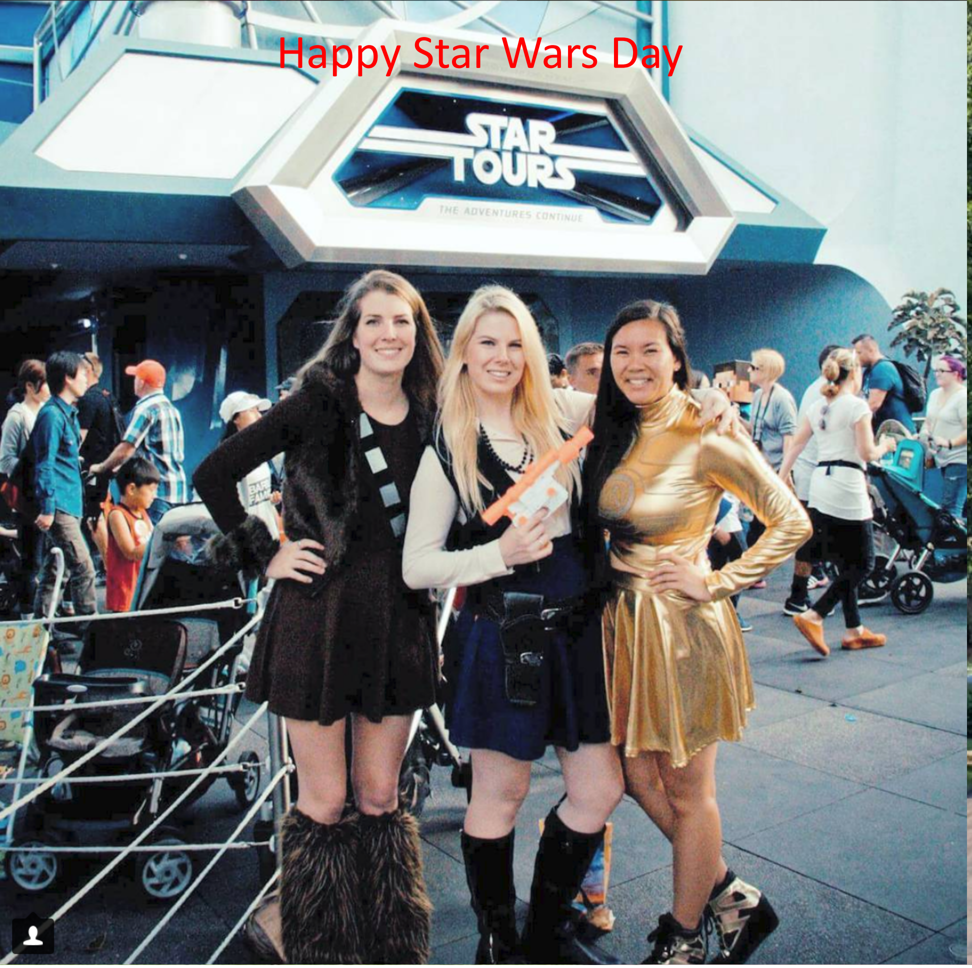


Implementing Heaps

Data Structures and Algorithms

Warm Up

Construct a Min Binary Heap by inserting the following values in this order:

5, 10, 15, 20, 7, 2

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

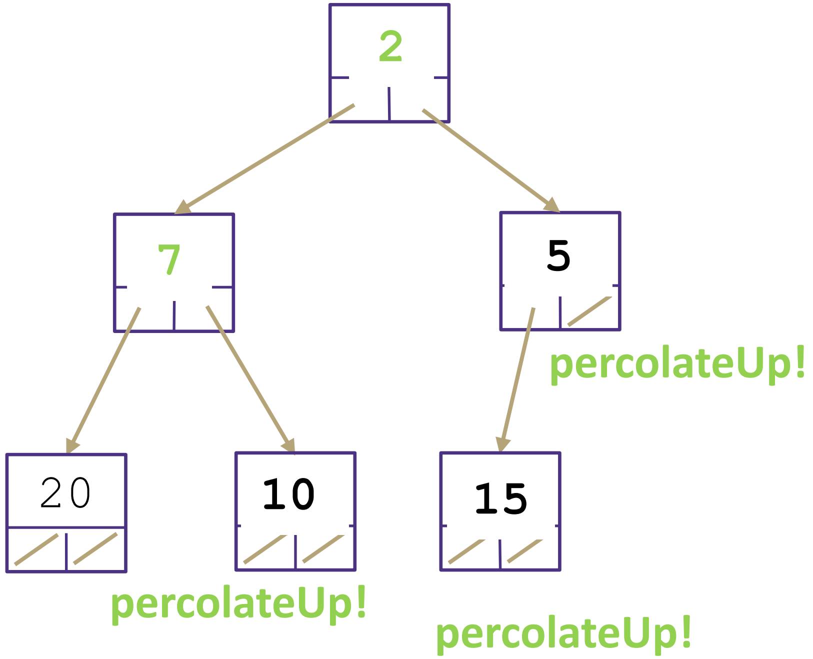
removeMin() – returns the element with the smallest priority, removes it from the collection

peekMin() – find, but do not remove the element with the smallest priority

insert(value) – add a new element to the collection

Min Binary Heap Invariants

1. **Binary Tree** – each node has at most 2 children
2. **Min Heap** – each node's children are larger than itself
3. **Level Complete** - new nodes are added from left to right completely filling each level before creating a new one

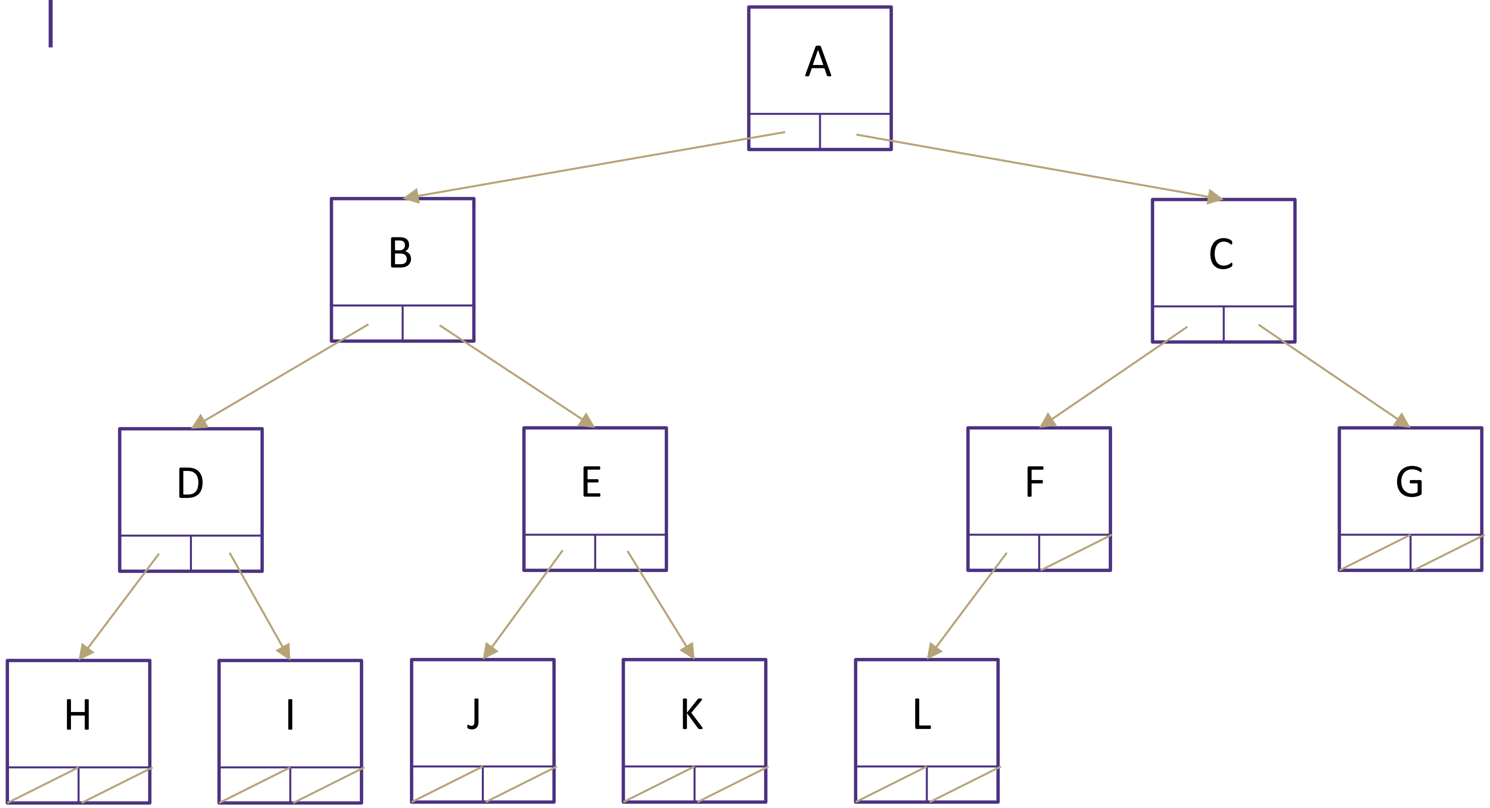


Midterm Grades Posted!

Statistics:

- Minimum: 22.0
- Maximum: 85.0
- Mean: 67.28 (79.1%)
- Median: 69.0 (81.1%)
- Standard Deviation: 10.05

Implementing Heaps



Fill array in **level-order** from left to right

0	1	2	3	4	5	6	7	8	9	10	11	12	13
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	

How do we find the minimum node?

$$peekMin() = arr[0]$$

How do we find the last node?

$$lastNode() = arr[size\ - 1]$$

How do we find the next open space?

$$openSpace() = arr[size]$$

How do we find a node's left child?

$$leftChild(i) = 2i + 1$$

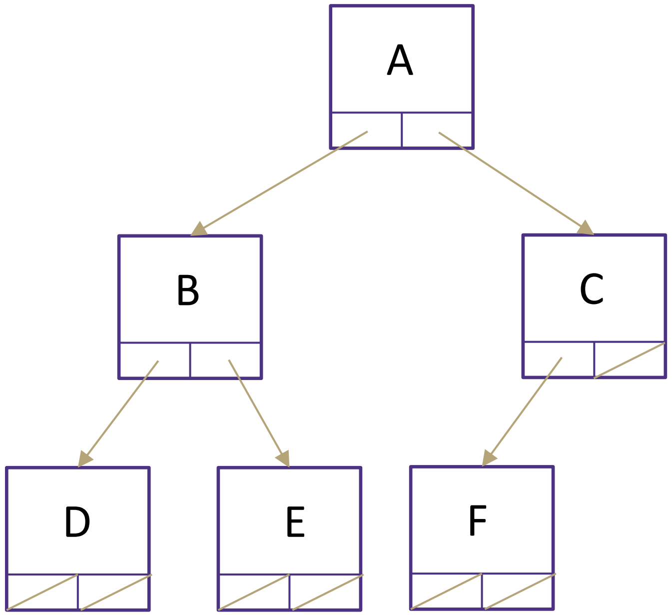
How do we find a node's right child?

$$rightChild(i) = 2i + 2$$

How do we find a node's parent?

$$parent(i) = \frac{i}{2}$$

Heap Implementation Runtimes



0	1	2	3	4	5	6	7
A	B	C	D	E	F		

`char peekMin()`

timeToFindMin

Tree	Θ(1)
Array	Θ(1)

`char removeMin()`

findLastNodeTime + removeRootTime + numSwaps * swapTime

Tree	n + 1 + log(n) * 1	Θ(n)
Array	1 + 1 + log(n) * 1	Θ(log(n))

`void insert(char)`

findNextSpace + addValue + numSwaps * swapTime

Tree	n + 1 + log(n) * 1	Θ(n)
Array	1 + 1 + log(n) * 1	Θ(log(n))

Building a Heap

Insert has a runtime of $\Theta(\log(n))$

If we want to insert a n items…

Building a tree takes $O(n\log(n))$

- Add a node, fix the heap, add a node, fix the heap

Can we do better?

- Add all nodes, fix heap all at once!

Cleaver building a heap – Floyd’s Method

Facts of binary trees

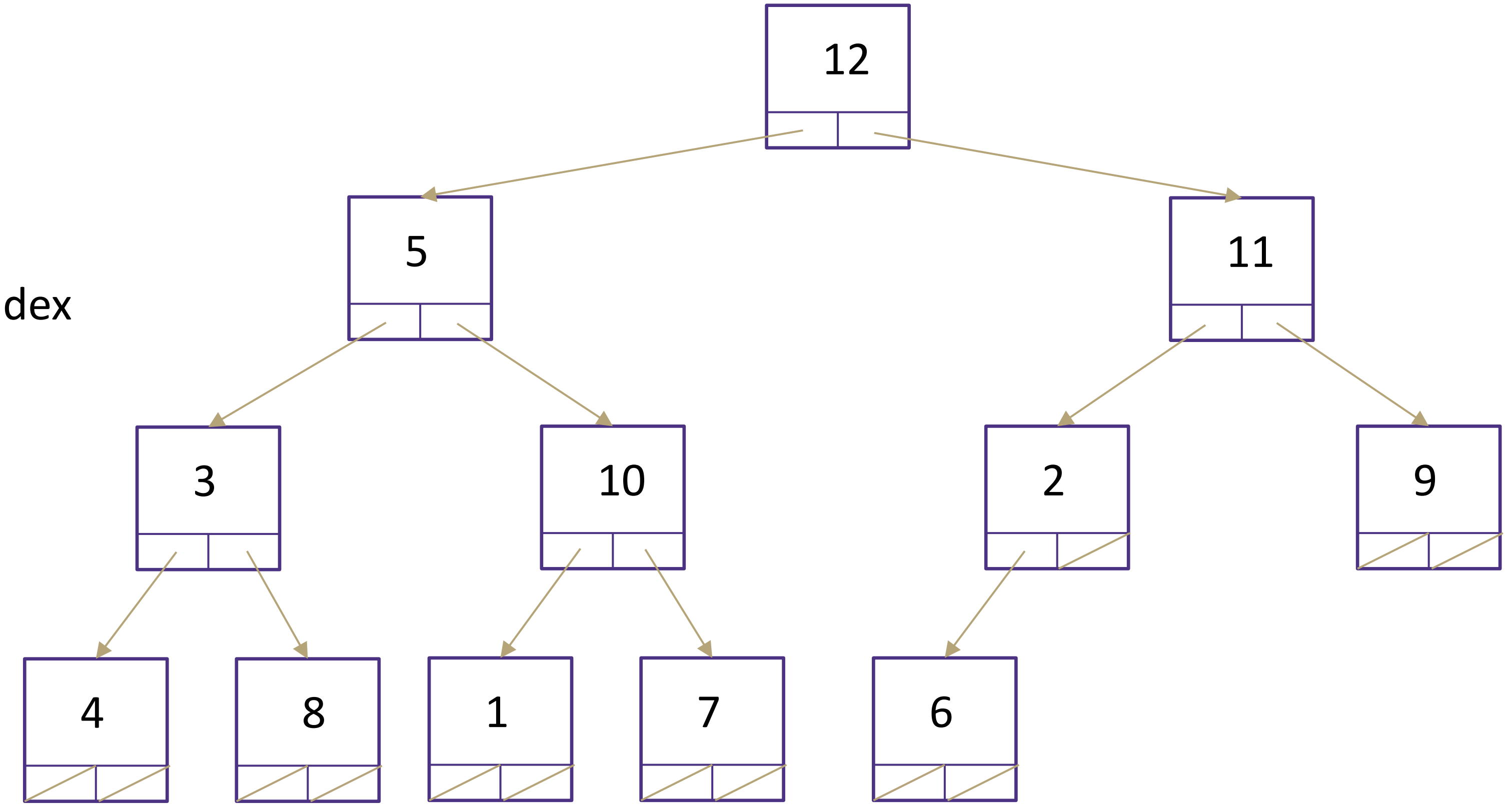
- Increasing the height by one level doubles the number of possible nodes
- A complete binary tree has half of its nodes in the leaves
- A new piece of data is much more likely to have to percolate down to the bottom than be the smallest element in heap

1. Dump all the new values into the bottom of the tree
   - Back of the array
2. Traverse the tree from bottom to top
   - Reverse order in the array
3. Percolate Down each level moving towards overall root

Floyd’s buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index

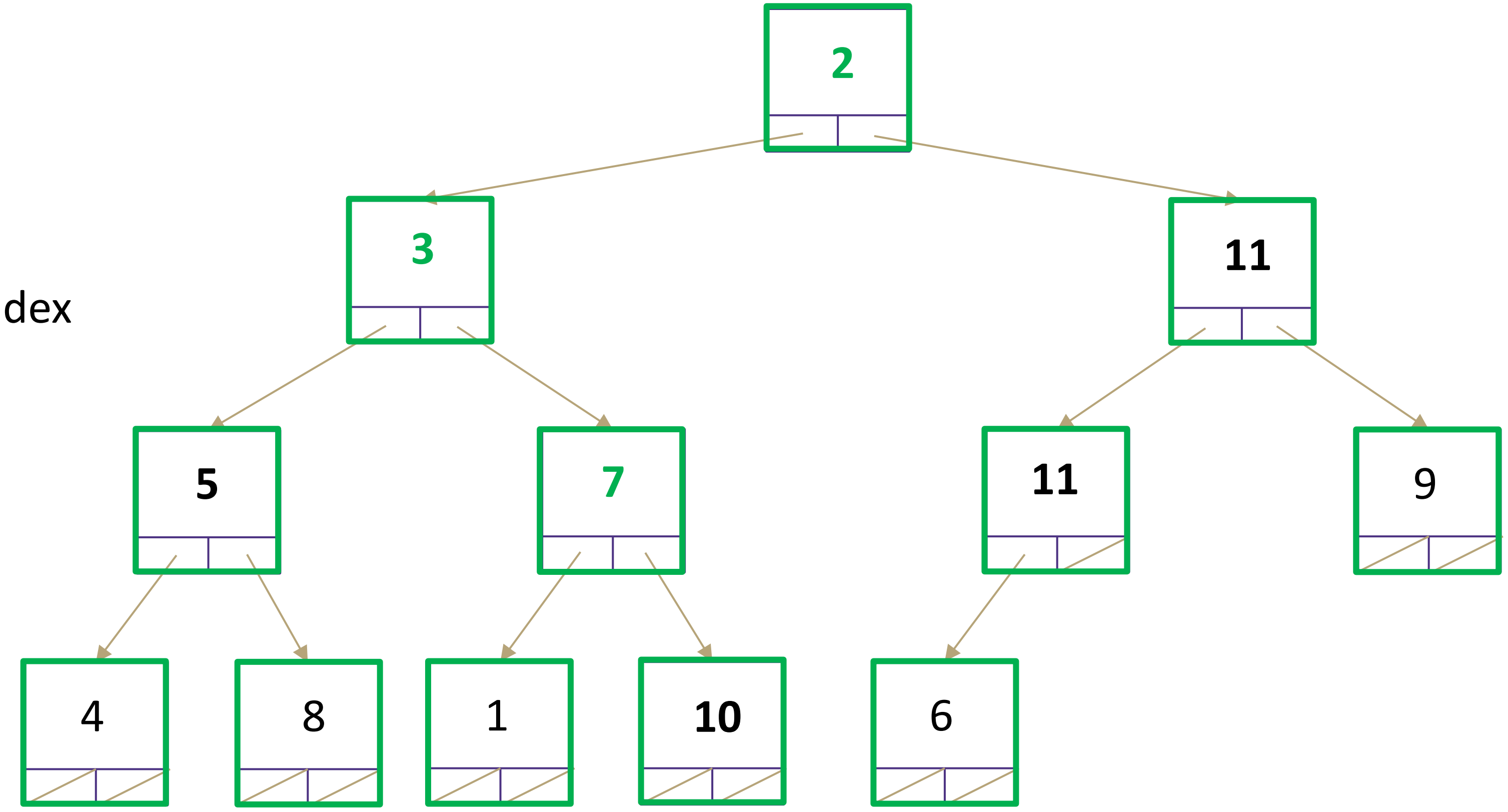


0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	1	7	6		

Floyd's buildHeap algorithm

Build a tree with the values:
12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6

1. Add all values to back of array
2. percolateDown(parent) starting at last index
   1. percolateDown level 4
   2. percolateDown level 3
   3. percolateDown level 2
   4. percolateDown level 1



0	1	2	3	4	5	6	7	8	9	10	11	12	13
12	5	11	3	10	2	9	4	8	1	7	6		

Floyd's Heap Runtime

We step through each node – n

We call percolateDown() on each n – log n

thus it's O(nlogn)

… let's look closer…

Are we sure percolateDown() runs log n each time?

- Half the nodes of the tree are leaves
  - Leaves run percolate down in constant time
- ¼ the nodes have at most 1 level to travel
- 1/8 the nodes have at most 2 levels to travel
- etc…

work(n) ≈ n/2 * 1 + n/4 * 2 + n/8 * 3 + …

Closed form Floyd's buildHeap

work(n) ≈ $\frac{n}{2} * 1 + \frac{n}{4} * 2 + \frac{n}{8} * 3 + \ldots$

factor out n

work(n) ≈ n($\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \ldots$)

find a pattern -> powers of 2

work(n) ≈ n($\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots$)

Summation!

$$work(n) \approx n \sum_{i=1}^{?} \frac{i}{2^i}$$

? = how many levels = height of tree = log(n)

Infinite geometric series

$$work(n) \approx n \sum_{i=1}^{\log n} \frac{i}{2^i}$$

$$if -1 < x < 1 \ then \sum_{i=0}^{\infty} x^i = \frac{1}{1 - x} = x$$

$$work(n) \approx n \sum_{i=1}^{\log n} \frac{i}{2^i} \leq n \sum_{i=0}^{\infty} \frac{i}{2^i} = n * 2$$

Floyd's buildHeap runs in O(n) time!

Sorting

Types of Sorts

Comparison Sorts

Compare two elements at a time

General sort, works for most types of elements

Element must form a "consistent, total ordering"

For every element a, b and c in the list the following must be true:

- If a <= b and b <= a then a = b
- If a <= b and b <= c then a <= c
- Either a <= b is true or <= a

What does this mean? compareTo() works for your elements

Comparison sorts run at fastest O(nlog(n)) time

Niche Sorts aka "linear sorts"

Leverages specific properties about the items in the list to achieve faster runtimes

niche sorts typically run O(n) time

In this class we'll focus on comparison sorts

Sort Approaches

In Place sort

A sorting algorithm is in-place if it requires only O(1) extra space to sort the array

Typically modifies the input collection

Useful to minimize memory usage

Stable sort

A sorting algorithm is stable if any equal items remain in the same relative order before and after the sort

Why do we care?

- Sometimes we want to sort based on some, but not all attributes of an item
- Items that “compareTo()” the same might not be exact duplicates
- Enables us to sort on one attribute first then another etc…

[**(8, “fox”)**, (9, “dog”), (4, “wolf”), **(8, “cow”)**]

[(4, “wolf”), **(8, “fox”)**, **(8, “cow”)**, (9, “dog”)] Stable

[(4, “wolf”), **(8, “cow”)**, **(8, “fox”)**, (9, “dog”)] Unstable

SO MANY SORTS

Quicksort, Merge sort, in-place merge sort, heap sort, insertion sort, intro sort, selection sort, timsort, cubesort, shell sort, bubble sort, binary tree sort, cycle sort, library sort, patience sorting, smoothsort, strand sort, tournament sort, cocktail sort, comb sort, gnome sort, block sort, stackoverflow sort, odd-even sort, pigeonhole sort, bucket sort, counting sort, radix sort, spreadsort, burstsort, flashsort, postman sort, bead sort, simple pancake sort, spaghetti sort, sorting network, bitonic sort, bogosort, stooge sort, insertion sort, slow sort, rainbow sort…

Insertion Sort

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

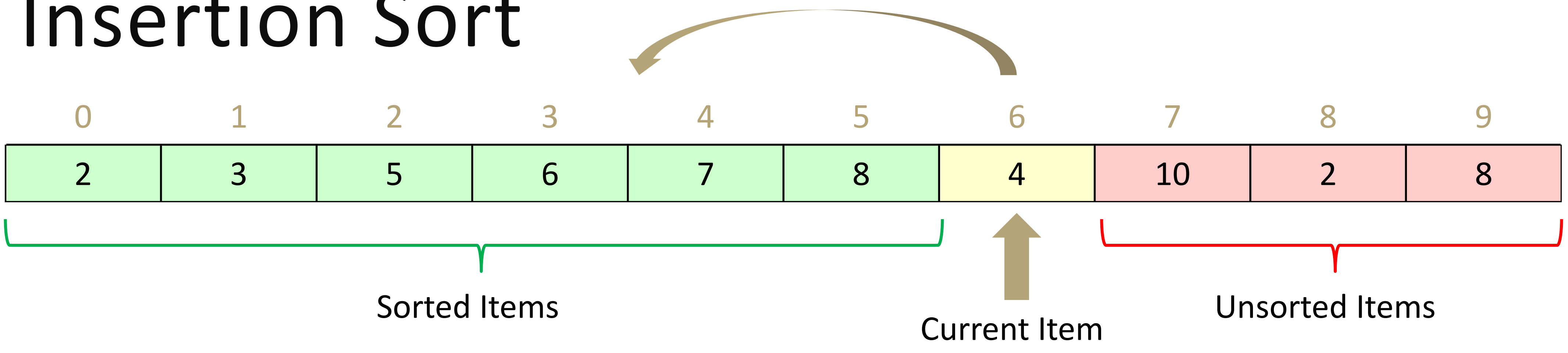
0	1	2	3	4	5	6	7	8	9
2	3	5	6	7	8	4	10	2	8

Sorted Items

Current Item

Unsorted Items

Insertion Sort




```
public void insertionSort(collection) {
   for (entire list)
      if(currentItem is bigger than nextItem)
         int newIndex = findSpot(currentItem);
         shift(newIndex, currentItem);
}
public int findSpot(currentItem) {
   for (sorted list)
      if (spot found) return
}
public void shift(newIndex, currentItem) {
   for (i = currentItem > newIndex)
      item[i+1] = item[i]
   item[newIndex] = currentItem
}
```

Worst case runtime? $O(n^2)$

Best case runtime? $O(n)$

Average runtime? $O(n^2)$

Stable? Yes

In-place? Yes

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

Selection Sort

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	18	14	11	15

Sorted Items | Current Item | Unsorted Items

Selection Sort

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

```
public void selectionSort(collection) {
   for (entire list)
      int newIndex = findNextMin(currentItem);
      swap(newIndex, currentItem);
}
public int findNextMin(currentItem) {
   min = currentItem
   for (unsorted list)
      if (item < min)
         min = currentItem
   return min
}
public int swap(newIndex, currentItem) {
   temp = currentItem
   currentItem = newIndex
   newIndex = currentItem
}
```

Worst case runtime? $O(n^2)$

Best case runtime? $O(n^2)$

Average runtime? $O(n^2)$

Stable? Yes

In-place? Yes

Heap Sort

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

1. run Floyd's buildHeap on your data

2. call removeMin n times

```
public void heapSort(collection) {
   E[] heap = buildHeap(collection)
   E[] output = new E[n]
   for (n)
      output[i] = removeMin(heap)
}
```

Worst case runtime?	O(nlogn)
Best case runtime?	O(nlogn)
Average runtime?	O(nlogn)
Stable?	No
In-place?	No

In Place Heap Sort

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

Heap

Sorted Items

Current Item

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

percolateDown(22)

Heap

Sorted Items

Current Item

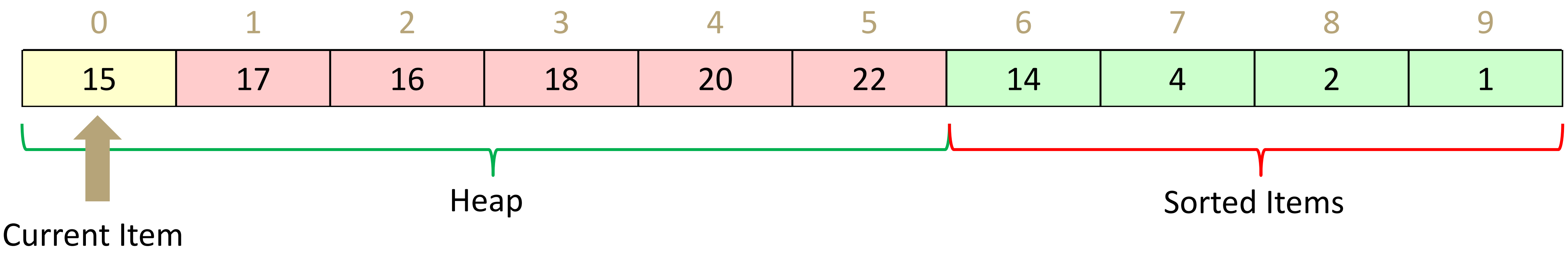
0	1	2	3	4	5	6	7	8	9
2	4	16	14	15	18	22	17	20	1

Heap

Sorted Items

Current Item

In Place Heap Sort




```
public void inPlaceHeapSort(collection) {
   E[] heap = buildHeap(collection)
   for (n)
      output[n – i - 1] = removeMin(heap)
}
```

Complication: final array is reversed!
- Run reverse afterwards (O(n))
- Use a max heap
- Reverse compare function to emulate max heap

Worst case runtime? O(nlogn)

Best case runtime? O(nlogn)

Average runtime? O(nlogn)

Stable? No

In-place? Yes

Divide and Conquer Technique

1. Divide your work into smaller pieces recursively
   - Pieces should be smaller versions of the larger problem
2. Conquer the individual pieces
   - Base case!
3. Combine the results back up recursively

```
divideAndConquer(input) {
   if (small enough to solve)
      conquer, solve, return results
   else
      divide input into a smaller pieces
      recurse on smaller piece
      combine results and return
}
```

Merge Sort

https://www.youtube.com/watch?v=XaqR3G_NVoo

Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Conquer

0
8

0
8

Combine

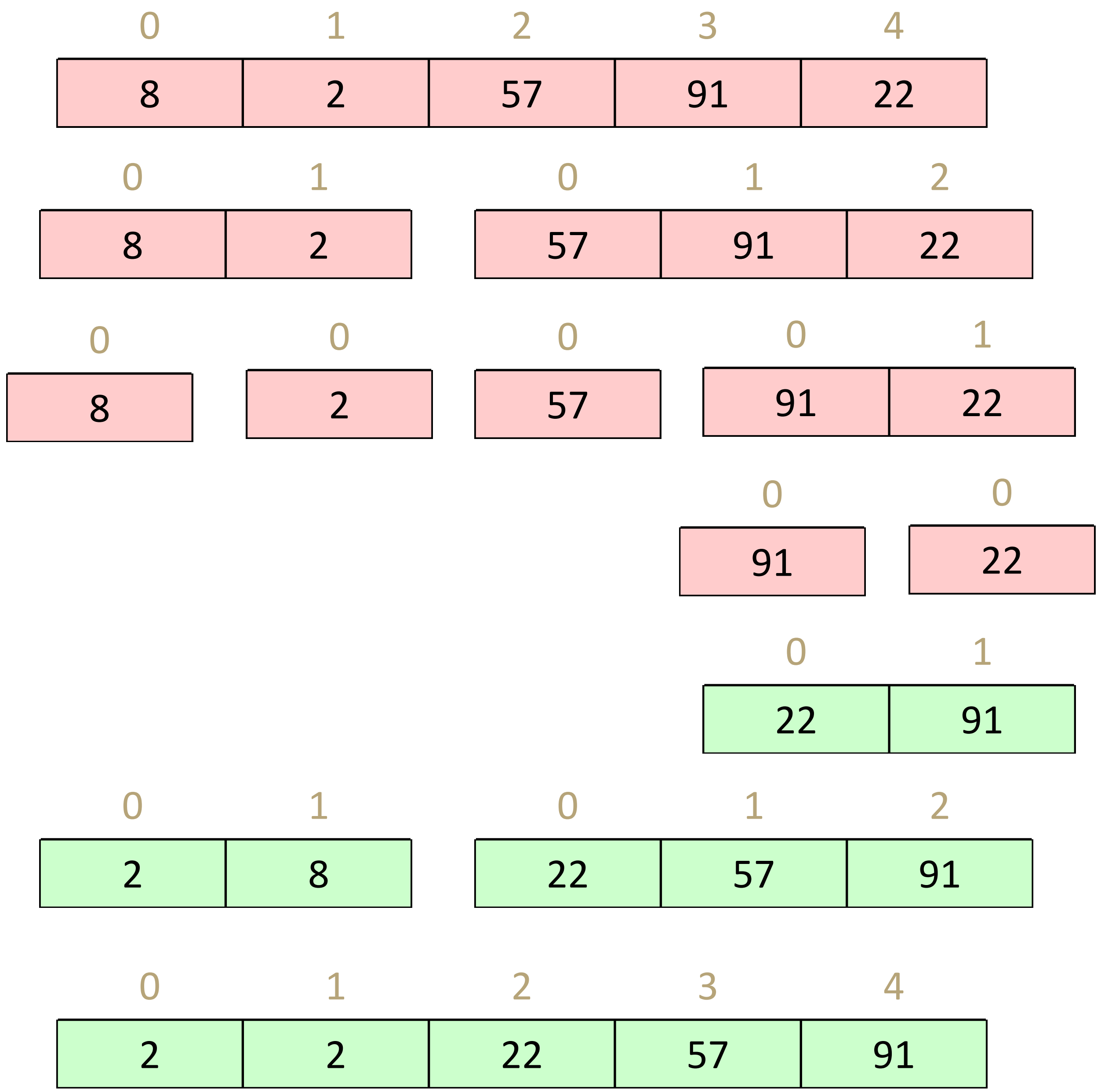
0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ..., mid])
      largerHalf = mergeSort(new [mid + 1, ...])
      return merge(smallerHalf, largerHalf)
}
```



Worst case runtime?

Best case runtime? $T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$

Average runtime?

Stable? No

In-place? No