LEC 20

CSE 373

Sorting I

BEFORE WE START

Which of the following is true about Path Compression?

a) It can be used on either QuickFind- or QuickUnion-based implementations of the DisjointSets ADT
b) It increases the asymptotic runtime of the find() operation with the time to modify the path
c) It improves the runtime of union() but doesn't improve the runtime of find()
d) The first call to find() will have no benefits from path compression

pollev.com/uwcse373

Instructor Aaron Johnston

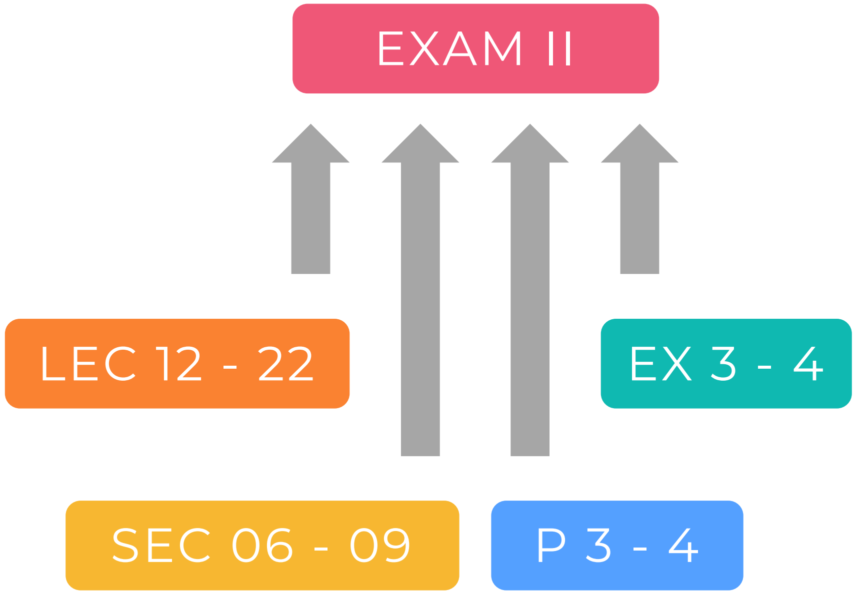
TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- EX3 late cutoff tonight 11:59pm
- EX4 released tonight, due ***Monday* 8/17**
  - Focuses on Minimum Spanning Trees & Sorting (including today's lecture and a little bit of Wednesday)
- P4 due next Wednesday 8/19
  - Starting now: ☺! Starting this weekend: ☹!
- TA-led Industry Panel Q&A
  - Come chat with a panel of your amazing TAs! Learn about their backgrounds/experiences and ask about careers in technology, finding internships, or preparing for interviews.
  - We're taking a survey to decide on a time. Please fill this out by Tuesday night: https://forms.gle/CBJVGeQXXCcDjUEQA

Exam II Logistics

- Due to overwhelmingly positive feedback about logistics, same as Exam I:
  - 48 hours to complete an exam written for 1-2 hours
  - Open notes & internet, groups up to 8
  - Submit via Gradescope, OH in lecture
- Released 8/21 12:01 AM PDT
- Due 8/22 11:59 PM PDT
  - **No late submissions!**
- Focuses on second half of the course, up through this Friday's lecture (Topo Sort)
  - But technically "cumulative" in that you *will* need to use skills from the first half (e.g. algorithmic analysis, use List/Stack/Queue/Map, etc.)
- Like Exam I, will emphasize conceptual and "why?" questions. Unlike Exam I, will require you to write short snippets of code!



STUDYING

- Topics list released tonight so you can start looking things over, practice materials published next Monday
- Remember to use the Learning Objectives!

Learning Objectives

After this lecture, you should be able to...

1. Implement the DisjointSets ADT as WeightedQuickUnion + PathCompression using an array, and describe its benefits
2. Define an ordering relation and stable sort and determine whether a given sorting algorithm is stable
3. Implement Selection Sort and Insertion Sort, compare runtimes and best/worst cases of the two algorithms, and decide when they are appropriate
4. Implement Heap Sort, describe its runtime, and implement the in-place variant

Lecture Outline

- *Review* Disjoint Sets
  - Implementing using Arrays

- Sorting
  - Definitions
  - Insertion & Selection Sort
  - Heap Sort

TRAVERSAL
(COMMONLY SHORTEST PATHS)

MINIMUM SPANNING TREES

Dijkstra's

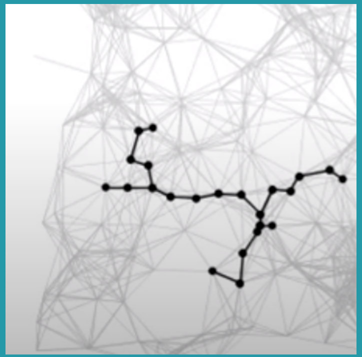
$\Theta(|V| \log|V| + |E| \log |V|)$

- Goes in order of shortest-path-so-far
- Choose when:
  - Want shortest path on *weighted* graph

Prim's

$\Theta(|E| \log |V|)$

- Goes vertex-by-vertex
- Choose when:
  - Want MST
  - Graph is dense (more edges)



Kruskal's

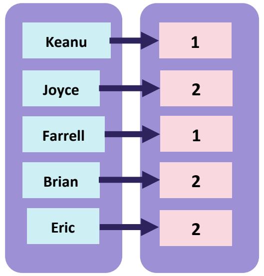
$\Theta(|E| \log |V|)$
or equivalently $\Theta(|E| \log |E|)$

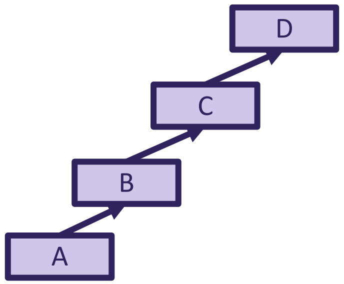
- Goes edge-by-edge
- Choose when:
  - Want MST
  - Graph is sparse (fewer edges)
  - Edges already sorted

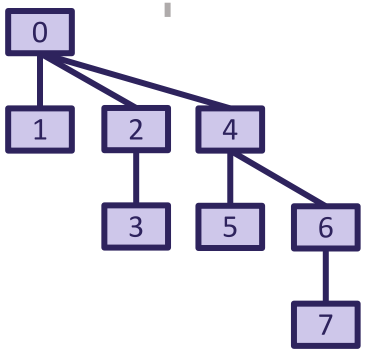
Review Disjoint Sets Implementation

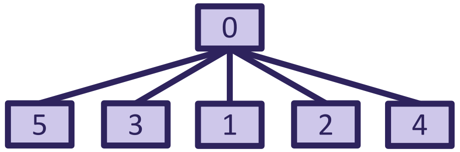
In-Practice Runtimes:

	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion	WQU + Path Compression
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$
`union(x, y)` `assuming root args`	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$









Review Kruskal's Runtime

$$\Theta(|E| \log |E|) \begin{cases} \Theta(|V|) \\ \Theta(|E| \log |E|) \\ \Theta(|E| \log |V|) \begin{cases} \Theta(|E|) \text{ iterations} \\ \Theta(\log |V|) \end{cases} \\ \Theta(|V| \log |V|) \begin{cases} \underline{\text{total}}\ \Theta(|V|) \text{ iterations} \\ \Theta(\log |V|) \end{cases} \end{cases}$$

```
kruskalMST(G graph)
  DisjointSets<V> msts; Set finalMST;
  initialize msts with each vertex as single-element MST
  sort all edges by weight (smallest to largest)

  for each edge (u,v) in ascending order:
    uMST = msts.find(u)
    vMST = msts.find(v)
    if (uMST != vMST):
      finalMST.add(edge (u, v))
      msts.union(uMST, vMST)
```

- find and union are log|V| in worst case, but amortized constant "in practice"
- Either way, dominated by time to sort the edges ☹
  - For an MST to exist, E can't be smaller than V, so assume it dominates
  - Note: some people write |E|log|V|, which is the same (within a constant factor)

DISJOINT SETS ADT

1
QuickFind

2
QuickUnion
Optimizes for the Union operation

3
Weighted QuickUnion
Avoids the worst case runtime for Find

4
Weighted QuickUnion + Path Compression
Makes future Find operations faster

DISJOINT SETS ADT

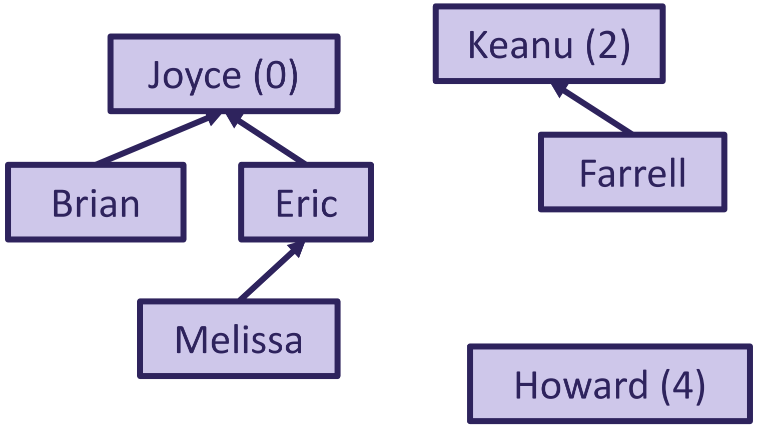
2
QuickUnion
Optimizes for the Union operation

3
Weighted QuickUnion
Avoids the worst case runtime for Find

4
Weighted QuickUnion + Path Compression
Makes future Find operations faster

5
ArrayWeighted QuickUnion + Path Compression
Better constant factors when stored in an array

Using Arrays for Up-Trees



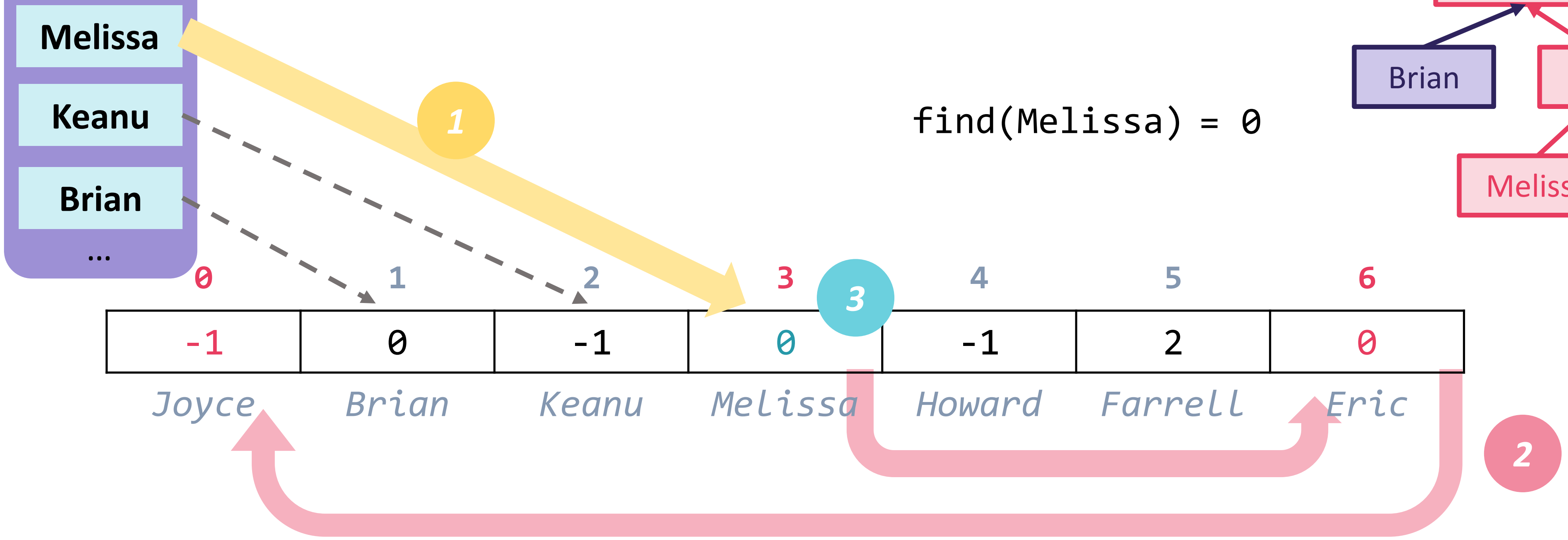
- Since every node can have at most one parent, what if we use an array to store the parent relationships?
- Proposal: each node corresponds to an index, where we store the index of the parent (or –1 for roots). Use the root index as the representative ID!
- Just like with heaps, tree picture still conceptually correct, but exists in our minds!

0	1	2	3	4	5	6
-1	0	-1	6	-1	2	0
Joyce	*Brian*	*Keanu*	*Melissa*	*Howard*	*Farrell*	*Eric*

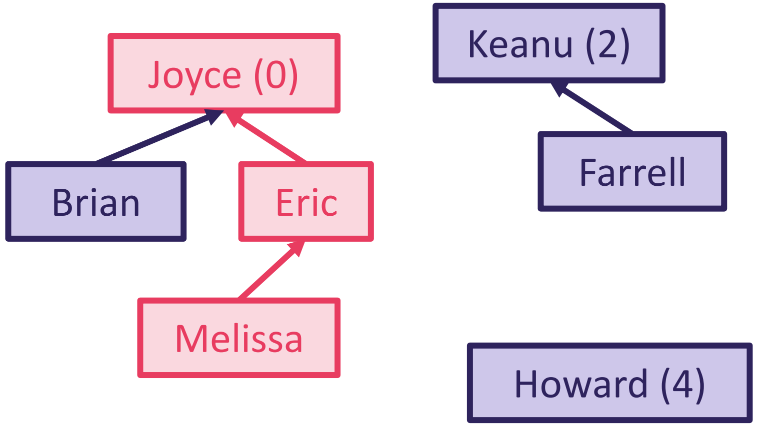
Using Arrays: Find

- Initial jump to element still done with extra Map
- But traversing up the tree can be done purely within the array!
  - Can still do path compression by setting all indices along the way to the root index!

```
find(A):
  index = jump to A node’s index
  while array[index] > 0:
    index = array[index]
  path compression
  return index
```



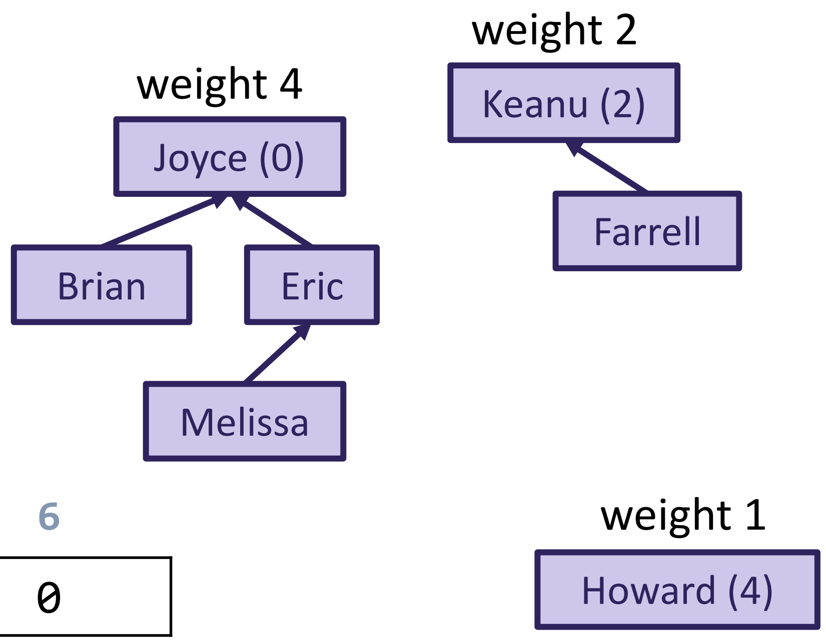
find(Melissa) = 0



Using Arrays: Union

- For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight)
- Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  use -1 * array[rootA] and -1 *
        array[rootB] to determine weights
  put lighter root under heavier root
```



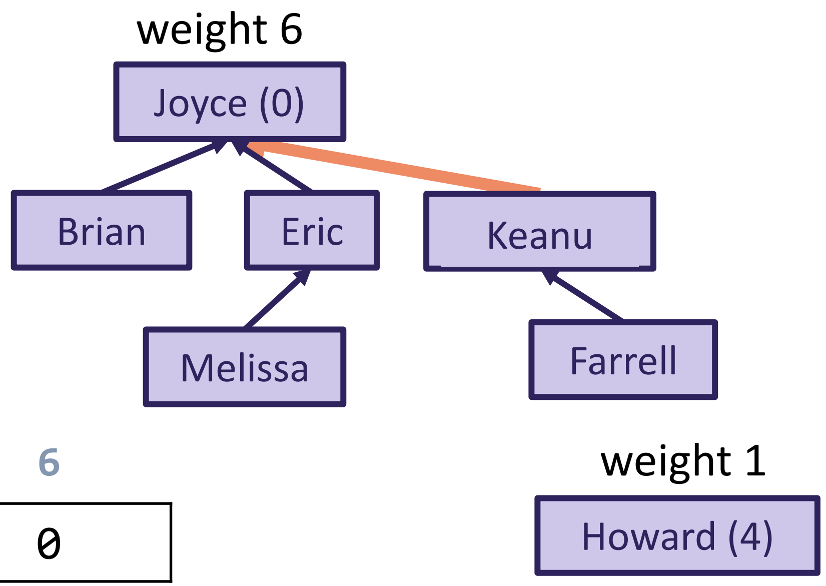
`union(Eric, Farrell)`

0	1	2	3	4	5	6
-4	0	-2	6	-1	2	0
Joyce	*Brian*	*Keanu*	*Melissa*	*Howard*	*Farrell*	*Eric*

Using Arrays: Union

- For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight)
- Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  use -1 * array[rootA] and -1 *
        array[rootB] to determine weights
  put lighter root under heavier root
```



`union(Eric, Farrell)`

0	1	2	3	4	5	6
-6	0	0	6	-1	2	0
Joyce	*Brian*	*Keanu*	*Melissa*	*Howard*	*Farrell*	*Eric*

Using Arrays for WQU+PC

- Same asymptotic runtime as using tree nodes, but check out all these other benefits:
  - More compact in memory
  - Better spatial locality, leading to better constant factors from cache usage
  - Simplify the implementation!

	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion	WQU + Path Compression	ArrayWQU+PC
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$	$O(\log^* n)$
`union(x, y)` `assuming root args`	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$	$O(\log^* n)$

Lecture Outline

- *Review* Disjoint Sets
  - Implementing using Arrays

- **Sorting**
  - **Definitions**
  - Insertion & Selection Sort
  - Heap Sort

Sorting

- Generally: given items, put them in order

- Why study sorting?
  - Sorting is incredibly common in programming
    - Often a component of other algorithms!
    - Very common in interviews
  - Interesting case study for approaching computational problems
    - We'll use some data structures we've already studied

Types of Sorts

1. Comparison Sorts

Compare two elements at a time.
Works whenever we could implement a compareTo method between elements.

We'll focus on comparison sorts: much more common, and very generalizable!

2. Niche Sorts

Leverage specific properties of data or problem to sort without directly comparing elements.
E.g. if you already know you'll only be sorting numbers < 5, make 5 buckets and add directly

Bonus topic beyond the scope of the class

Sorting: Definitions (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

- A **sort** is a permutation (re-arrangement) of a sequence of elements that puts the keys into non-decreasing order, relative to the ordering relation
  - $x_1 \leq x_2 \leq x_3 \leq \ldots \leq x_N$

```
int temperature
```

- Built-in, simple ordering relation

```
class Movie {
  String name;
  int year;
}
```

- More complex: Whenever we sort, *we also must decide* what ordering relation to use for that application
  - Sort by name?
  - Sort by year?
  - Some combination of both?

Sorting: Stability

- A sort is **stable** if the relative order of *equivalent* keys is maintained after sorting

INPUT

Anita *2010*	Basia *2018*	Caris *2019*	Duska *2020*	Duska *2015*	Anita *2016*

Stable sort using name as key

Anita *2010*	Anita *2016*	Basia *2018*	Caris *2019*	Duska *2020*	Duska *2015*

Unstable sort using name as key

Anita *2016*	Anita *2010*	Basia *2018*	Caris *2019*	Duska *2015*	Duska *2020*

- Stability and Equivalency only matter for complex types
  - i.e. when there is more data than just the key

Anita	Basia	Anita	Duska	Esteban	Duska	Caris

Anita	Anita	Basia	Caris	Duska	Duska	Esteban

Sorting: Performance Definitions

- Runtime performance is sometimes called the **time complexity**
  - Example: Dijkstra’s has time complexity O(E log V).

- <u>Extra</u> memory usage is sometimes called the **space complexity**
  - Dijkstra’s has space complexity Θ(V)
    - Priority Queue, distTo and edgeTo maps
  - The input graph takes up space Θ(V+E), but we don’t count this as part of the space complexity since the graph itself already exists and is an input to Dijkstra’s

Lecture Outline

- *Review* Disjoint Sets
  - Implementing using Arrays

- **Sorting**
  - Definitions
  - **Insertion & Selection Sort**
  - Heap Sort

Sorting Strategy 1: Iterative Improvement

- Invariants/Iterative improvement
  - Step-by-step make one more part of the input your desired output.

- We'll write iterative algorithms to satisfy the following invariant:
- After k iterations of the loop, the first k elements of the array will be sorted.

INVARIANT

Iterative Improvement

After k iterations of the loop, the first k elements of the array will be sorted

Selection Sort

https://www.youtube.com/watch?v=Ns4TPTC8whw

Every iteration, **select** the smallest unsorted item to fill the next spot.

0	1	2	3	4	5	6	7	8	9
2	3	6	7	18	10	14	9	11	15

Sorted Items | Current Item | Unsorted Items

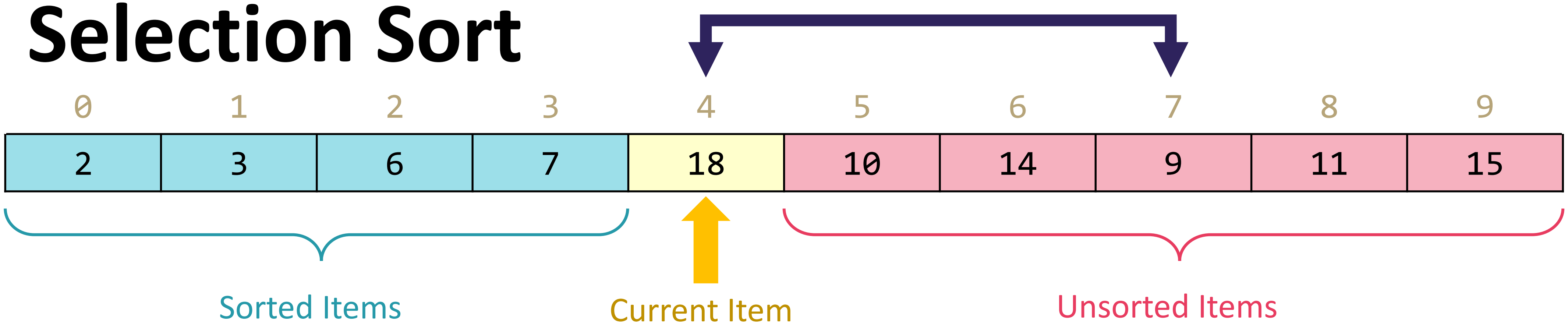
0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	14	18	11	15

Sorted Items | Current Item | Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	6	7	9	10	14	18	11	15

Sorted Items | Current Item | Unsorted Items

Selection Sort




```
void selectionSort(list) {
  for each current in list:
    target = findNextMin(current)
    swap(target, current)
}
int findNextMin(current) {
  min = current
  for each item in unsorted items:
    if (item < min):
      min = current
  return min
}
int swap(target, current) {
  temp = current
  current = target
  target = temp
}
```

Worst case runtime? $\Theta(n^2)$

Best case runtime? $\Theta(n^2)$

In-practice runtime? $\Theta(n^2)$

Stable? No

In-place? Yes

Selection Sort Stability

0	1	2	3	4	5	6
5a	3	4	5b	2	6	8

0	1	2	3	4	5	6
2	3	4	5b	5a	6	8

...

0	1	2	3	4	5	6
2	3	4	5b	5a	6	8

Swapping non-adjacent items can result in instability of sorting algorithms

Insertion Sort

Every iteration, **insert** the next unsorted item into the sorted items

https://www.youtube.com/watch?v=ROalU379l3U

0	1	2	3	4	5	6	7	8	9
2	3	6	7	5	1	4	10	2	8

Sorted Items

Current Item

Unsorted Items

0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

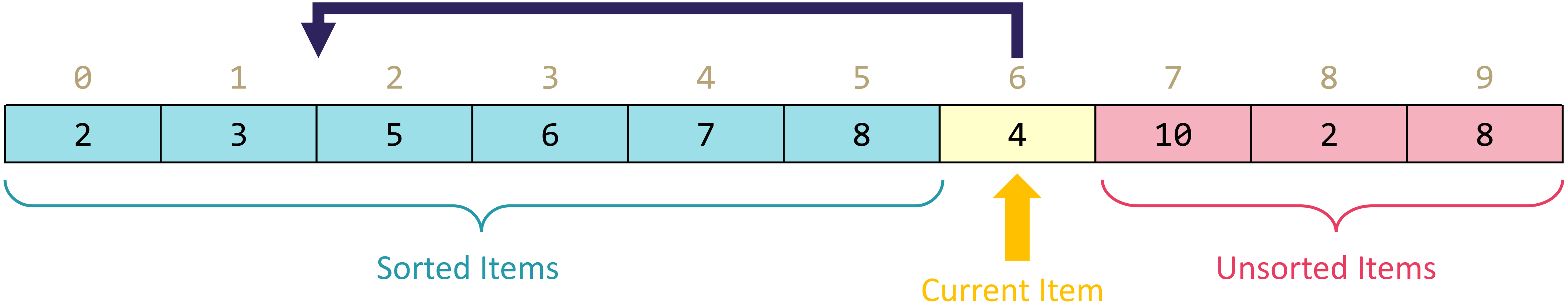
0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

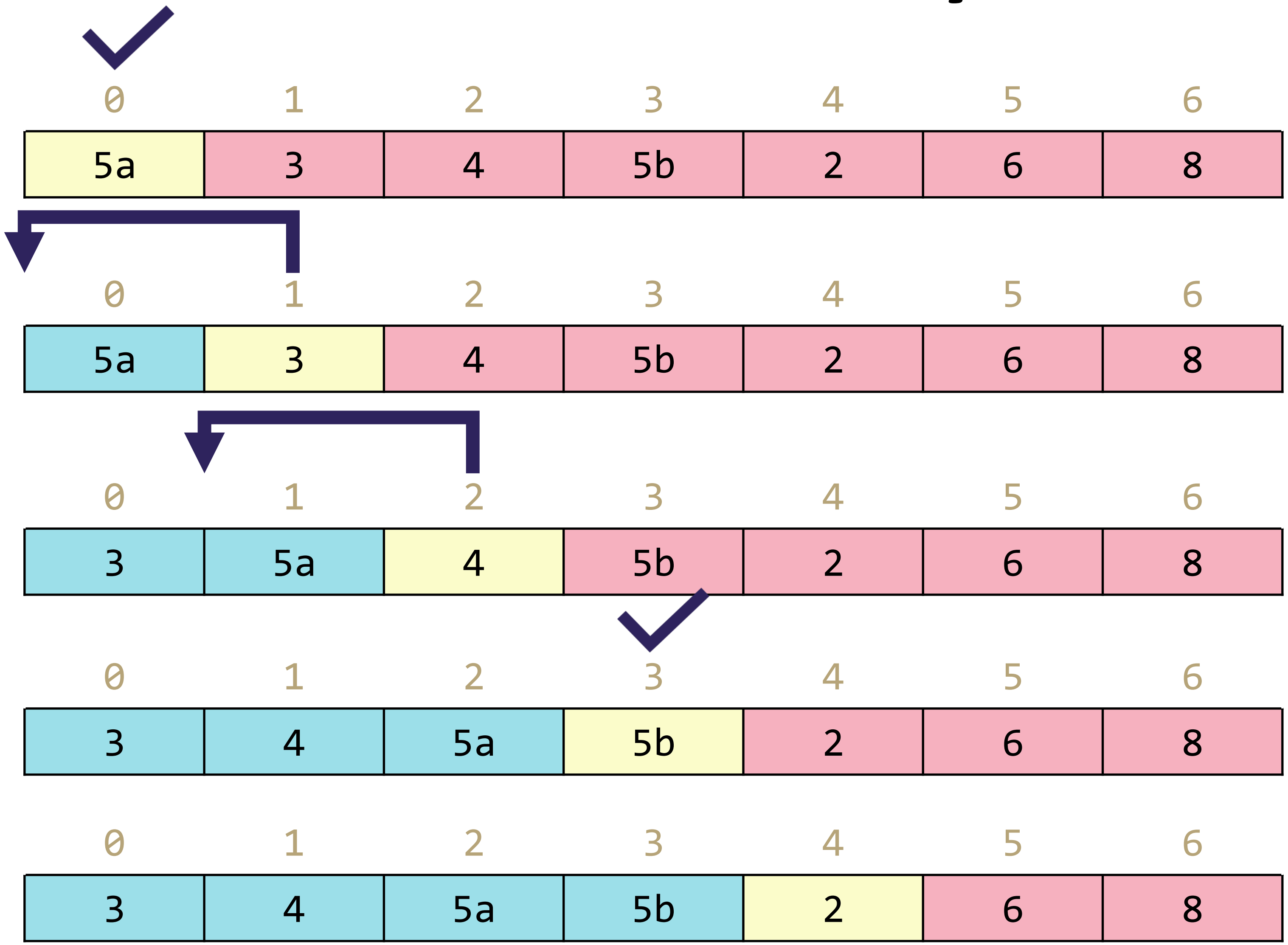
Insertion Sort




```
void insertionSort(list) {
  for each current in list:
    target = findSpot(current)
    shift(target, current)
}
int findSpot(current) {
  for each spot in sorted items going backwards:
    if (current goes in spot):
      return spot
}
void shift(target, current) {
  for (i = current; i > target; i--):
    item[i+1] = item[i]
  item[target] = current
}
```

Worst case runtime?	$\Theta(n^2)$
Best case runtime?	$\Theta(n)$
In-practice runtime?	$\Theta(n^2)$
Stable?	Yes
In-place?	Yes

Insertion Sort Stability



Insertion sort is **stable**!

- All swaps happen **between adjacent items** to get current item into correct relative position within sorted portion of array
- Duplicates will always be compared against one another **in their original orientation**, so can maintain stability with proper if logic

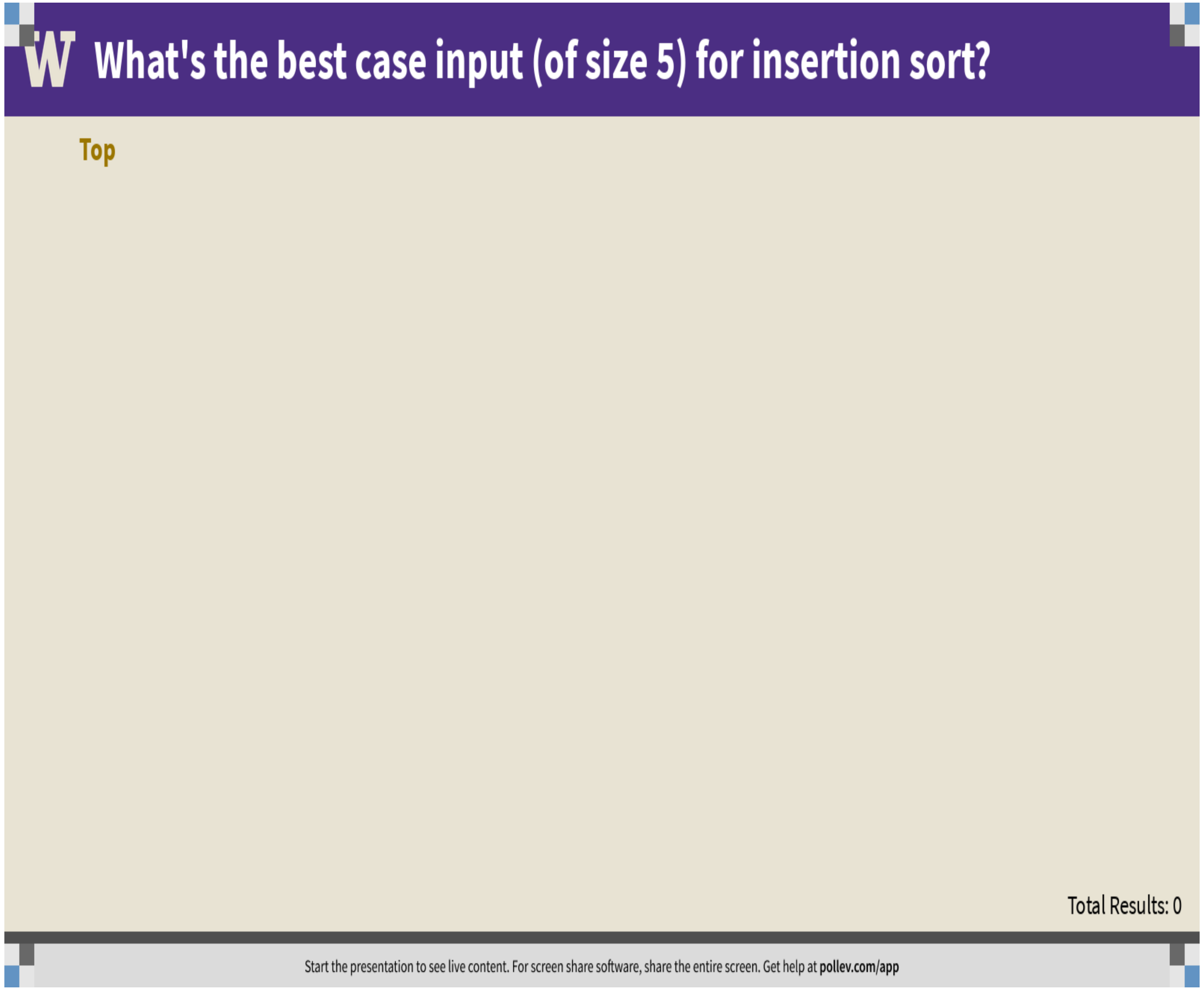
Poll Everywhere

pollev.com/uwcse373

```
void insertionSort(list) {
   for each current in list:
     target = findSpot(current)
     shift(target, current)
}
int findSpot(current) {
   for each spot in sorted items going backwards:
     if (current goes in spot):
       return spot
}
void shift(target, current) {
   for (i = current; i > target; i--):
     item[i+1] = item[i]
   item[target] = current
}
```

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n)$

Insertion Sort best case: when the input is already sorted!



Selection vs. Insertion Sort

```
void selectionSort(list) {
   for each current in list:
      target = findNextMin(current)
      swap(target, current)
}
```

"Look through unsorted to **select** the smallest item to replace the current item"

- Then **swap** the two elements

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n^2)$
In-practice runtime? $\Theta(n^2)$
Stable? No
In-place? Yes

Minimizes writing to an array (doesn't have to shift everything)

```
void insertionSort(list) {
   for each current in list:
      target = findSpot(current)
      shift(target, current)
}
```

"Look through sorted to **insert** the current item in the spot where it belongs"

- Then **shift** everything over to make space

Worst case runtime? $\Theta(n^2)$
Best case runtime? $\Theta(n)$
In-practice runtime? $\Theta(n^2)$
Stable? Yes
In-place? Yes

Almost always preferred: Stable & can take advantage of an already-sorted list. (LinkedList means no shifting ☺, though doesn't change asymptotic runtime)

0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

Lecture Outline

- *Review* Disjoint Sets
  - Implementing using Arrays

- **Sorting**
  - Definitions
  - Insertion & Selection Sort
  - **Heap Sort**

Sorting Strategy 2: Impose Structure

- Consider what contributes to Selection sort runtime of $\Theta(n^2)$
  - Unavoidable n iterations to consider each element
  - Finding next minimum element to swap requires a $\Theta(n)$ linear scan! Could we do better?

$\Theta(n)$ iterations

$\Theta(n)$

```
void selectionSort(list) {
   for each current in list:
      target = findNextMin(current)
      swap(target, current)
}
int findNextMin(current) {
   min = current
   for each item in unsorted items:
      if (item < min):
         min = current
   return min
}
```

- If only we knew a way to *structure* our *data* to make it fast to find the smallest item remaining in our dataset...

MIN PRIORITY QUEUE ADT

Heap Sort

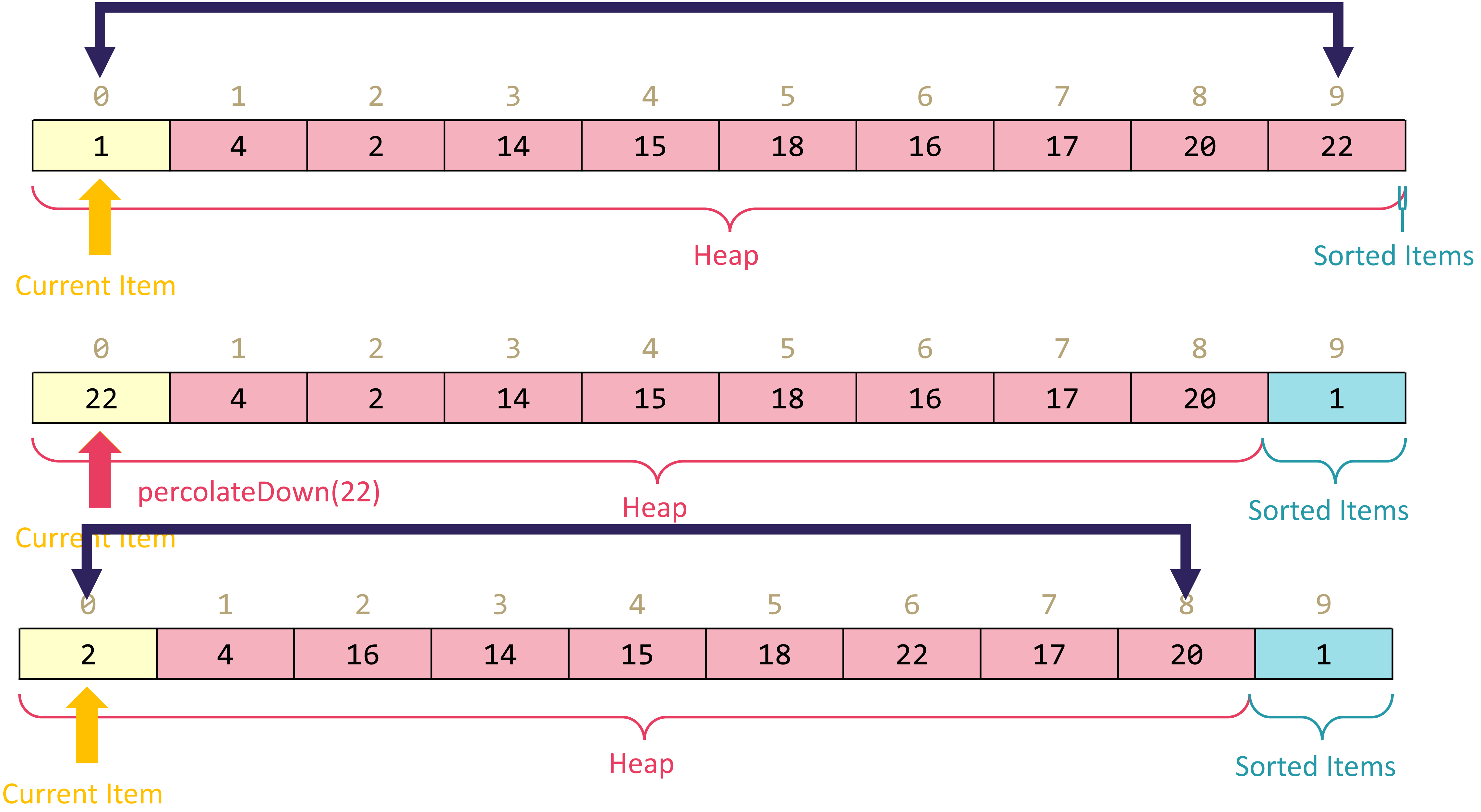
https://www.youtube.com/watch?v=Xw2D9aJRBY4

1. run Floyd's buildHeap on your data
2. call removeMin n times to pull out every element!

```
void heapSort(list) {
   E[] heap = buildHeap(list)
   E[] output = new E[n]
   for (i = 0; i < n; i++):
      output[i] = removeMin(heap)
}
```

Worst case runtime?	$\Theta(n \log n)$
Best case runtime?	$\Theta(n)$
In-practice runtime?	$\Theta(n \log n)$
Stable?	No
In-place?	If we get clever…

In-Place Heap Sort



In Place Heap Sort

0	1	2	3	4	5	6	7	8	9
15	17	16	18	20	22	14	4	2	1

Current Item

Heap

Sorted Items

```
void inPlaceHeapSort(list) {
   buildHeap(list) // alters original array
   for (n : list)
      list[n - i - 1] = removeMin(heap part of list)
}
```

Complication: final array is reversed! Lots of fixes:

- Run reverse afterwards ($O(n)$)
- Use a max heap
- Reverse compare function to emulate max heap

Worst case runtime? $\Theta(n \log n)$

Best case runtime? $\Theta(n)$

In-practice runtime? $\Theta(n \log n)$

Stable? No

In-place? Yes