap sorting N items:

- 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.



Size: 5

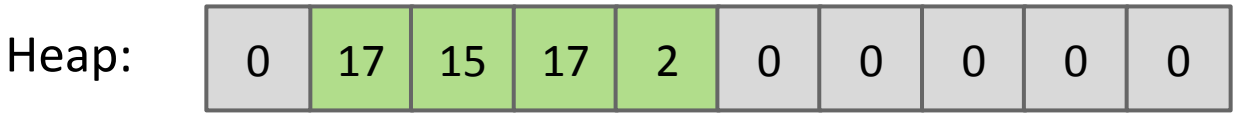




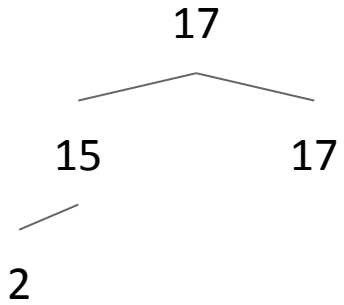
Naive Heap Sort: Phase 2: Heap Deletion

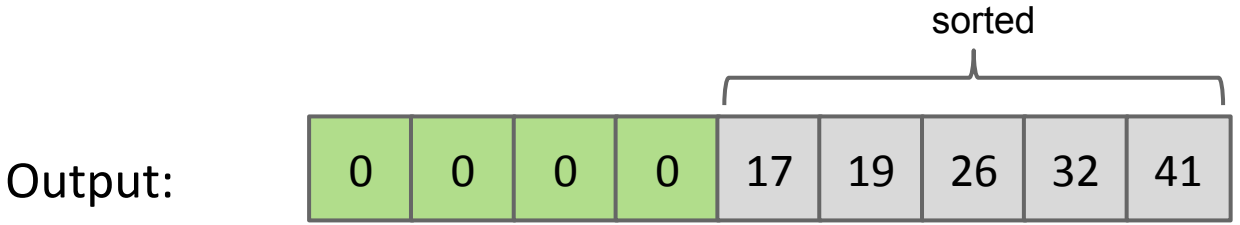
Heap sorting N items:

- 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.**



Size: 4





Naive Heap Sort: Phase 2: Heap Deletion

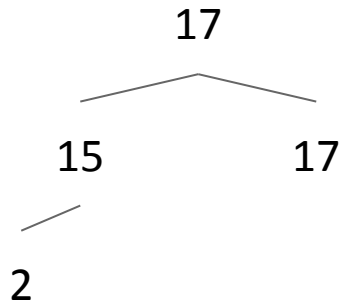
Heap sorting N items:

- 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	15	2	0	0	0	0	0

Size: 4



Output:

0	0	0	0	17	19	26	32	41

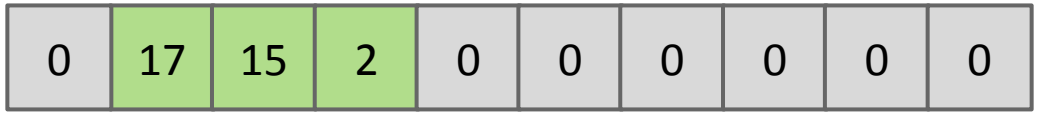
sorted

Naive Heap Sort: Phase 2: Heap Deletion

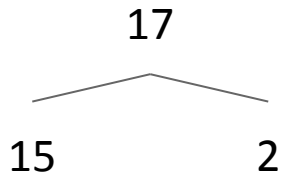
Heap sorting N items:

- 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.**

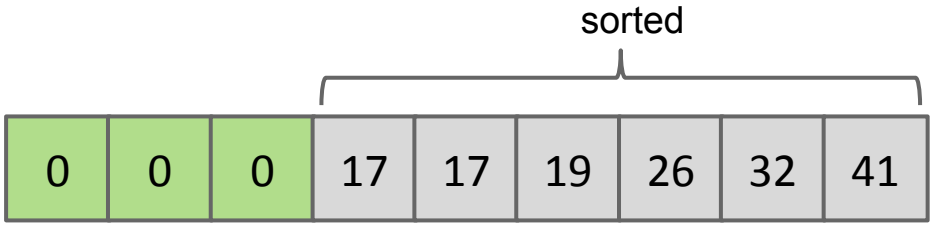
Heap:



Size: 3



Output:

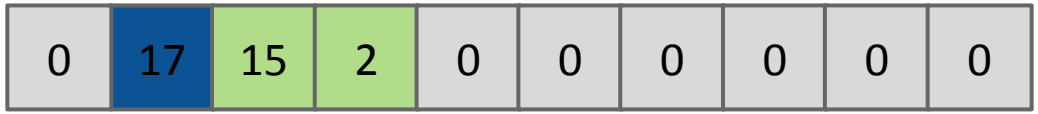


Naive Heap Sort: Phase 2: Heap Deletion

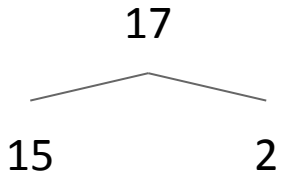
Heap sorting N items:

- 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.

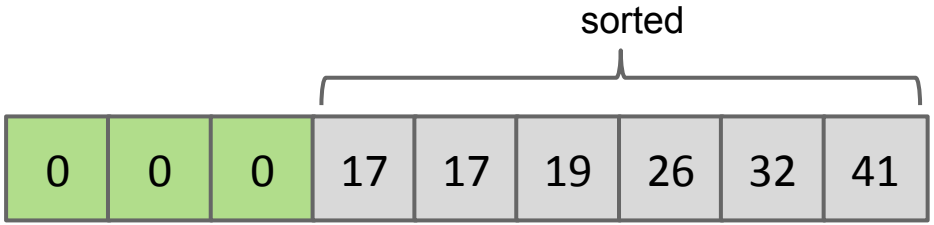
Heap:



Size: 3



Output:



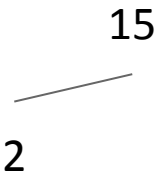
Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

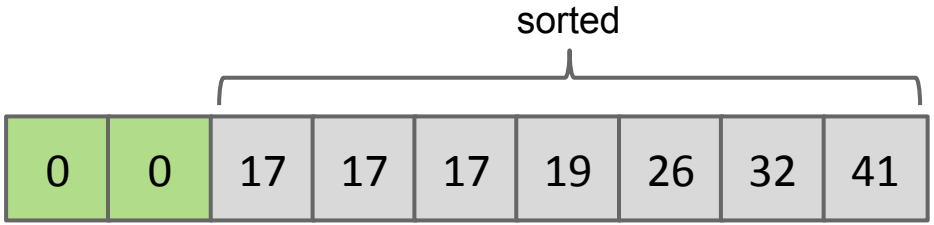
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - **Delete largest item from the max heap.**
  - **Put deleted item at the end of the unused part of the output array.**

Heap:

Size: 2



Output:



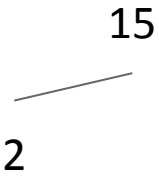
Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

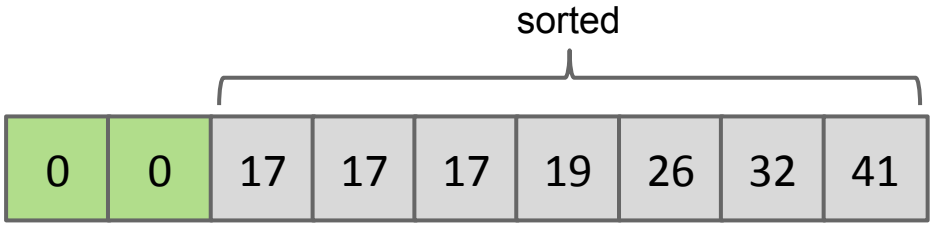
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.

Heap:

Size: 2



Output:



Naive Heap Sort: Phase 2: Heap Deletion

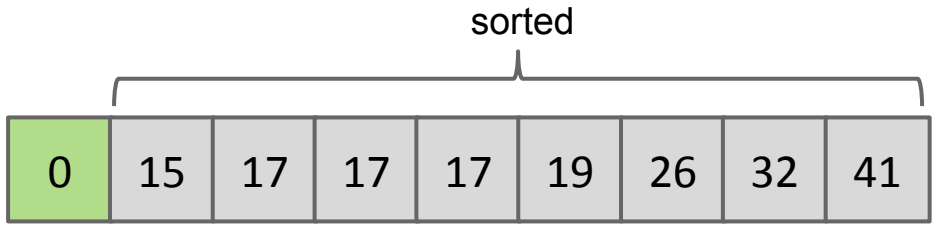
Heap sorting N items:

- 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:

Size: 1



Naive Heap Sort: Phase 2: Heap Deletion

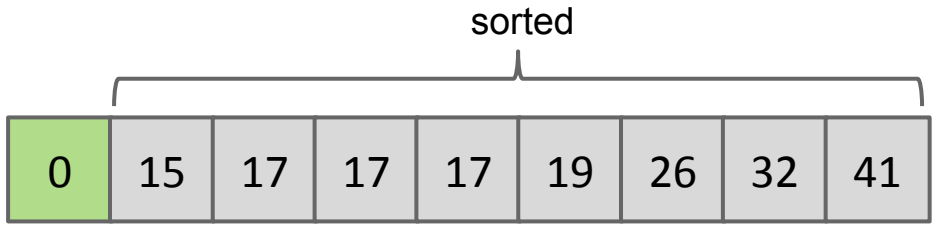
Heap sorting N items:

- 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:

Size: 1



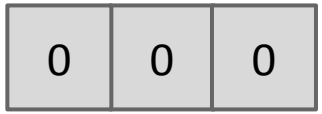
Naive Heap Sort: Phase 2: Heap Deletion

Heap sorting N items:

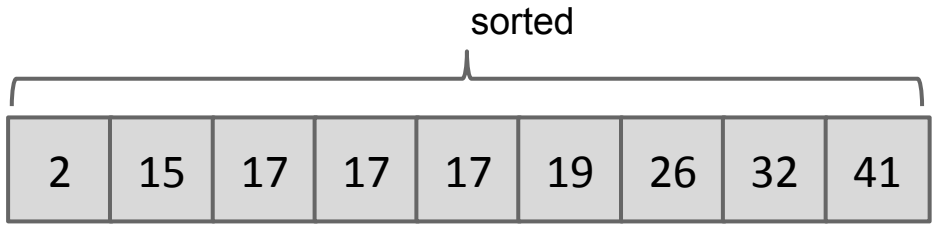
- Insert all items into a max heap, and discard input array. Create output array.
- Repeat N times:
  - Delete largest item from the max heap.
  - Put deleted item at the end of the unused part of the output array.

2

Heap:



Size: 0



Naive Heapsort Runtime: yellkey.com/TODO

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 29, CS61B, Spring 2024

Goal: Sorting

The Sorting Problem

Selection Sort

Heapsort

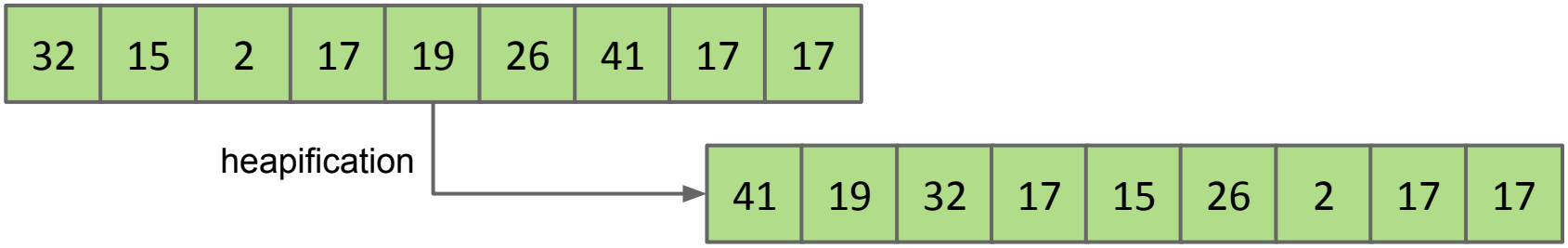
- Naive Heapsort
- **In-Place Heapsort**
- Heapsort Runtime

Mergesort

In-place Heapsort

Alternate approach, treat input array as a heap!

- Rather than inserting into a new array of length N + 1, use a process known as "bottom-up heapification" to convert the array into a heap.
  - To bottom-up heapify, just sink nodes in reverse level order.
- Avoids need for extra copy of all data.
- Once heapified, algorithm is almost the same as naive heap sort.



For this algorithm we don't leave spot 0 blank.

In-place heap sort: Demo

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:

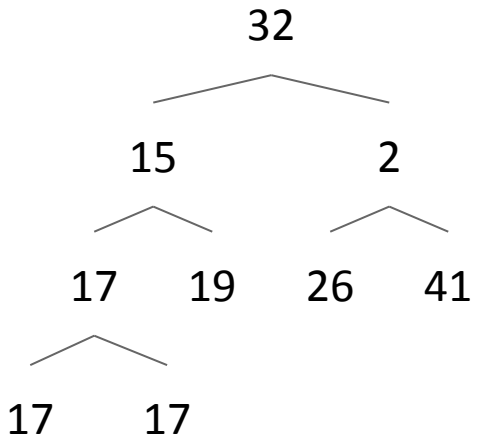
In-place Heap Sort: Phase 1: Heapification

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:

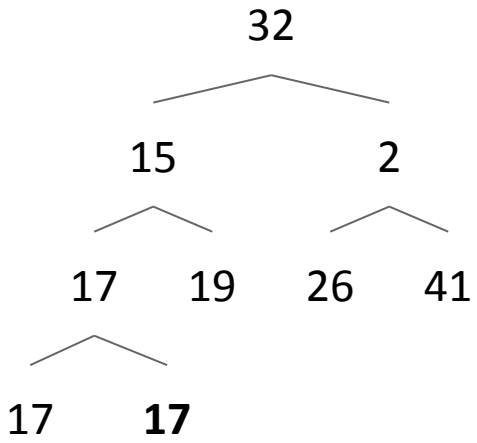


In-place Heap Sort: Phase 1: Heapification

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:

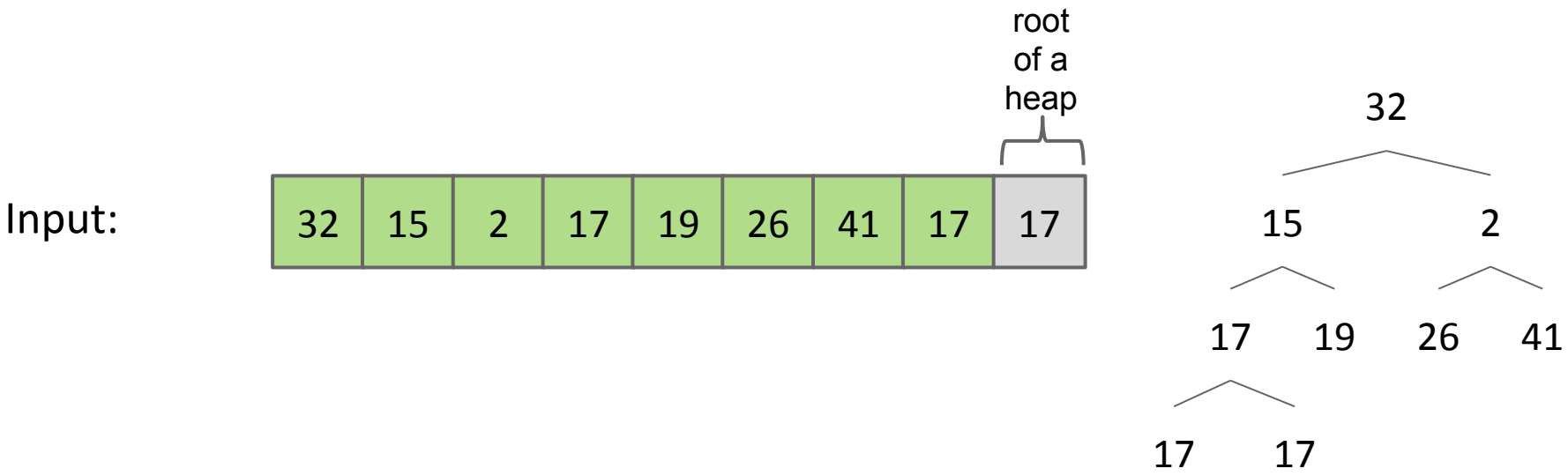


Sinking 17 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

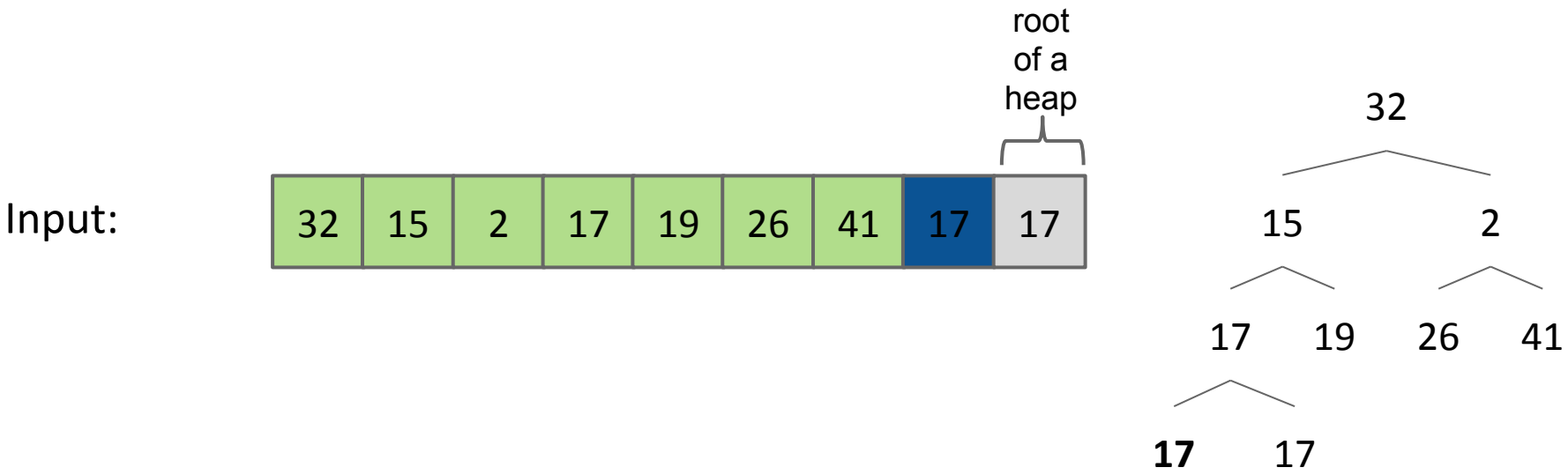
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**



In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

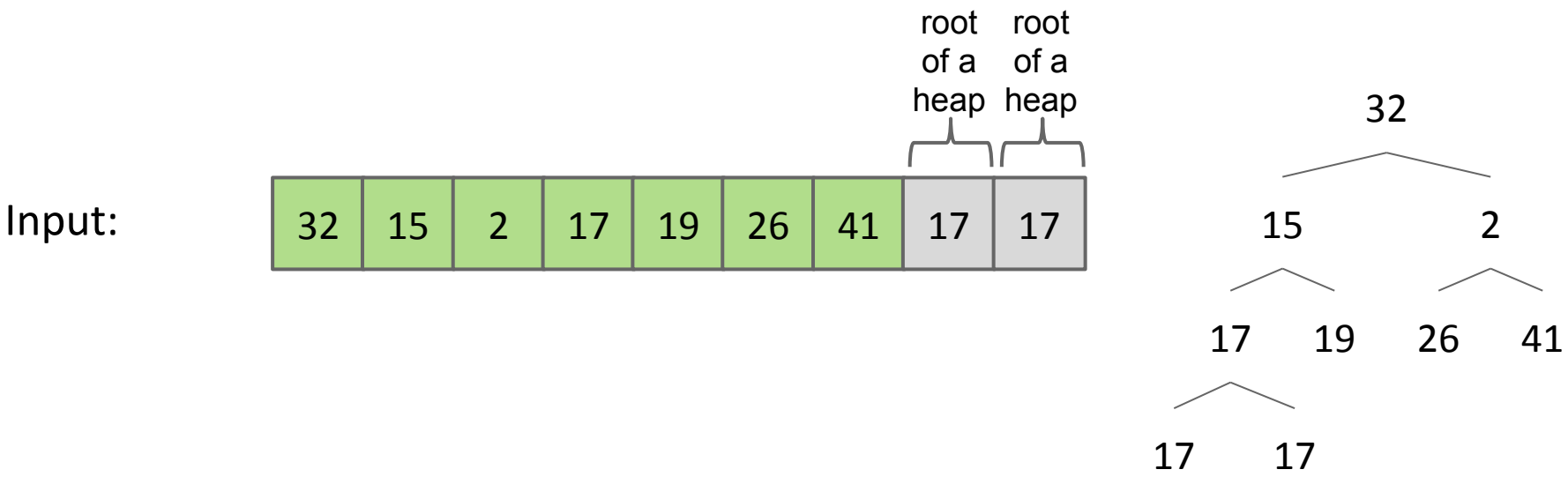
- Bottom-up heapify input array:
  - **Sink nodes in reverse level order: sink(k)**
  - After sinking, guaranteed that tree rooted at position k is a heap.



In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

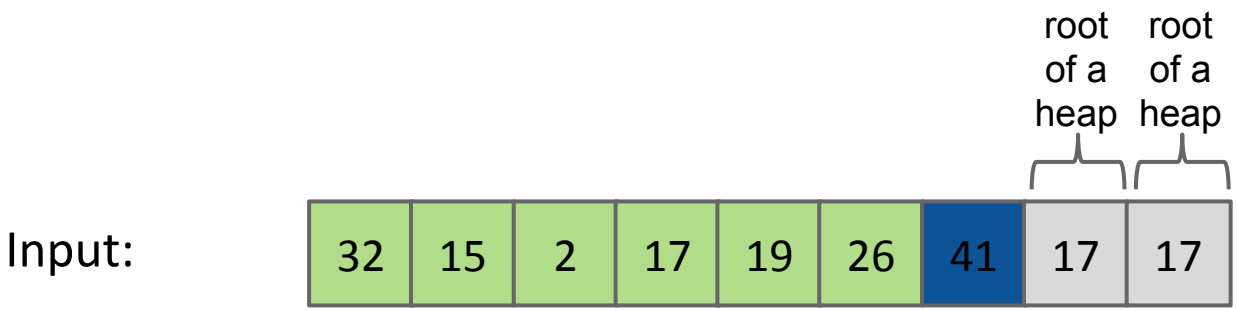
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

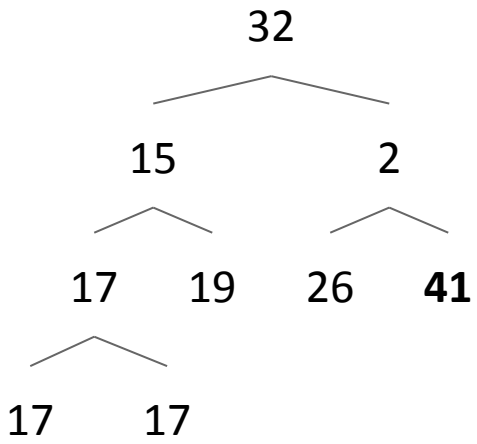


In-place Heap Sort: Phase 1: Heapification

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.



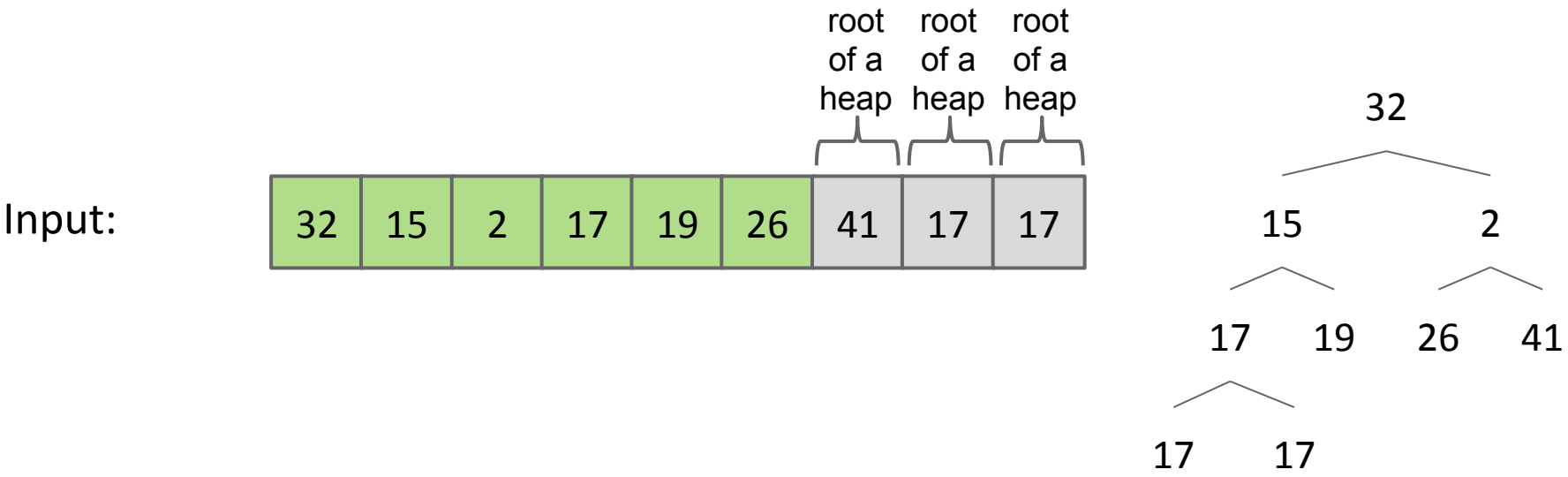


Sinking 41 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

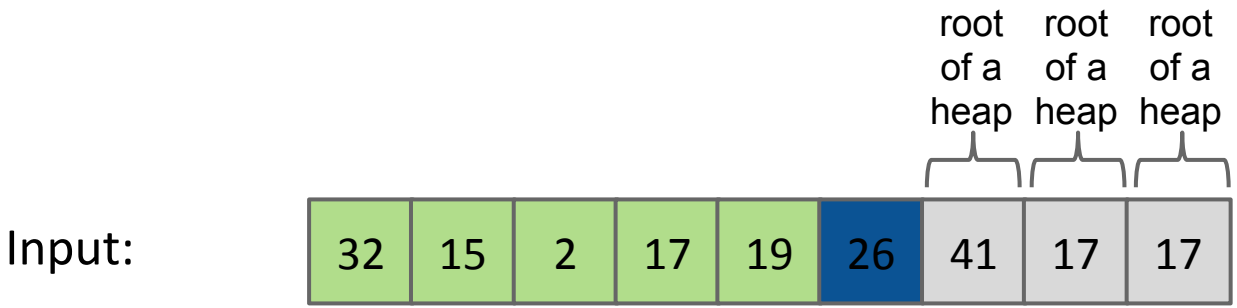
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

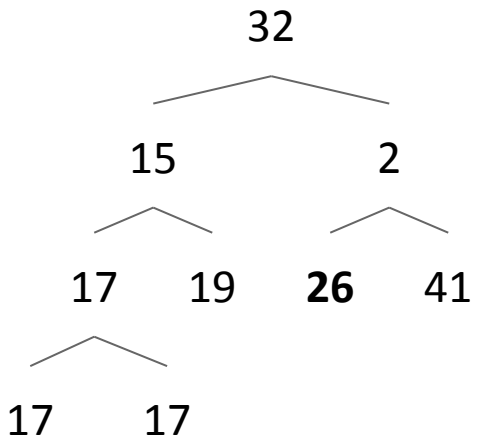


In-place Heap Sort: Phase 1: Heapification

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.



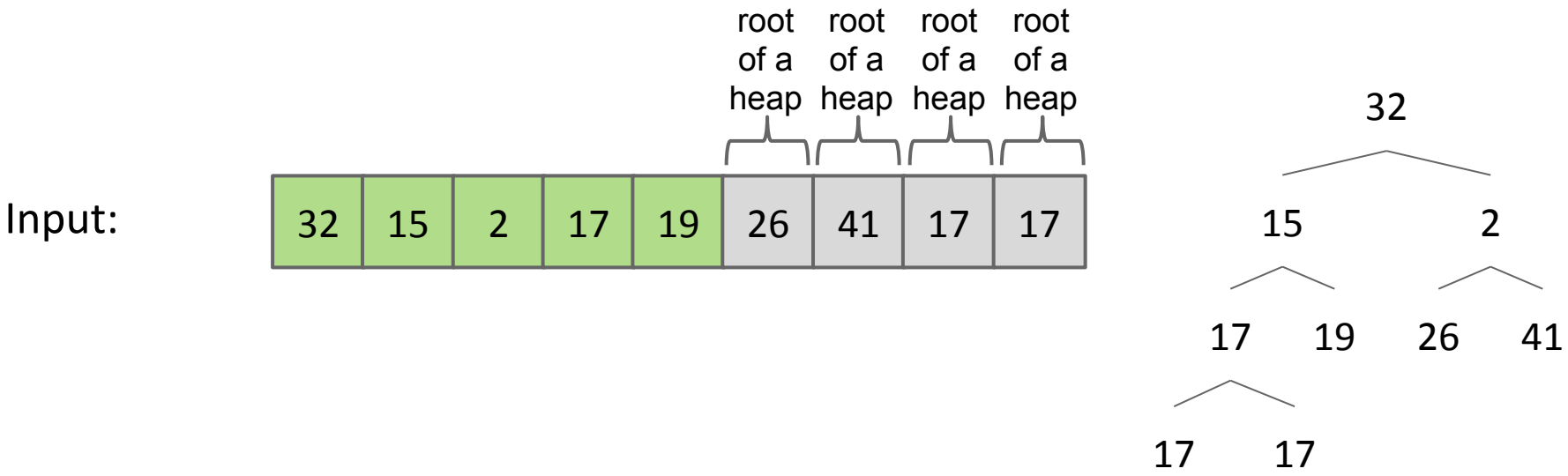


Sinking 26 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

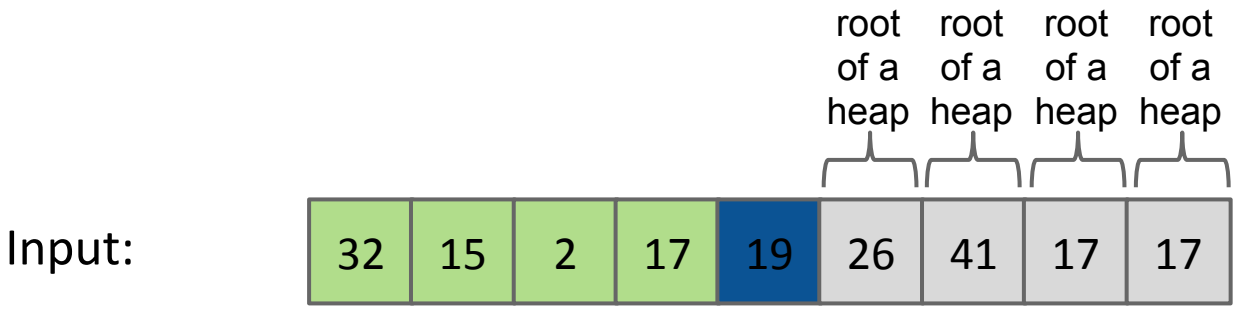
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

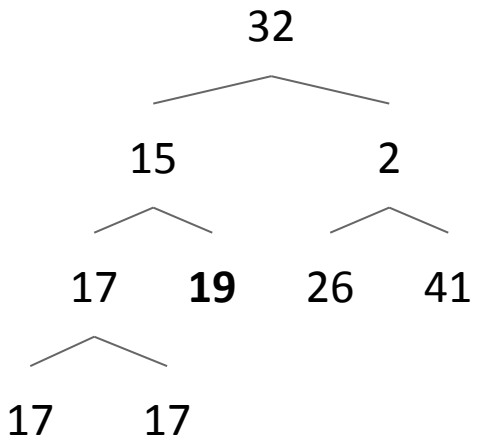


In-place Heap Sort: Phase 1: Heapification

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.



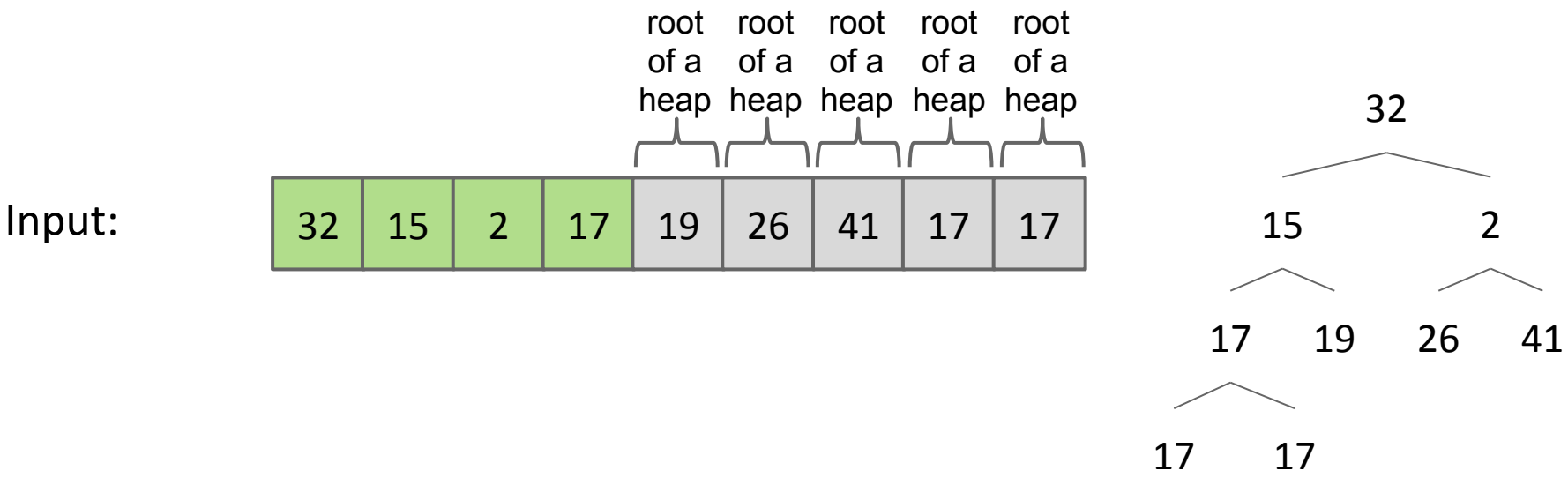


Sinking 19 has no effect.

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

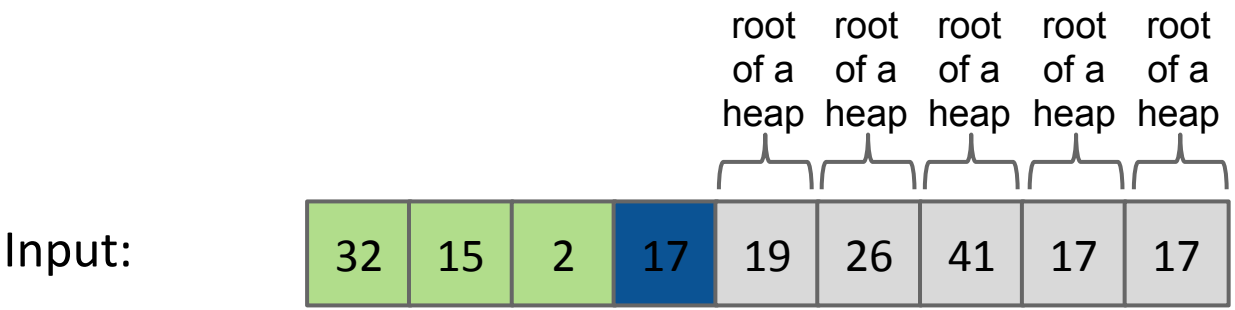
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

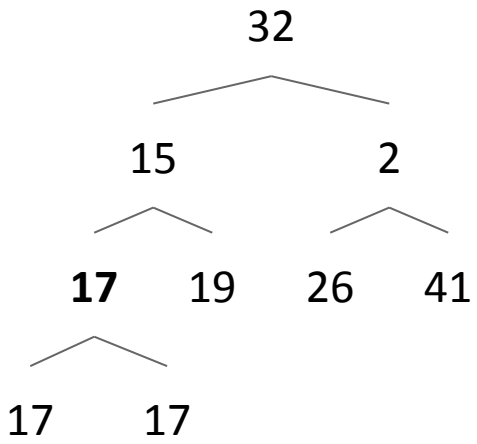


In-place Heap Sort: Phase 1: Heapification

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.



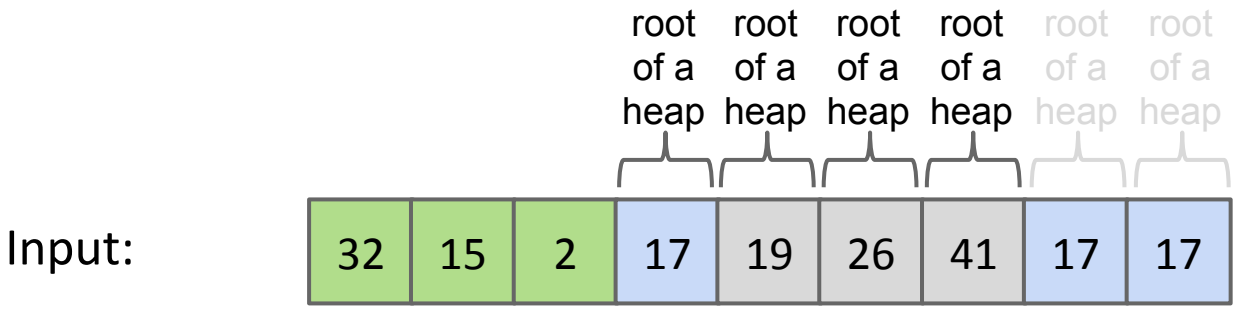


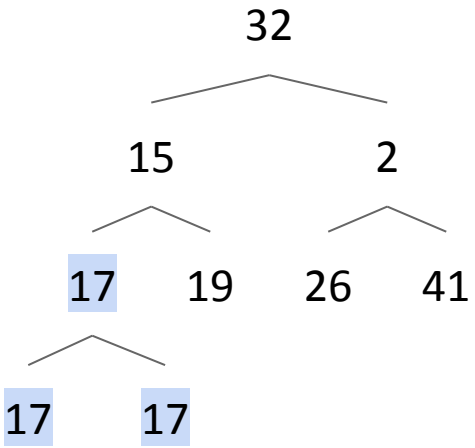
Sinking 17 has no effect.

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.**



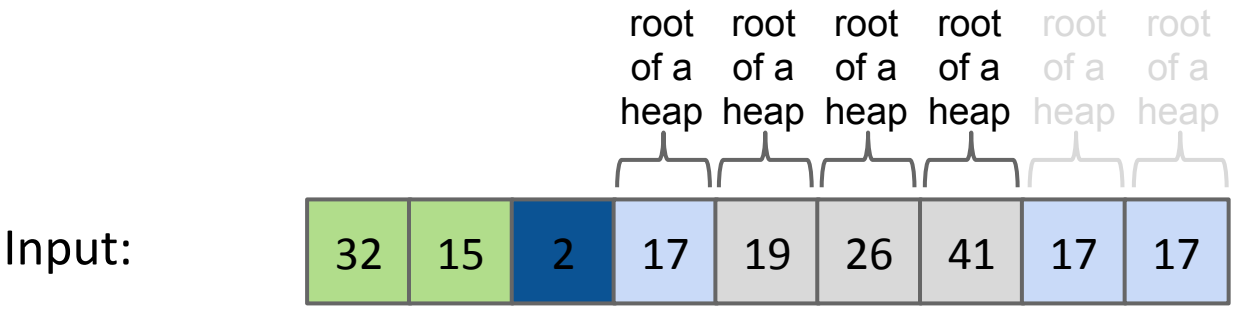


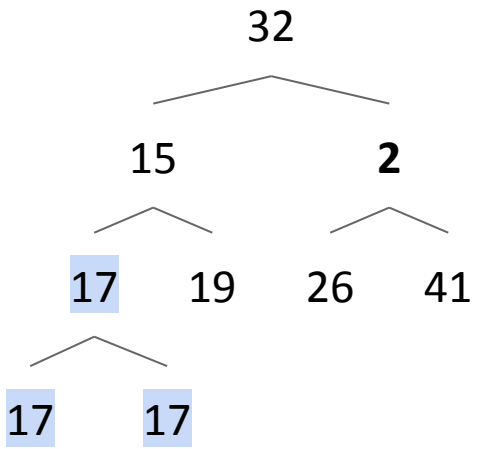
The blue coloring is to make it clear that the three 17s are all part of the same heap. I've also grayed out the "root of a heap" statement about the last two 17s since this is redundant information (all subheap nodes are also roots of that subheap).

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.



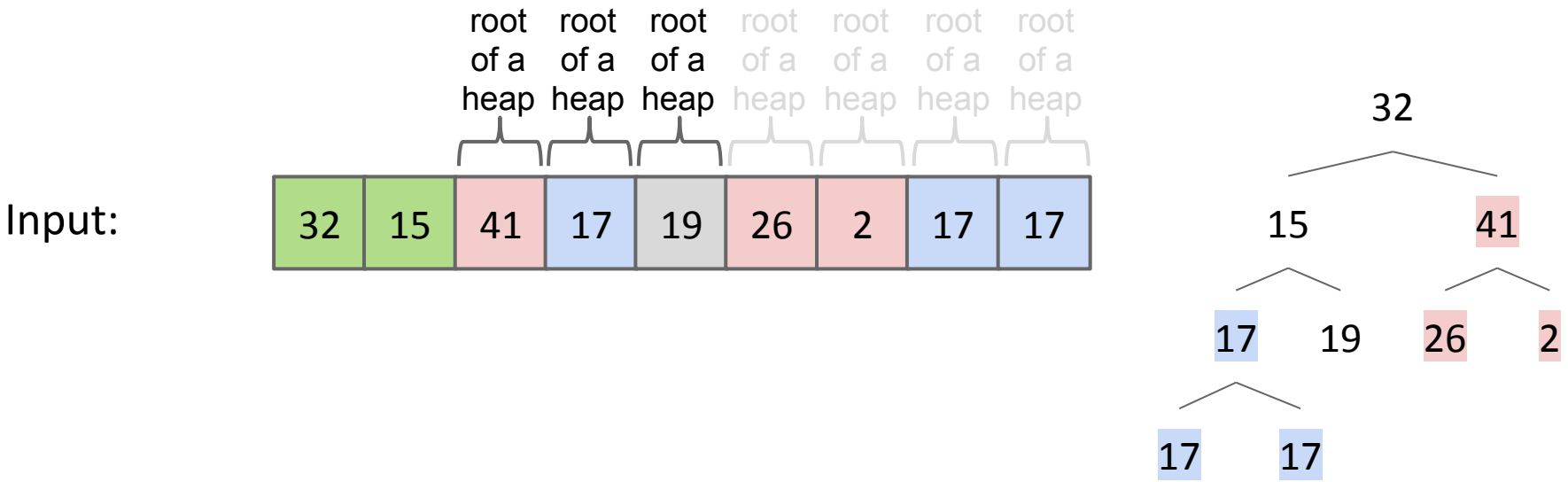


Sinking 2 does something!

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

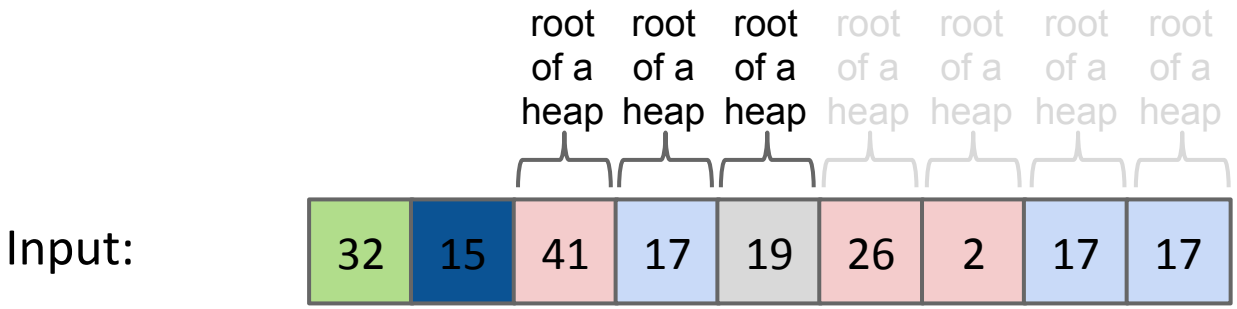
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

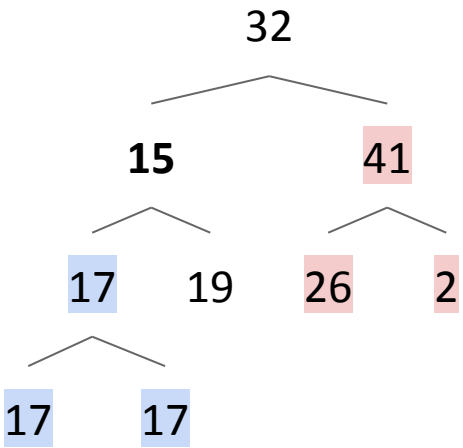


In-place Heap Sort: Phase 1: Heapification

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.



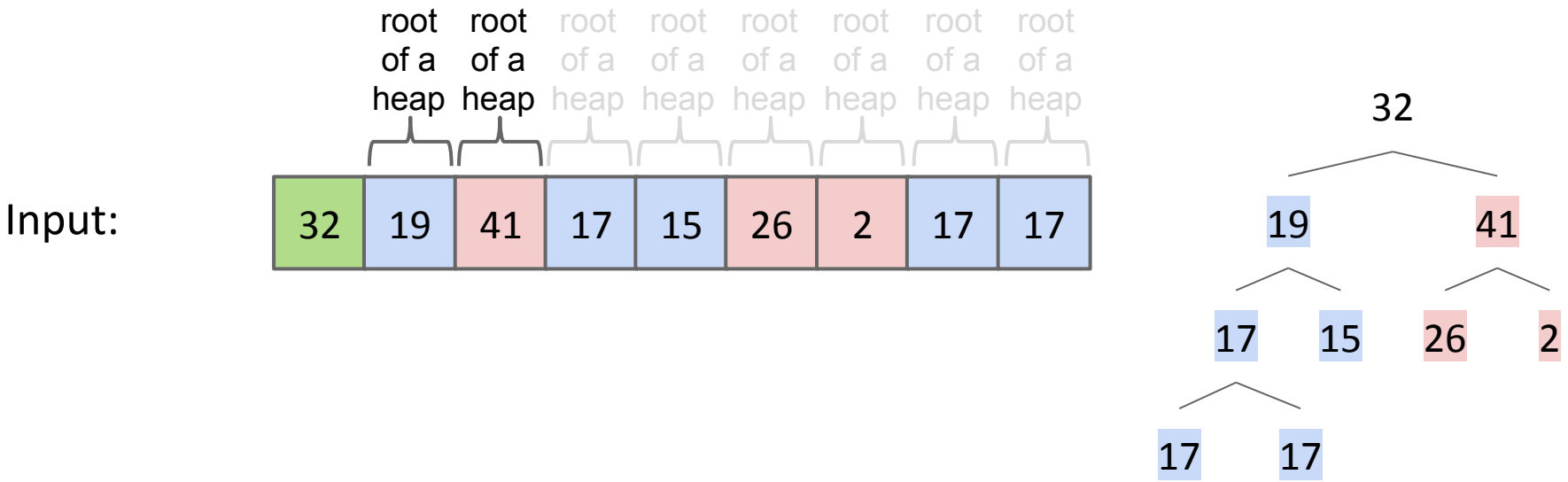


Sinking 15 does something!

In-place Heap Sort: Phase 1: Heapification

Heap sorting N items:

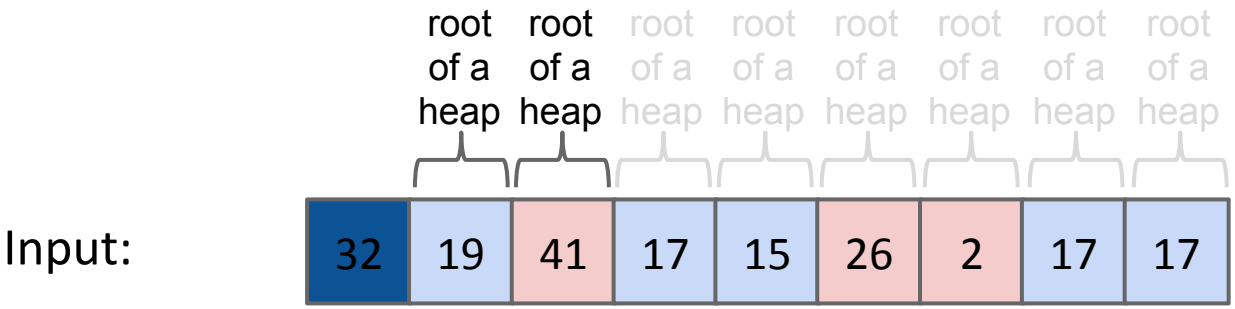
- Bottom-up heapify input array:
  - Sink nodes in reverse level order: sink(k)
  - **After sinking, guaranteed that tree rooted at position k is a heap.**

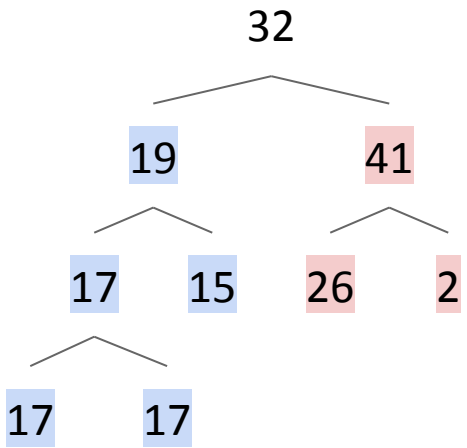


In-place Heap Sort: Phase 1: Heapification

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.



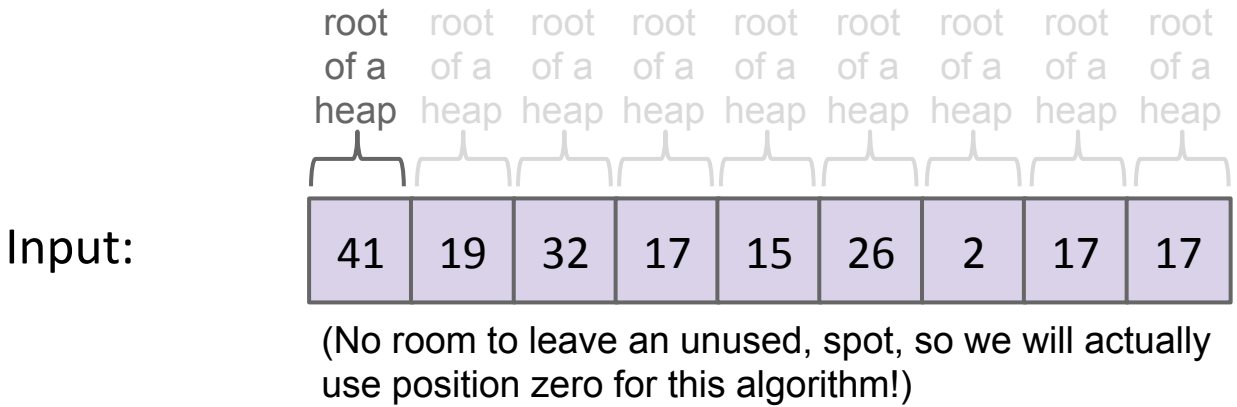


Sinking 32 does something!

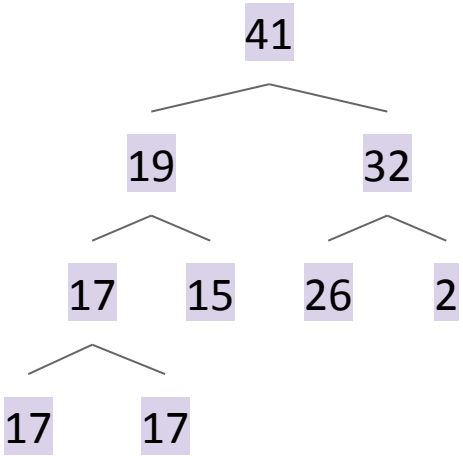
In-place Heap Sort: Phase 1: Heapification

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.**



(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.

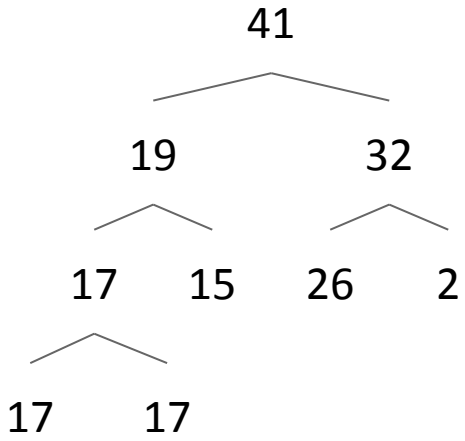
In-place Heap Sort

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:

Size: 9



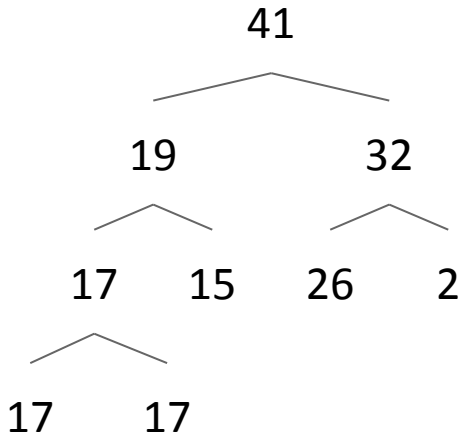
In-place Heap Sort

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:

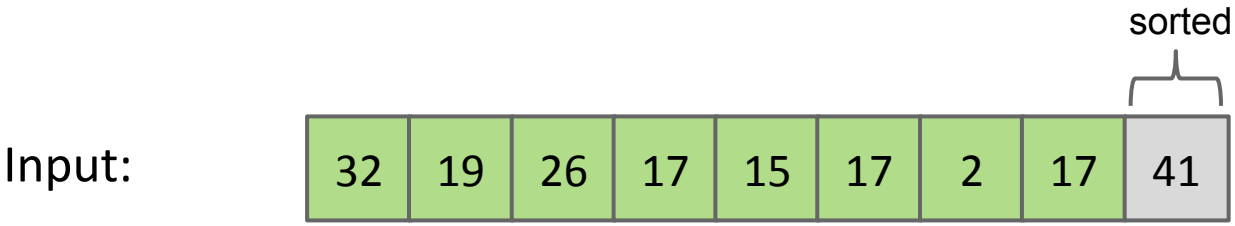
Size: 9

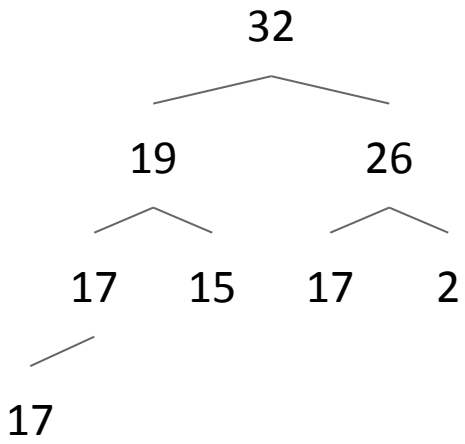


In-place Heap Sort

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.**

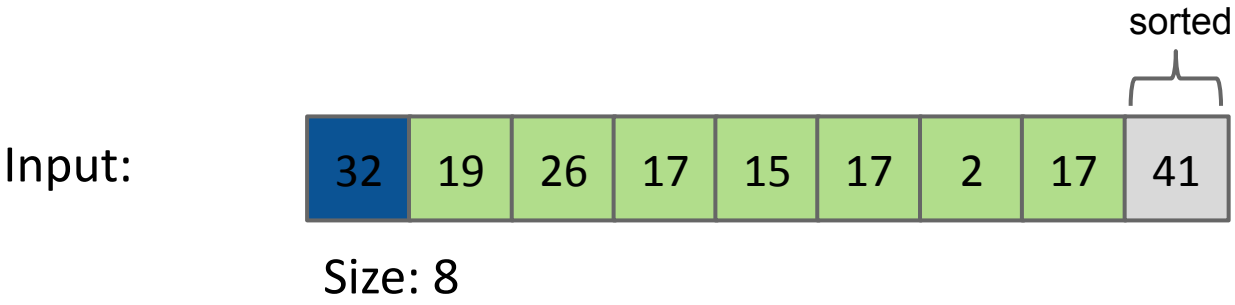


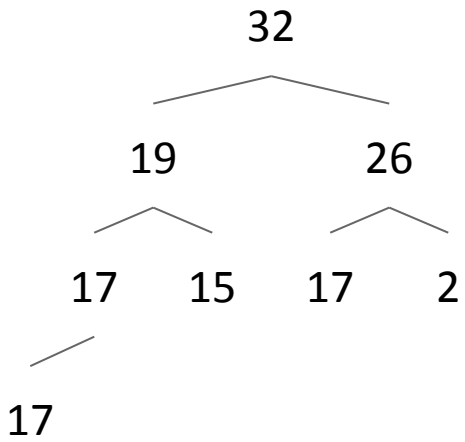


In-place Heap Sort

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.

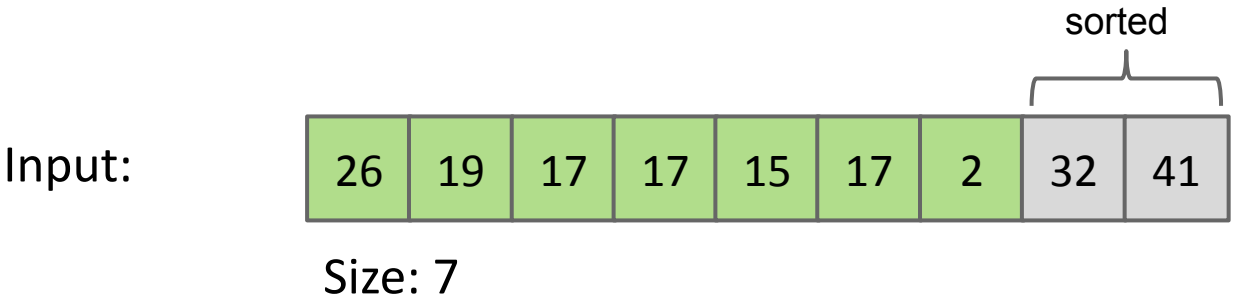


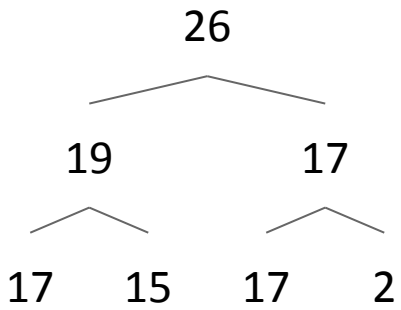


In-place Heap Sort

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.**

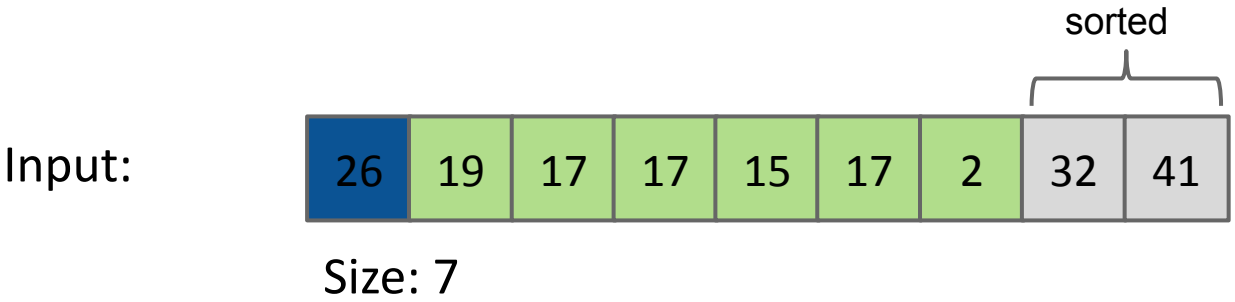


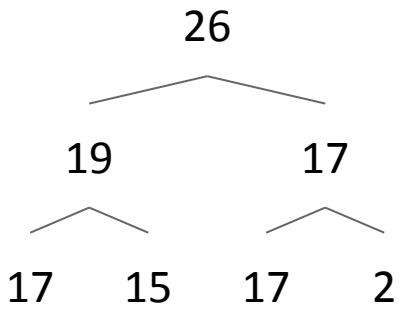


In-place Heap Sort

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.





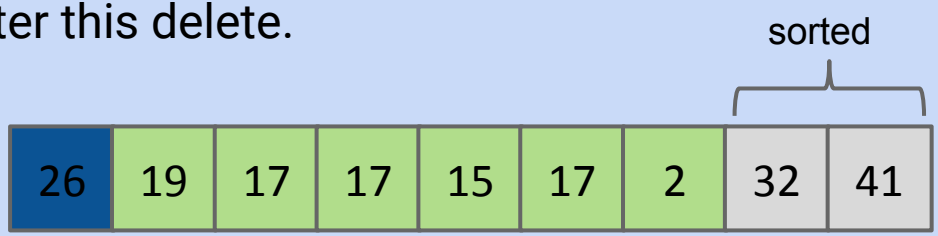
In-place Heap Sort

Heap sorting N items:

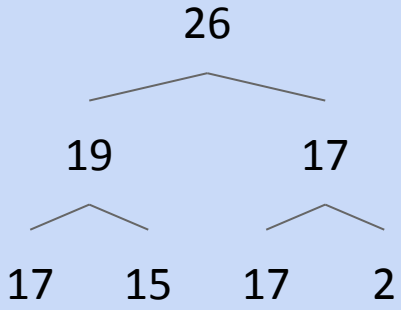
- Bottom-up heapify input array (done!).
- Repeat N times:
  - **Delete largest item from the max heap, swapping root with last item in the heap.**

Give the array after this delete.

Input:



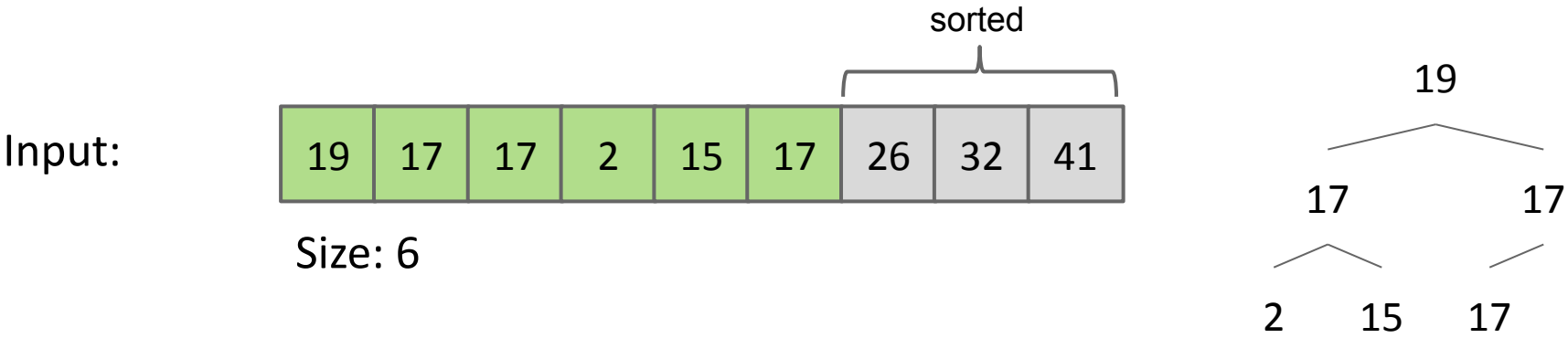
Size: 7



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.

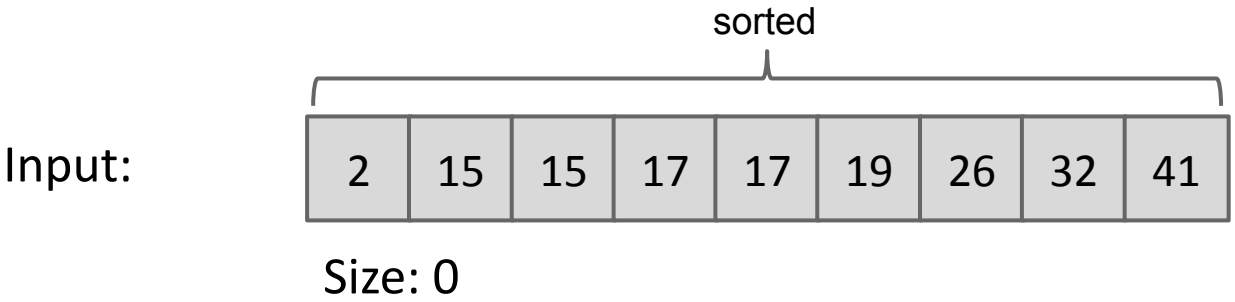


From here on out, the process is just the same, so verbose steps are omitted...

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.



Heapsort Runtime

Lecture 29, CS61B, Spring 2024

Goal: Sorting

The Sorting Problem

Selection Sort

Heapsort

- Naive Heapsort
- In-Place Heapsort
- **Heapsort Runtime**

Mergesort

In-place Heapsort Runtime: yellkey.com/TODO

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.

In-place Heapsort: yellkey.com/TODO

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: http://yellkey.com/tell

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.