Week 13 - Monday

COMP 2100

Last time

- What did we talk about last time?
- Exam 2 post mortem
- Quicksort

Questions?

Project 4

Assignment 7

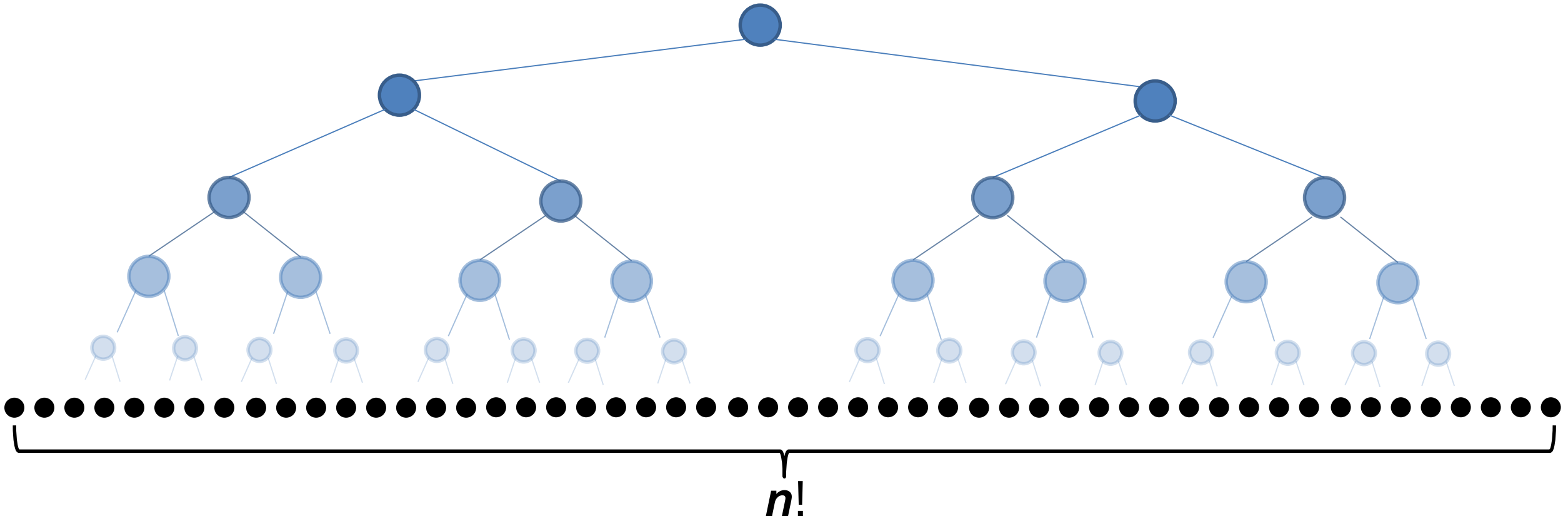
Lower Bound on Sorting

The fastest sort

- How many ways are there to order ***n*** items?
- ***n*** different things can go in the first position, leaving ***n*** – 1 to go in the second position, leaving ***n*** – 2 things to go into the third position…
- ***n*** (***n*** – 1) (***n*** – 2) … (2)(1) = ***n***!
- In other words, there are ***n***! different orderings, and we have to do some work to find the ordering that puts everything in sorted order

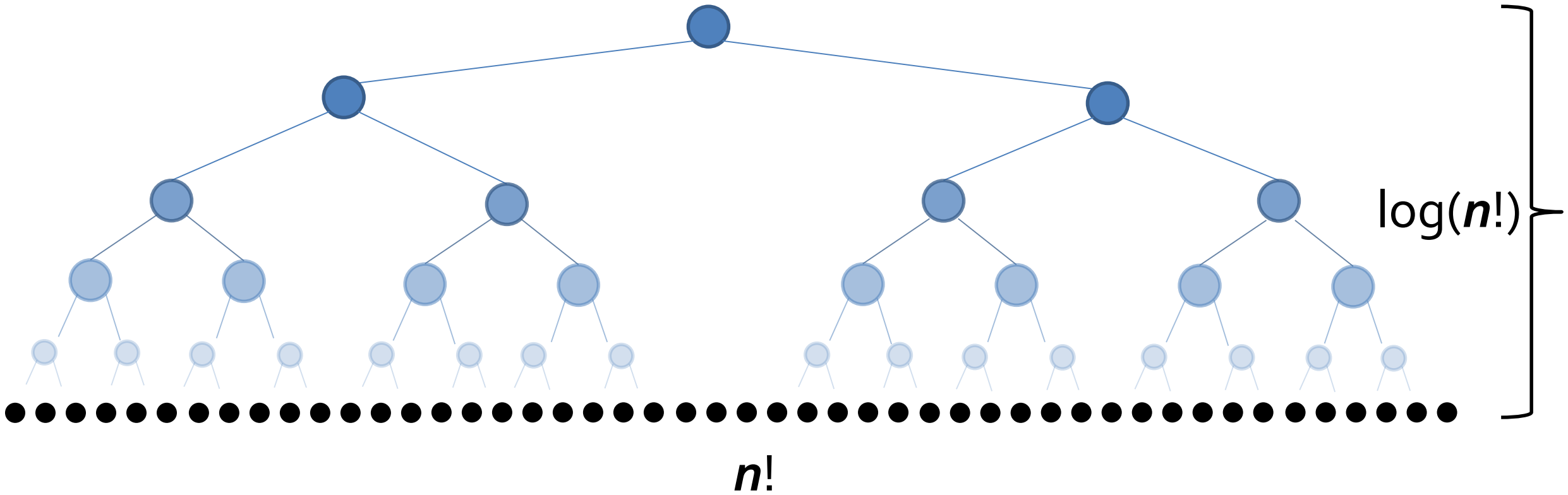
A different kind of tree

- Imagine a tree of **decisions**
- Some sequence of decisions will lead to a leaf of the tree
- Each leaf of the tree **represents** one of those ***n***! orders



Tree height

- What is the smallest height the tree could have?
- A perfectly balanced binary tree with ***k*** leaves will have a height of $\log_2(k)$
- Since we have ***n***! leaves, the smallest height will be $\log_2(n!)$



Comparison-based sorts

- **Any** comparison-based sort is going to compare two values and make a decision based on that
- No matter what your algorithm is, if each comparison is a decision in the tree that leads you down to a sorted order, the best you can possibly do is $\log_2(n!)$
- But what is $\log_2(n!)$?
- I wish I could show you the math that backs this up, but Stirling's approximation says that $\log_2(n!)$ is $\Theta(n \log n)$
- **Take away:** No comparison-based sort can **ever** be better than $\Theta(n \log n)$ for worst-case running time

Counting Sort

Counting sort justification

- Lets focus on an unusual sort that lets us (potentially) get better performance than $\Theta(n \log n)$
- But, I thought $\Theta(n \log n)$ was the theoretical maximum!
- Some sorts don't require **comparison** of values

Counting sort paradigm

- You use counting sort when you know that your data is in a narrow range, like, the numbers between **1** and **10** or even **1** and **100**
- As long as the range of possible values is in the neighborhood of the length of your list, counting sort can do well
- **Example: 150** integer grades between **1** and **100**
- Doesn't work for sorting **`double`** or **`String`** values

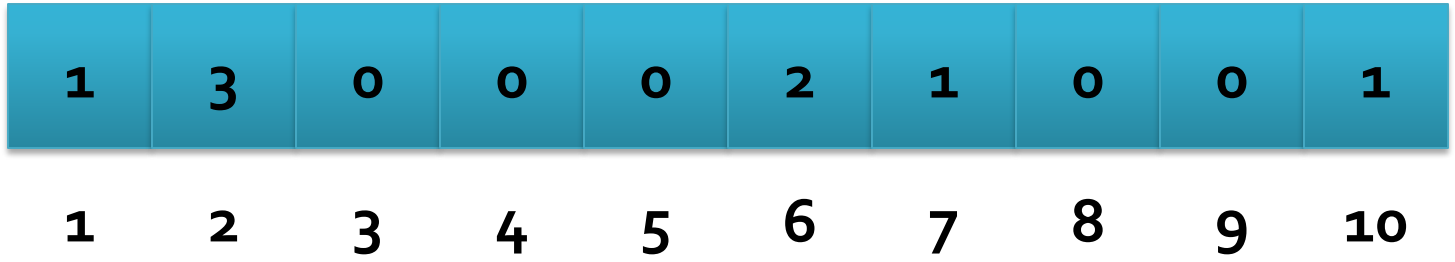
Counting sort algorithm

- Make an array with enough elements to hold every possible **value** in your range of values
  - If you need 1 – 100, make an array with length 100
- Sweep through your original list of numbers, when you see a particular value, increment the corresponding index in the value array
- To get your final sorted list, sweep through your value array and, for every entry with value ***k*** > 0, print its index ***k*** times

Counting sort example

- We know our values will be in the range [1,10]
- Our example array:

- Our values array:



- The result:

Counting sort implementation

```
public static void countingSort(
    int[] numbers, int min, int max) {
    …
}
```

- The numbers in values are guaranteed to fall between **`min`** (inclusive) and **`max`** (inclusive).
- Note that the **`min`** and **`max`** could be negative.

How long does it take?

- It takes $O(n)$ time to scan through the original array
- But, now we have to take into account the number of values we expect
- So, let's say we have m possible values
- It takes $O(m)$ time to scan back through the value array, with $O(n)$ additional updates to the original array
- Time: $O(n + m)$

Radix Sort

Radix sort

- We can "generalize" counting sort somewhat
- Instead of looking at the value as a whole, we can look at individual digits (or even individual characters)
- First, we collect everything whose ones places is a 0
  - Then, we collect everything whose ones place is a 1
    - Then, we collect everything whose ones place is a 2
      - …
- If we store everything that ends with a 0, then everything that ends with a 1, then everything that ends with a 2, etc., will the list be sorted?
  - No!
- But if we take that list of numbers and then repeat the process on the tens places, hundreds place, and so on, for however many places we need, they will be sorted!

Intuition

7	45	0	54	37	108	51

- If we had a way to sort everything first by the ones place:

0	51	54	45	7	37	108

- Then by the tens place:

0	7	108	37	45	51	54

- Then by the hundreds place:

0	7	37	45	51	54	108

- This array would be sorted, since it only goes up to the hundreds

Radix sort

- For decimal numbers, we would only need 10 buckets (0 – 9)
- We count how many things would go into each bucket, which can allow us to copy the values with a particular digit into a scratch array based on the size of each digit's range
- Then we copy everything back into the original array
- The book discusses MSD and LSD string sorts, which are similar

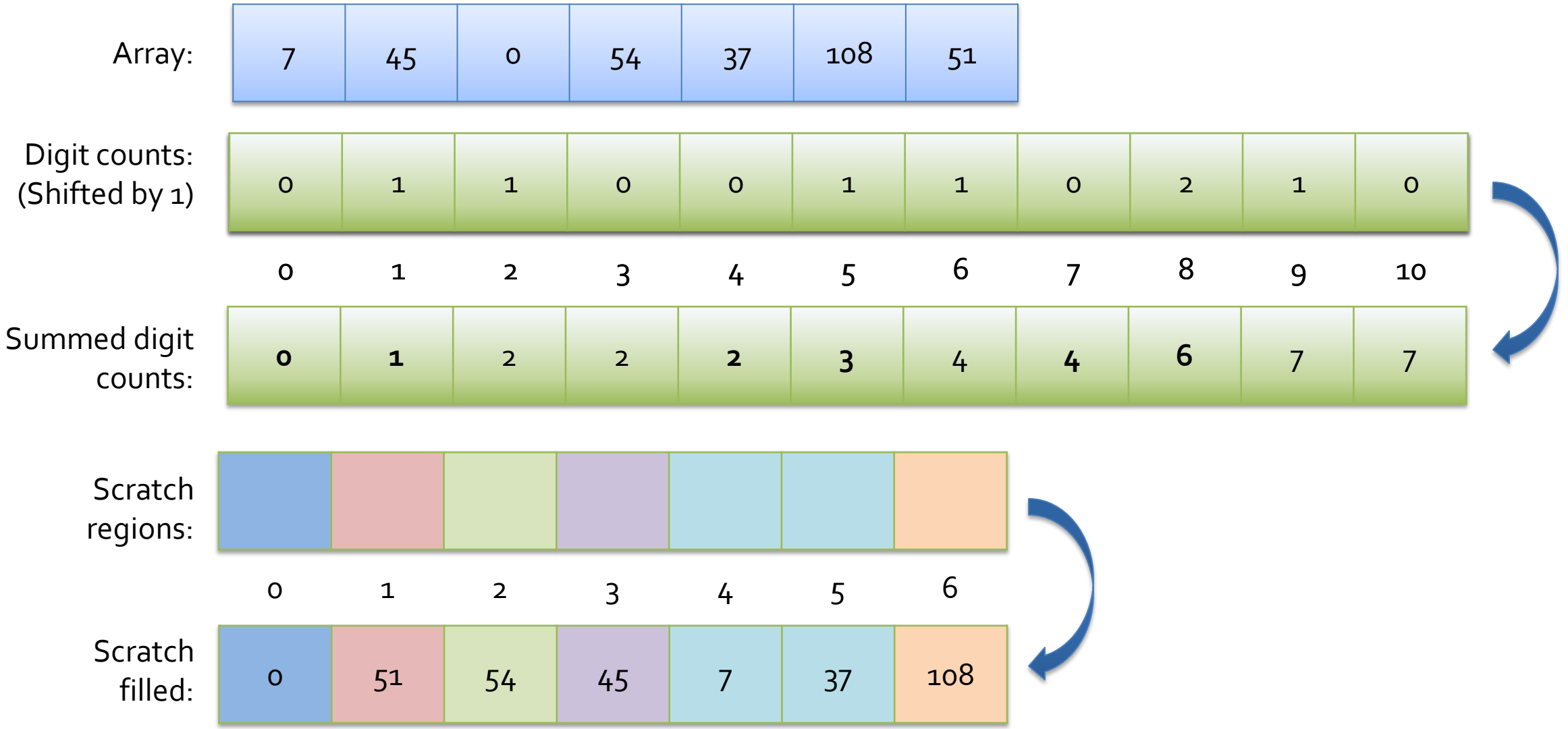
Radix sort

- Pros:
  - Best, worst, and average case running time of O(***nk***) where ***k*** is the number of digits we look at
  - Stable for least significant digit (LSD) version
  - Surprisingly fast
- Cons:
  - Requires a fixed number of digits to be checked (even if most numbers are shorter)
  - Unstable for most significant digit (MSD) version
  - Works poorly for floating point and non-digit based keys
    - But can work for strings!
  - Not in-place

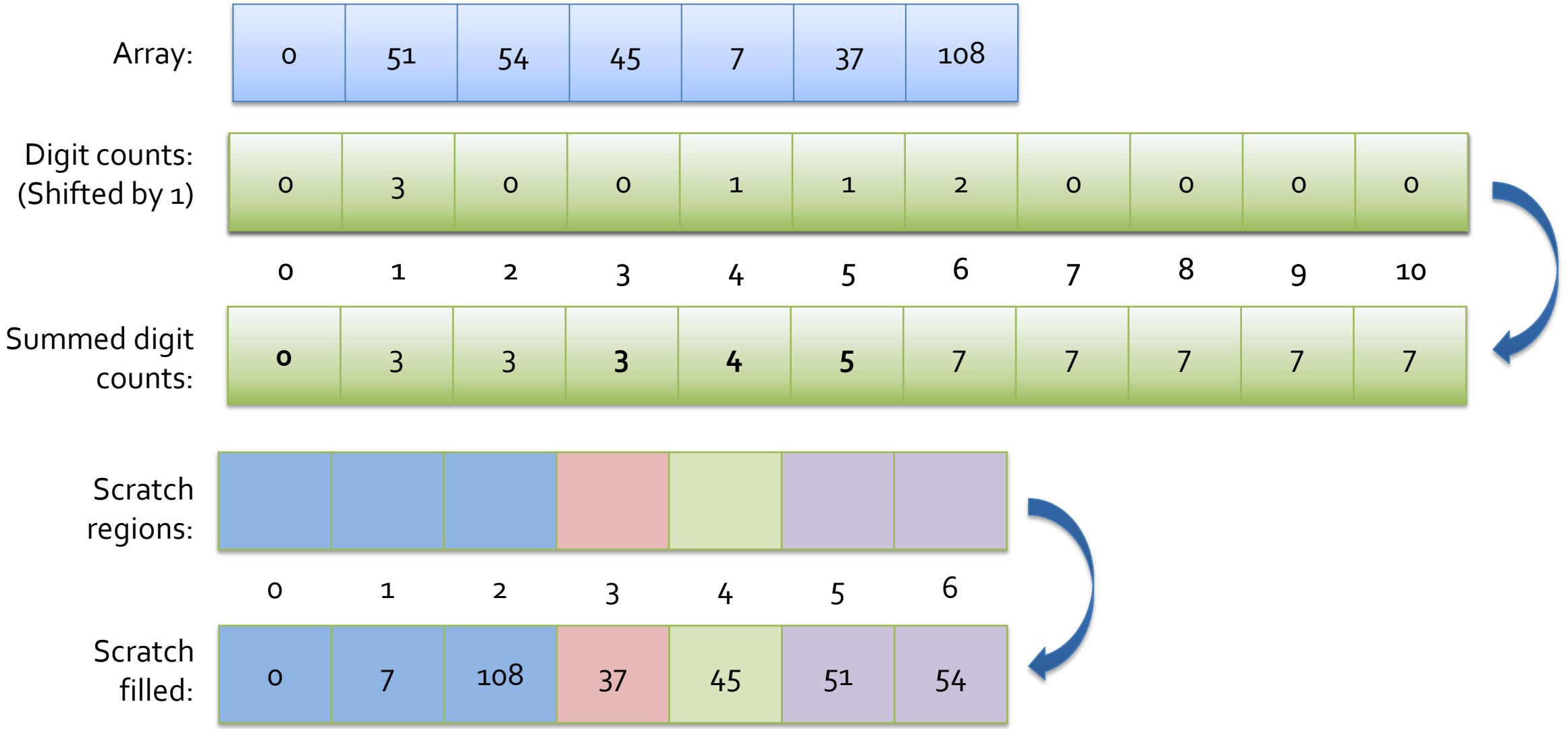
Radix sort algorithm

- For integers, make 10 buckets (0-9)
  - Actually, we make 11 buckets to make computing the starting points for each range of digit values easier
- Loop through our numbers, counting how many numbers would go into each bucket based on the digit in the current place (except store it in the next digit up)
- Loop through our digit counts, summing the previous values to find the starting points of each range
- Loop through our numbers again, copying each one into a scratch array in the first open spot in its digit range
  - Increase the counter for the digit range so we know the next open spot
- Copy everything from the scratch array back into the original array
- If there are more digits to consider, move to the next digit and repeat the process

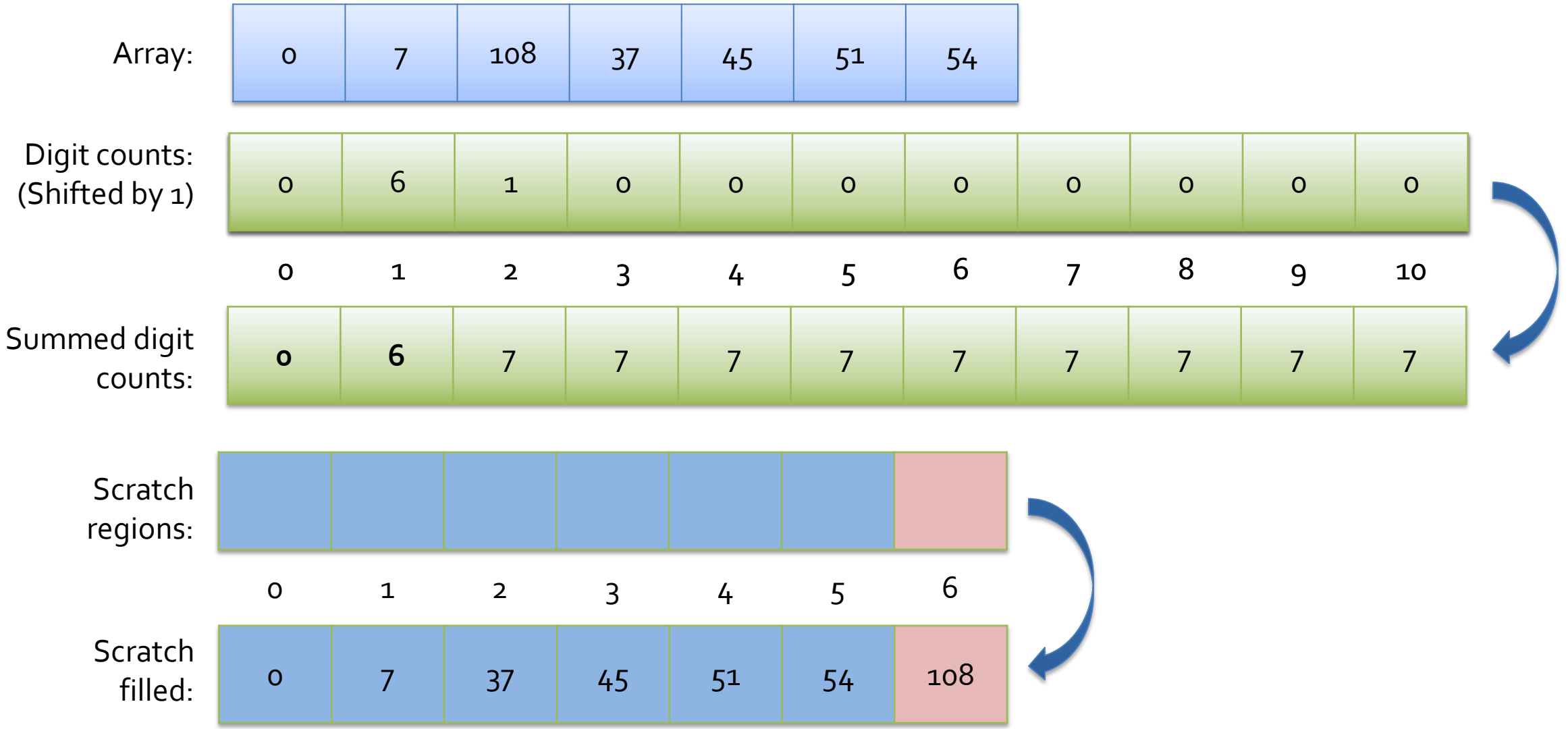
Radix sort example (ones place)



Radix sort example (tens place)



Radix sort example (hundreds place)



Radix sort implementation

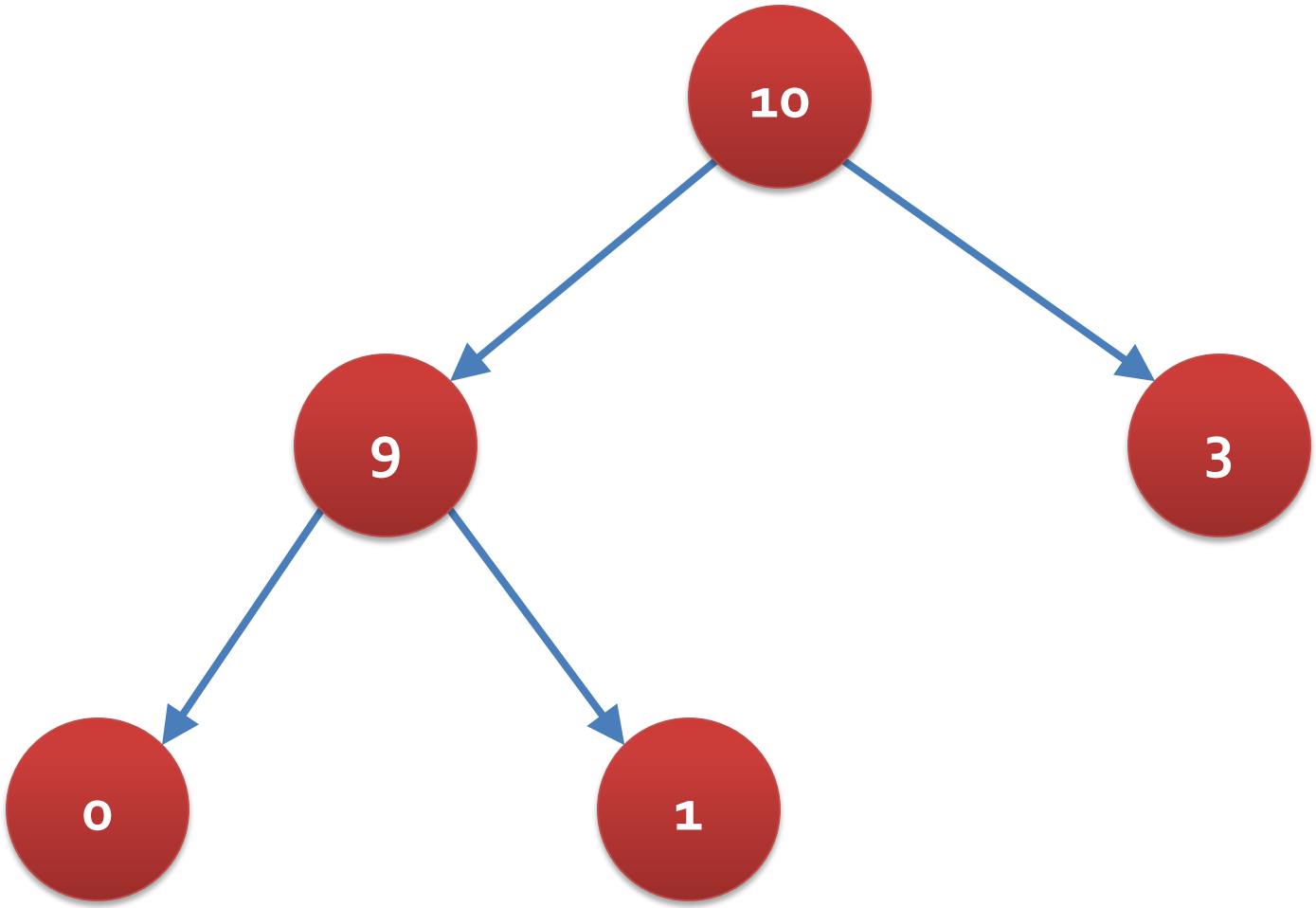
- In recap:
  1. Count how many keys go into each bucket
  2. After doing so, add up values in the count array so that the starting point of each bucket in the final array is known
  3. Copy all values into their correct bucket ranges in a scratch array
  4. Copy all values back into the original array
- Repeat for each place value: ones, tens, hundreds, etc.
- After ordering everything in increasing place values, the array will be sorted

Heaps

Heaps

- A **maximum heap** is a **complete** binary tree where
  - The left and right children of the root have key values less than the root
  - The left and right subtrees are also maximum heaps
- We can define **minimum heaps** similarly

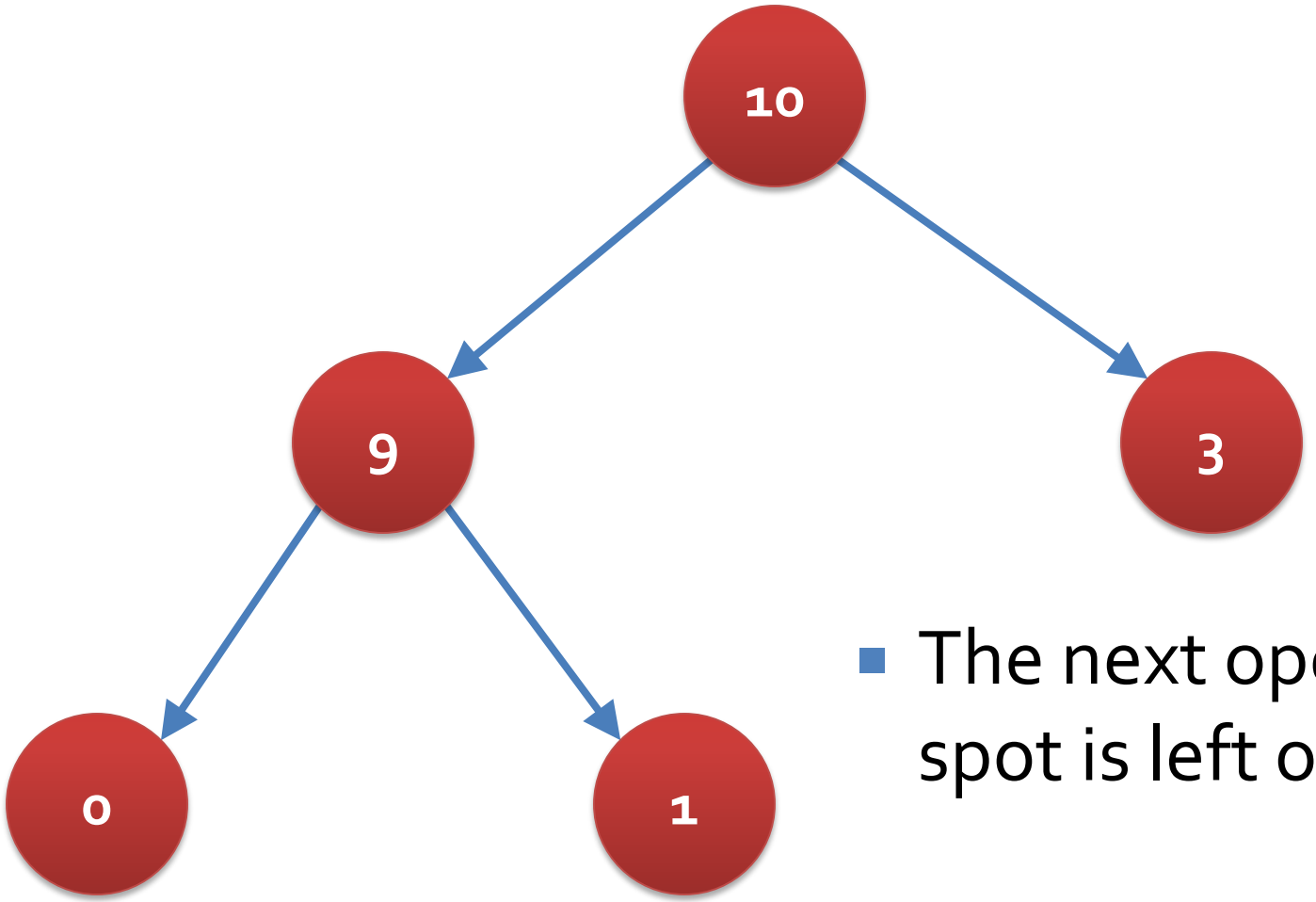
Heap example



How do you know where to add?

- We have to keep the tree complete
  - Recall that a complete binary tree is one where every level is filled, except possibly the last one, which is filled in from the left
- We always add to the next open spot in the current level
  - Or make a new level if the current level is full

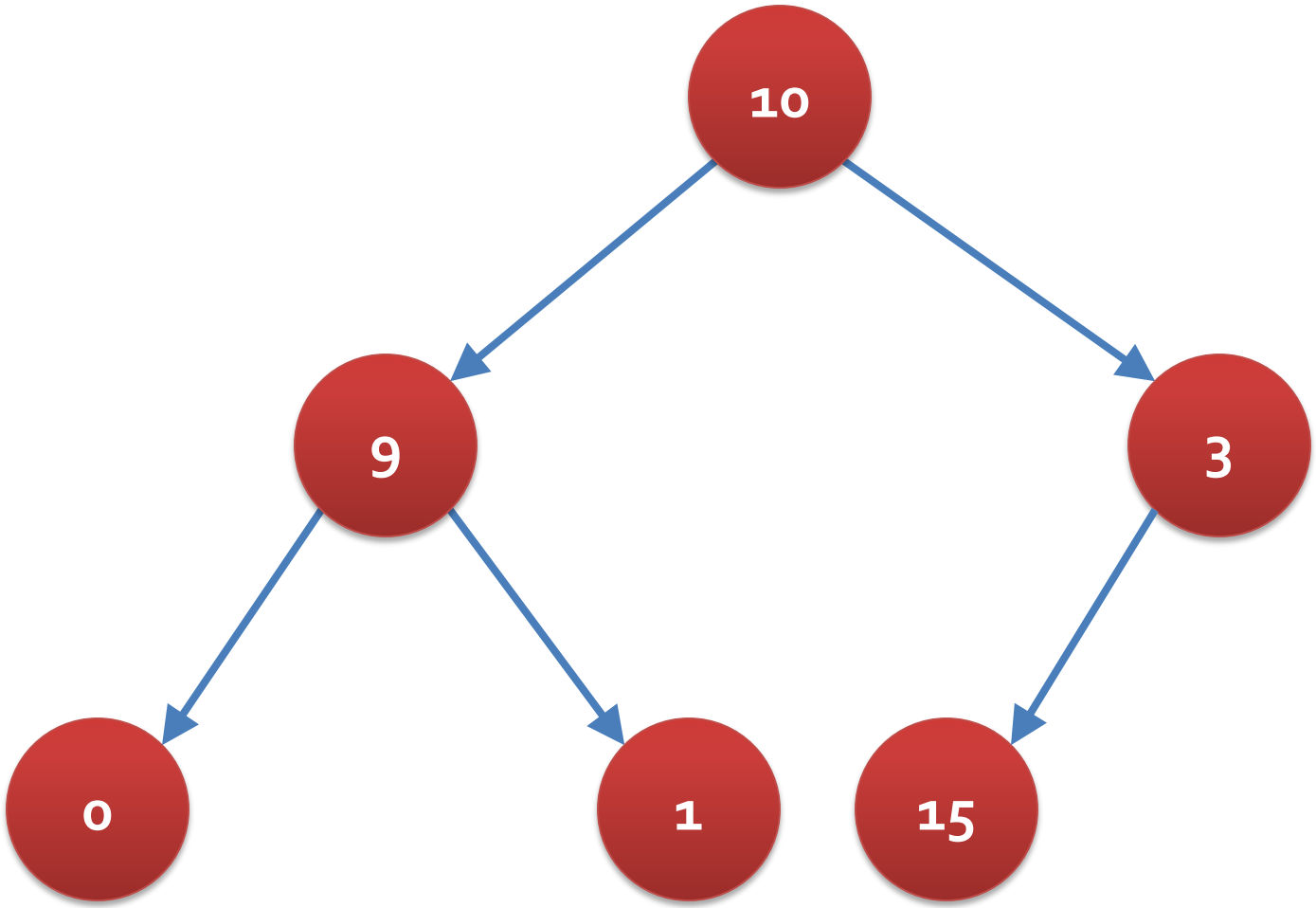
New node



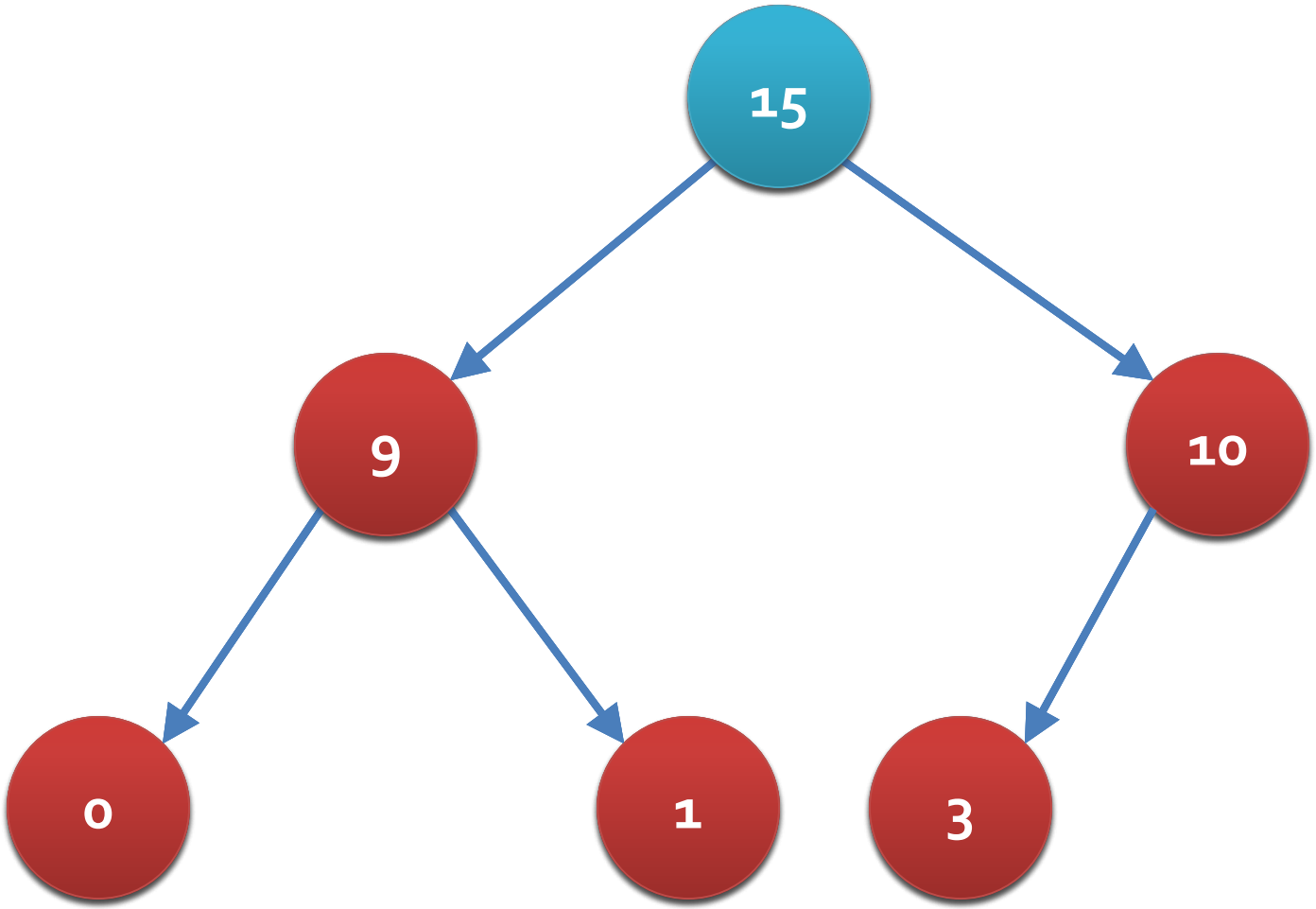
- The next open spot is left of 3

Add 15

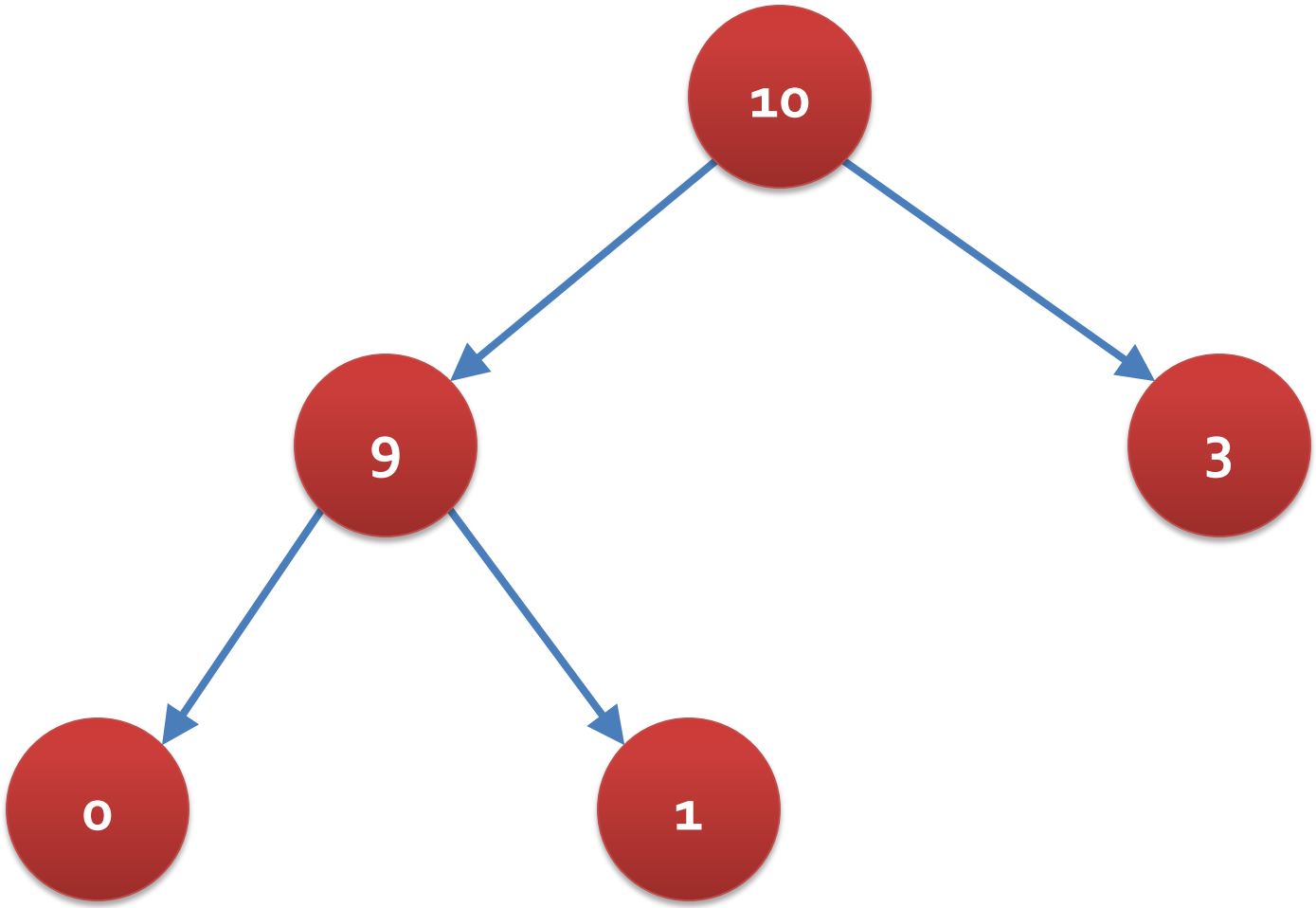
- Oh no!



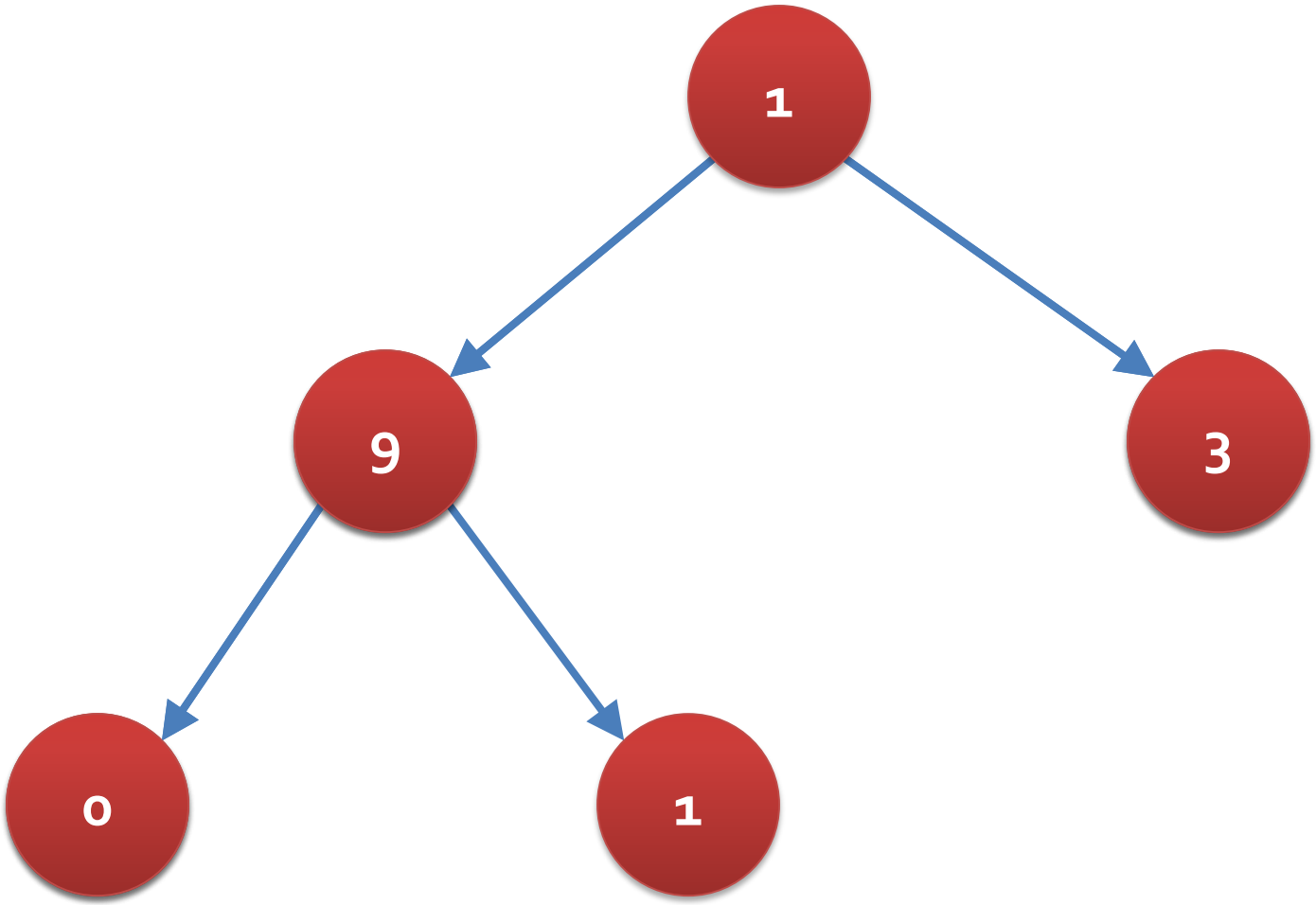
After an add, bubble up



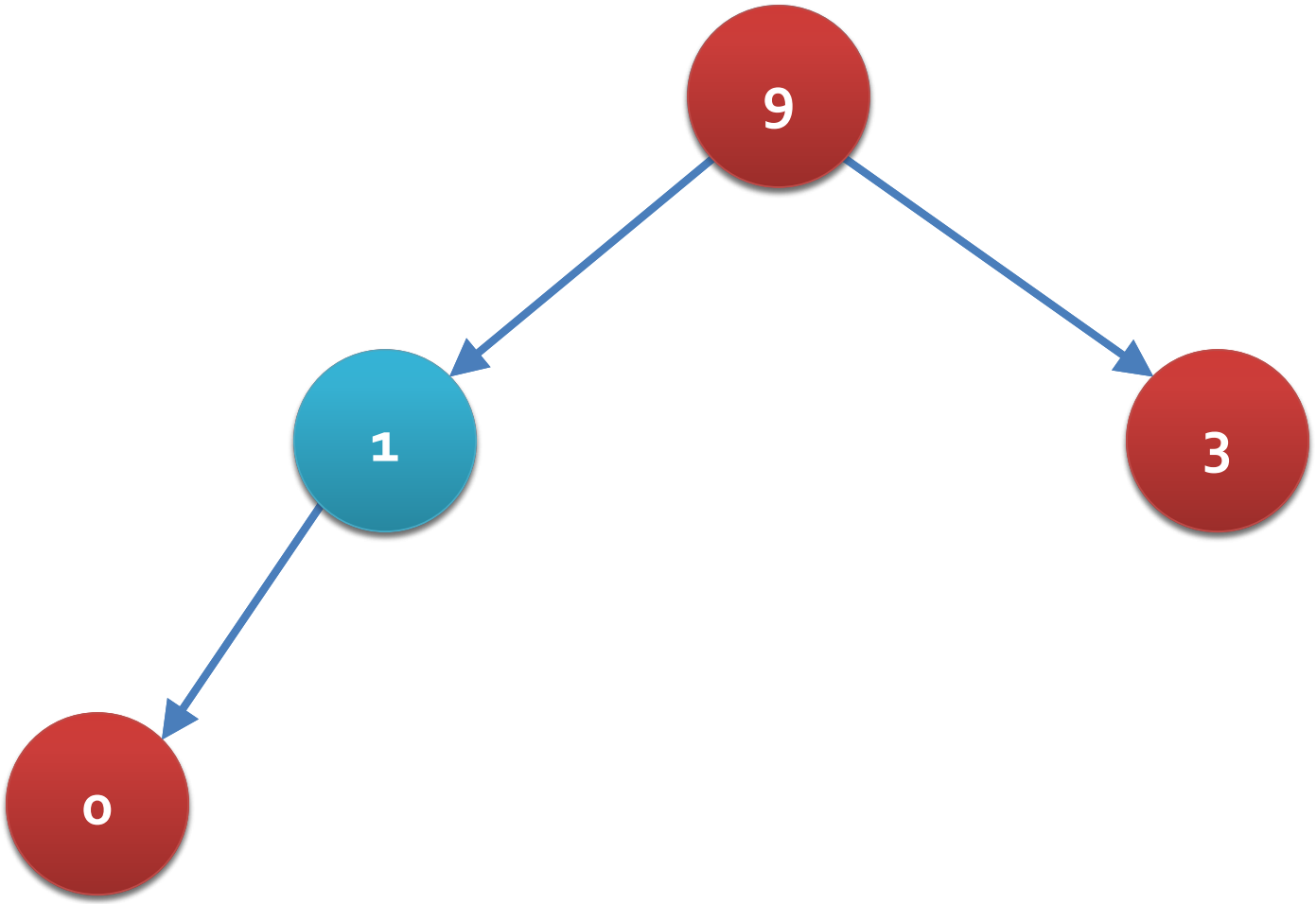
Only the root can be deleted



Replace it with the "last" node



Then, bubble down



Operations

- Heaps only have:
  - Add
  - Remove Largest
  - Get Largest
- Which cost:
  - Add: $O(\log n)$
  - Remove Largest: $O(\log n)$
  - Get Largest: $O(1)$
- Heaps are a perfect data structure for a priority queue

Upcoming

Next time…

- Finish heaps
- Heapsort
- TimSort
- Visualization of sorting

Reminders

- Work on Project 4
- Work on Assignment 7
- Keep reading Section 2.4