Week 15 - Friday

COMP 2100

Last time

- What did we talk about last time?
- Student questions
- Review up to Exam 2

Questions?

Project 4

Student Questions

Graphs

Graphs

- Edges
- Nodes
- Types
  - Undirected
  - Directed
  - Multigraphs
  - Weighted
  - Colored
  - Triangle inequality

Traversals

- Depth First Search
  - Cycle detection
  - Connectivity

- Breadth First Search

Dijkstra's Algorithm

- Start with two sets, ***S*** and ***V***:
  - ***S*** has the starting node in it
  - ***V*** has everything else

1. Set the distance to all nodes in ***V*** to ∞
2. Find the node ***u*** in ***V*** with the smallest ***d***(***u***)
3. For every neighbor ***v*** of ***u*** in V
   a) If ***d***(***v***) > ***d***(***u***) + ***d***(***u***,***v***)
   b) Set ***d***(***v***) = ***d***(***u***) + ***d***(***u***,***v***)
4. Move ***u*** from ***V*** to ***S***
5. If ***V*** is not empty, go back to Step 2

Minimum Spanning Tree (MST)

- Start with two sets, ***S*** and ***V***:
  - ***S*** has the starting node in it
  - ***V*** has everything else

1. Find the node ***u*** in ***V*** that is closest to any node in ***S***
2. Put the edge to ***u*** into the MST
3. Move ***u*** from ***V*** to ***S***
4. If ***V*** is not empty, go back to Step 1

Euler paths and tours

- An Euler path visits all edges exactly once
- An Euler tour is an Euler path that starts and ends on the same node
- If a graph only has an Euler path, exactly 2 nodes have odd degree
- If a graph has an Euler tour, all nodes have even degree
- Otherwise, the graph has no Euler tour or path

Bipartite graphs

- A bipartite graph is one whose nodes can be divided into two disjoint sets X and Y
- There can be edges between set X and set Y
- There are no edges inside set X or set Y
- A graph is bipartite if and only if it contains no odd cycles
- **If you want to show a graph is bipartite, divide it into two sets**
- **If you want to show a graph is not bipartite, show an odd cycle**

Maximum matching

- A **perfect matching** is when every node in set X and every node in set Y is matched
- It is not always possible to have a perfect matching
- We can still try to find a **maximum matching** in which as many nodes are matched up as possible

Matching algorithm

1. Come up with a legal, maximal matching
2. Take an **augmenting path** that starts at an unmatched node in X and ends at an unmatched node in Y
3. If there is such a path, switch all the edges along the path from being in the matching to being out and vice versa
4. If there is another augmenting path, go back to Step 2

NP-completeness

- A tour that visits every node exactly once is called a Hamiltonian tour
- Finding the shortest Hamiltonian tour is called the Traveling Salesman Problem
- Both problems are NP-complete (well, actually NP-hard)
- NP-complete problems are believed to have no polynomial time algorithm

B-trees

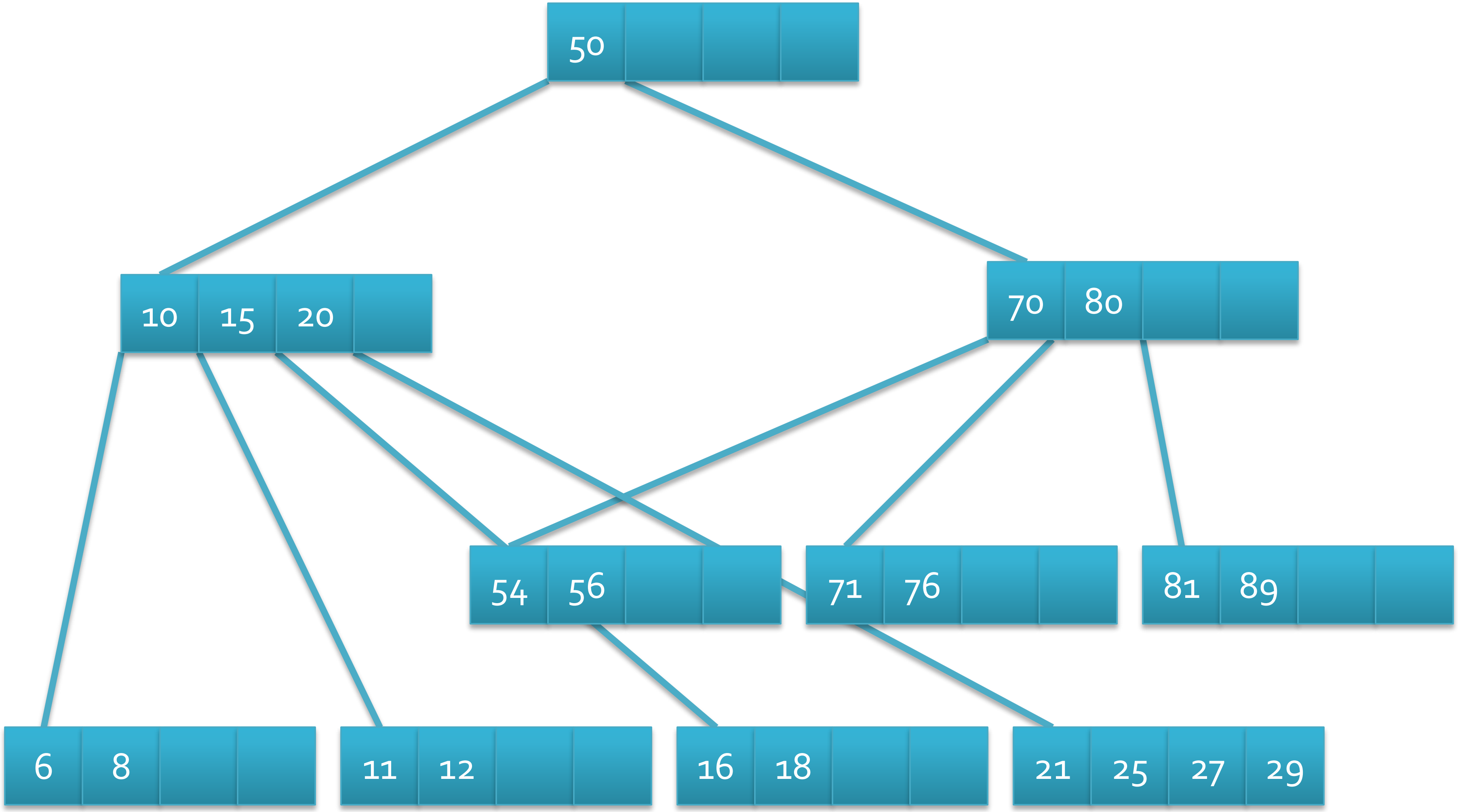
Why B-trees?

- For a tree in secondary storage
- Each read of a block from disk storage is slow
    - We want to get a whole node at once
    - Each node will give us information about lots of child nodes
    - We don't have to make many decisions to get to the node we want

B-tree definition

- A B-tree of order m has the following properties:
  1. The root has at least two subtrees unless it is a leaf
  2. Each nonroot and each nonleaf node holds k keys and $k + 1$ pointers to subtrees where $m/2 \leq k \leq m$
  3. Each leaf node holds k keys where $m/2 \leq k \leq m$
  4. **All leaves are on the same level**

B-tree of order 4



B-tree operations

- Go down the leaf where the value should go
- If the node is full
  - Break it into two half full nodes
  - Put the median value in the parent
  - If the parent is full, break it in half, etc.
- Otherwise, insert it where it goes
- Deletes are the opposite process
  - When a node goes below half full, merge it with its neighbor

Variations

- B*-tree
  - Shares values between two neighboring leaves until they are both full
  - Then, splits two nodes into three
  - Maintains better space utilization
- B^+-tree
  - Keeps (copies of) all keys in the leaves
  - Has a linked list that joins all leaves together for fast sequential access

Maximum flow

- A common flow problem on flow networks is to find the **maximum flow**
- A maximum flow is a non-negative amount of flow on each edge such that:
  - The maximum amount of flow gets from ***s*** to ***t***
  - No edge has more flow than its capacity
  - The flow going into every node (except ***s*** and ***t***) is equal to the flow going out

Augmenting path

- When we were talking about matching, we mentioned **augmenting paths**
- Augmenting paths in flows are a little different
- A flow augmenting path:
    - Starts at ***s*** and ends at ***t***
    - May cross some edges in the direction of the edge (forward edges)
    - May cross some edges in the opposite direction (backwards edges)
    - Increases the flow by the minimum of the unused capacity in the forward edges or the maximum of the flow in the backwards edges

Sorting

Insertion sort

- We do ***n*** rounds
    - For round ***i***, assume that the elements 0 through ***i*** – 1 are sorted
    - Take element ***i*** and move it up the list of already sorted elements until you find the spot where it fits
- O(n^2) in the worst case
- O(n) in the best case
- Adaptive and the fastest way to sort 10 numbers or fewer

Merge sort algorithm

- Take a list of numbers, and divide it in half, then, recursively:
    - Merge sort each half
    - After each half has been sorted, merge them together in order
- O(***n*** log ***n***) best and worst case time
- Not in-place

Heap sort

- Make the array have the heap property:
    1. Let ***i*** be the index of the parent of the last two nodes
    2. Bubble the value at index ***i*** down if needed
    3. Decrement ***i***
    4. If ***i*** is not less than zero, go to Step 2
1. Let ***pos*** be the index of the last element in the array
2. Swap index 0 with index ***pos***
3. Bubble down index 0
4. Decrement ***pos***
5. If ***pos*** is greater than zero, go to Step 2

- O(***n*** log ***n***) best and worst case time
- In-place

Quicksort

1. Pick a pivot
2. Partition the array into a left half smaller than the pivot and a right half bigger than the pivot
3. Recursively, quicksort the left part (items smaller than the pivot)
4. Recursively quicksort the right part (items larger than the pivot

- O($\boldsymbol{n}^2$) worst case time but O($\boldsymbol{n}$ log $\boldsymbol{n}$) best case and average case
- In-place

Counting sort

- 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
- Runs in O(***n*** **+** |Values|) time

Radix sort

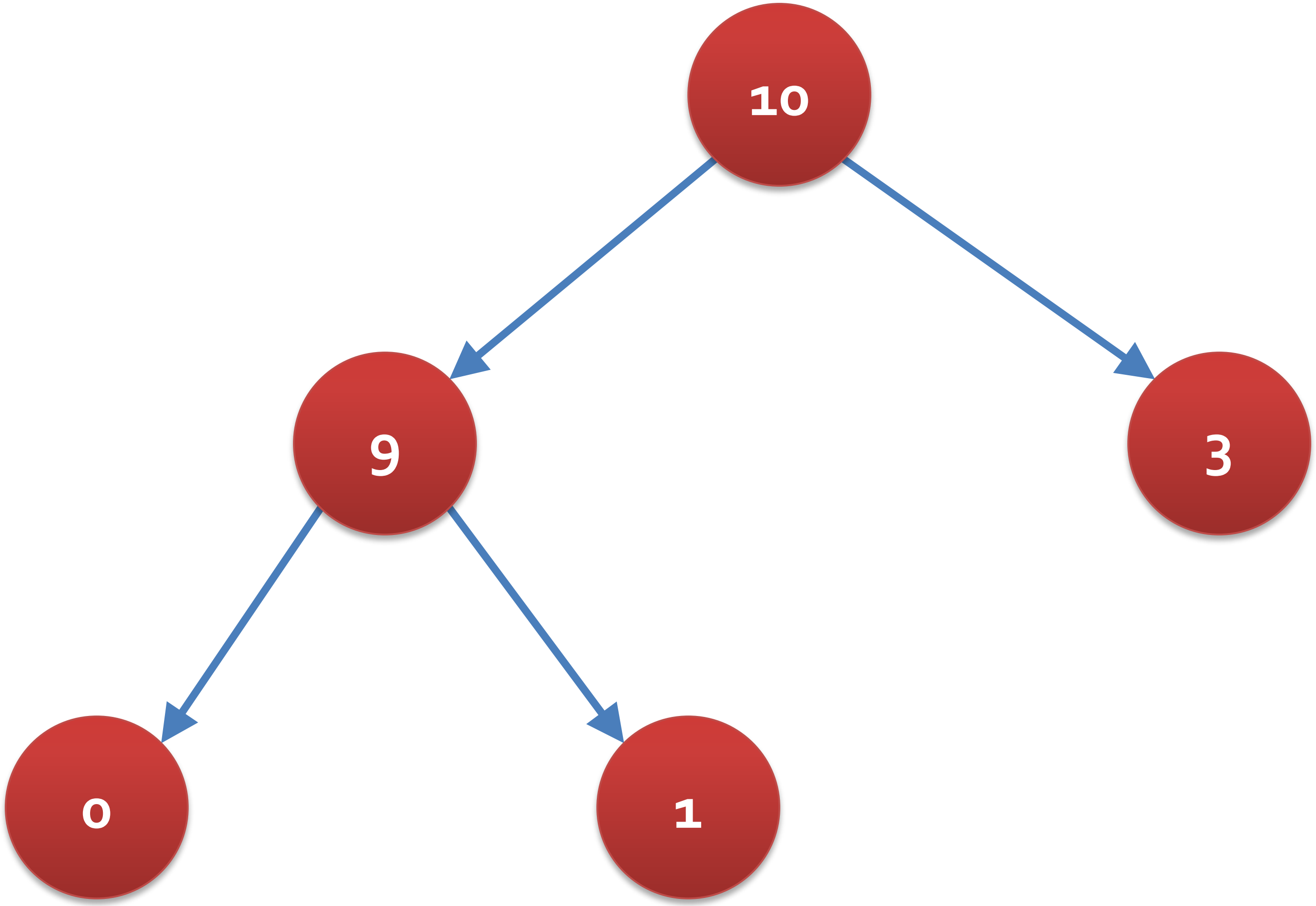
- We can "generalize" counting sort somewhat
- Instead of looking at the value as a whole, we can look at individual digits (or even individual characters)
- For decimal numbers, we would only need 10 buckets (0 – 9)
- First, we bucket everything based on the least significant digits, then the second least, etc.
- Runs in O(***nk***) time, where ***k*** is the number of digits we have to examine

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

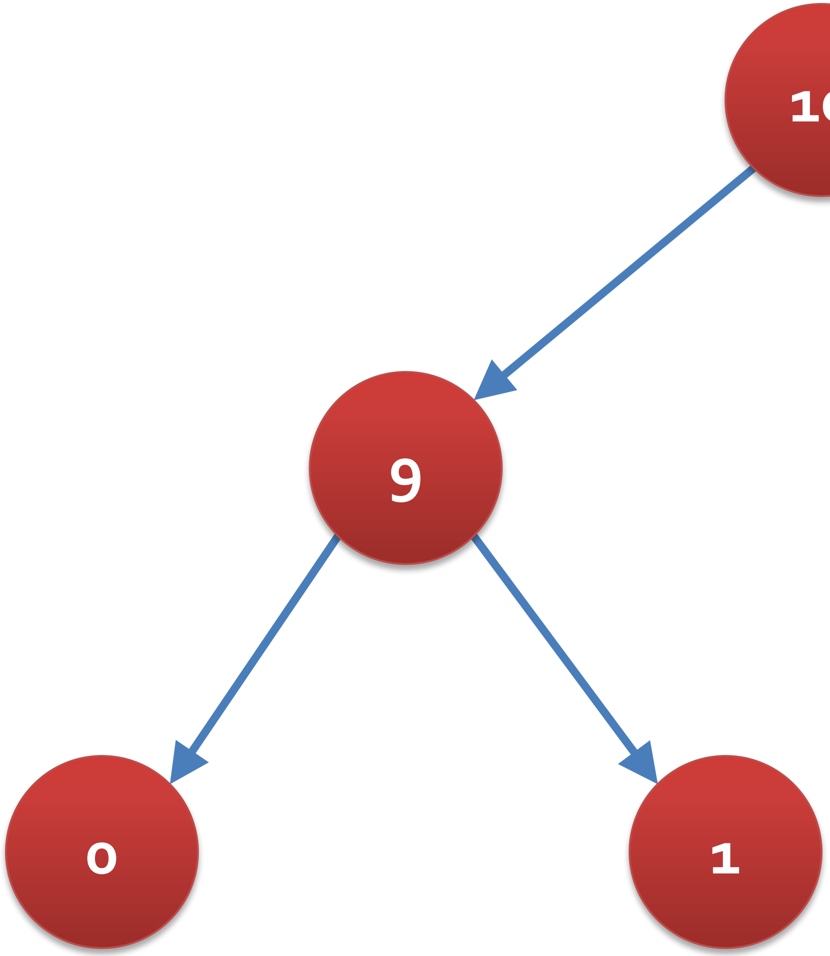
Heap example



How do you know where to add?

- Always in the first open spot on the bottom level of the tree, moving from left to right
- If the bottom level of the tree is full, start a new level

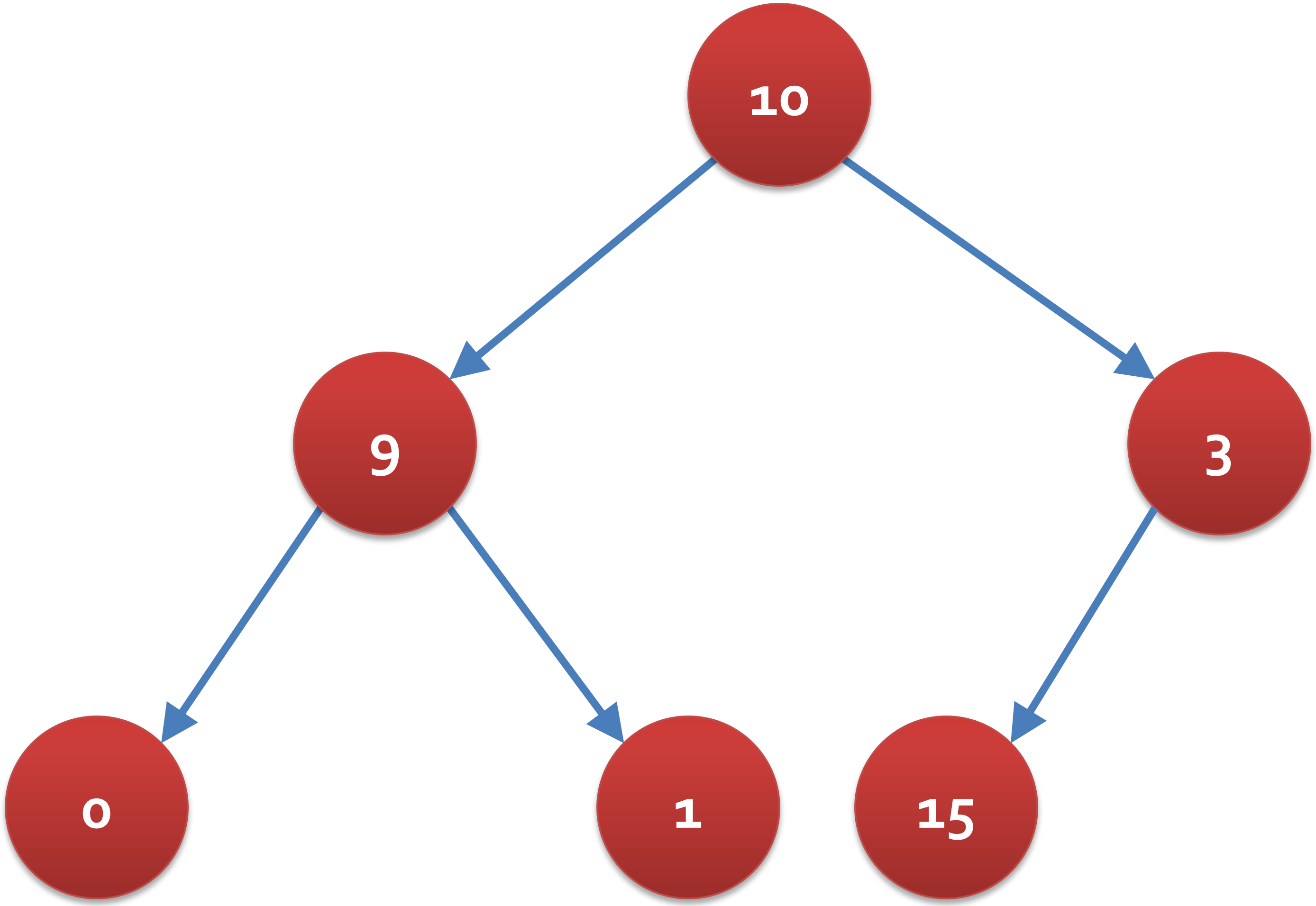
New node



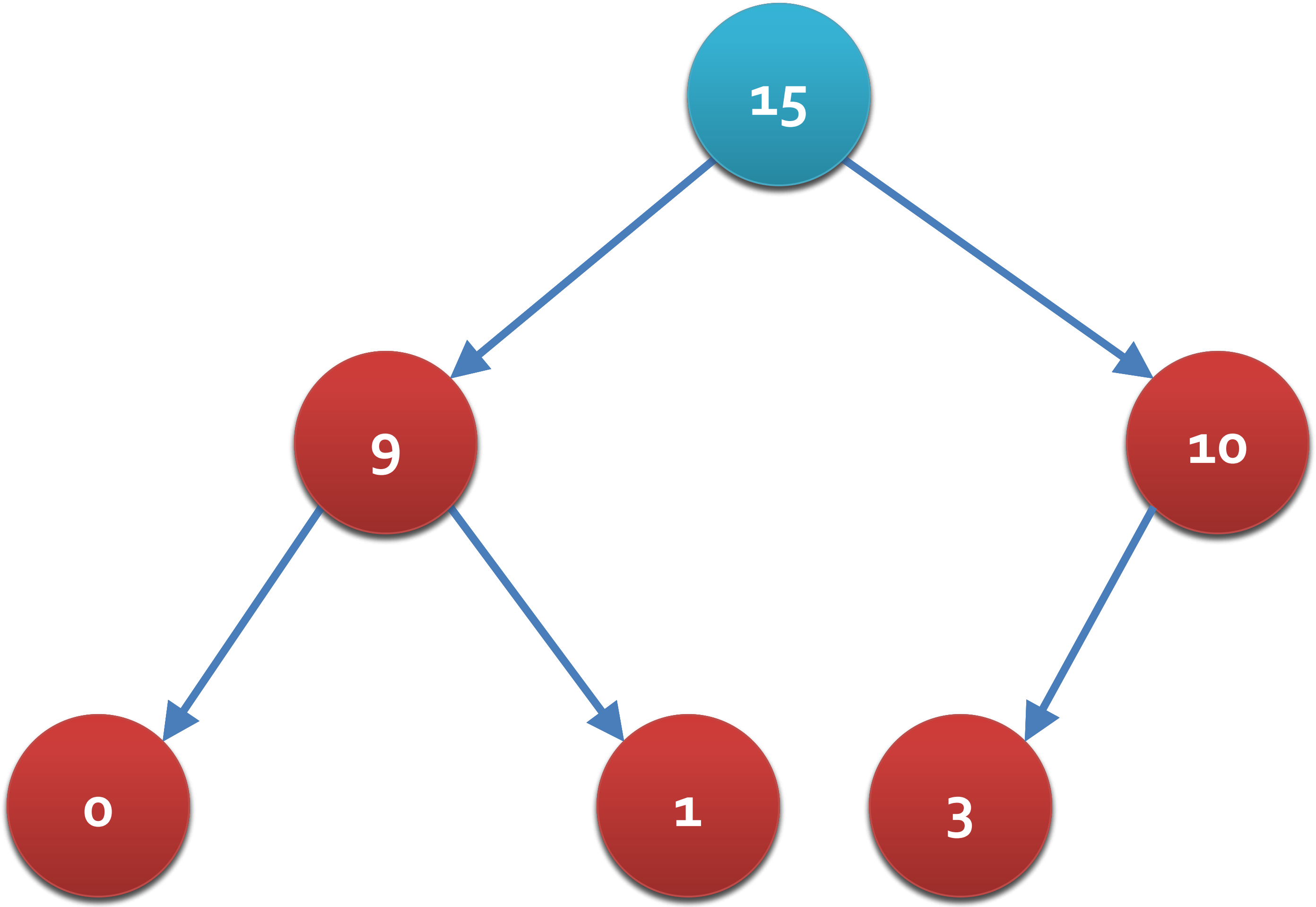
- The next open spot is left of 3

Add 15

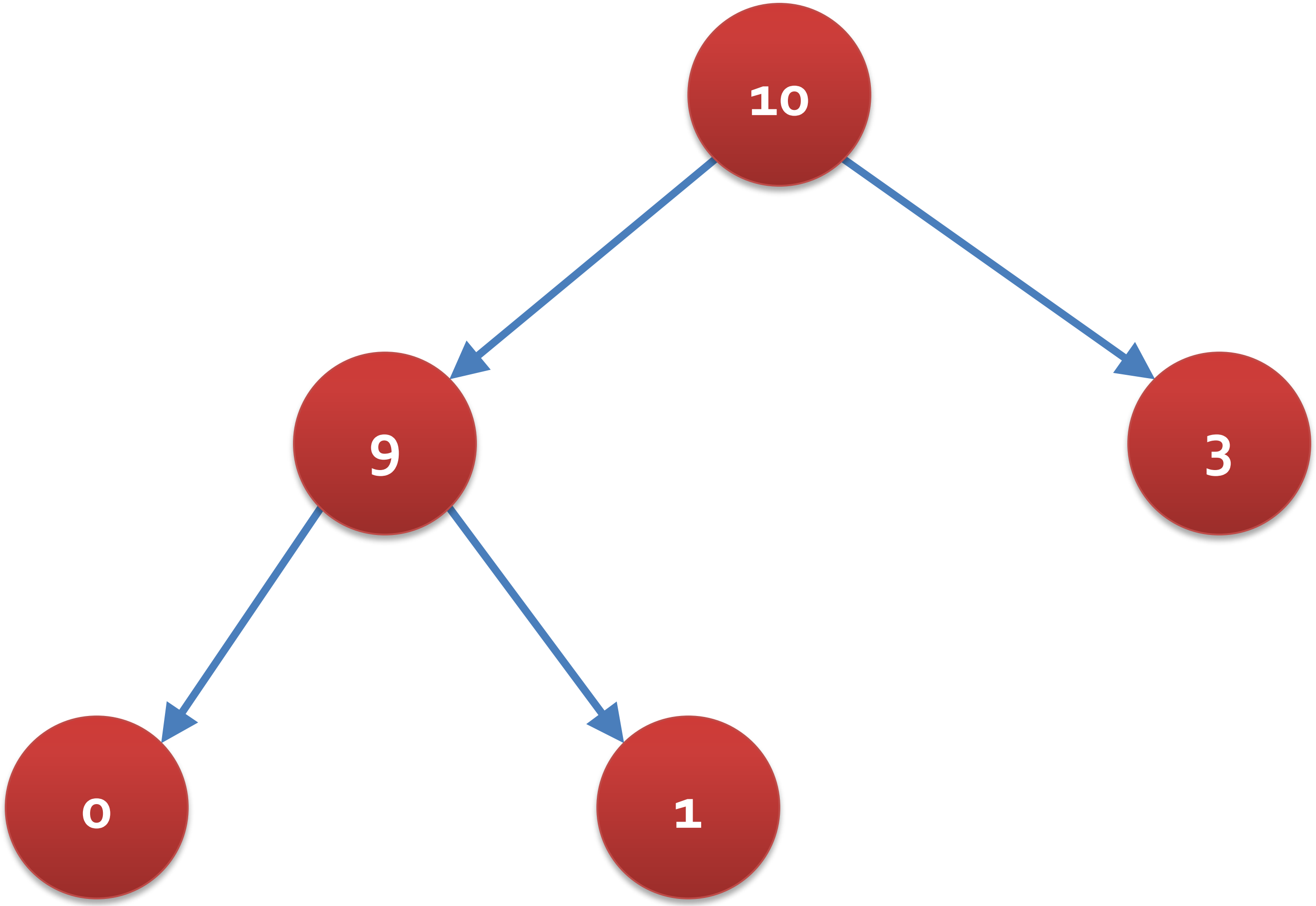
- Oh no!



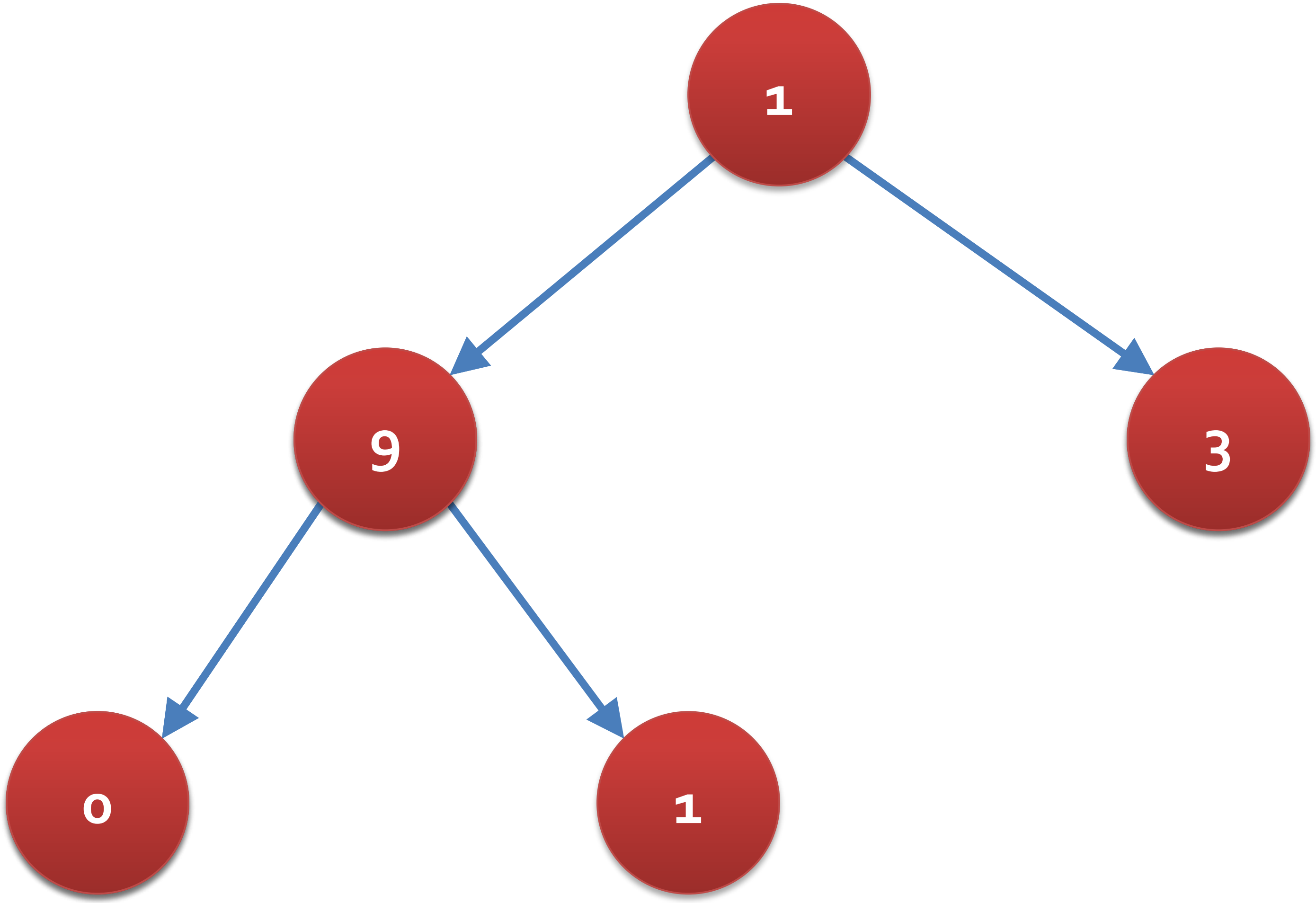
After an add, bubble up



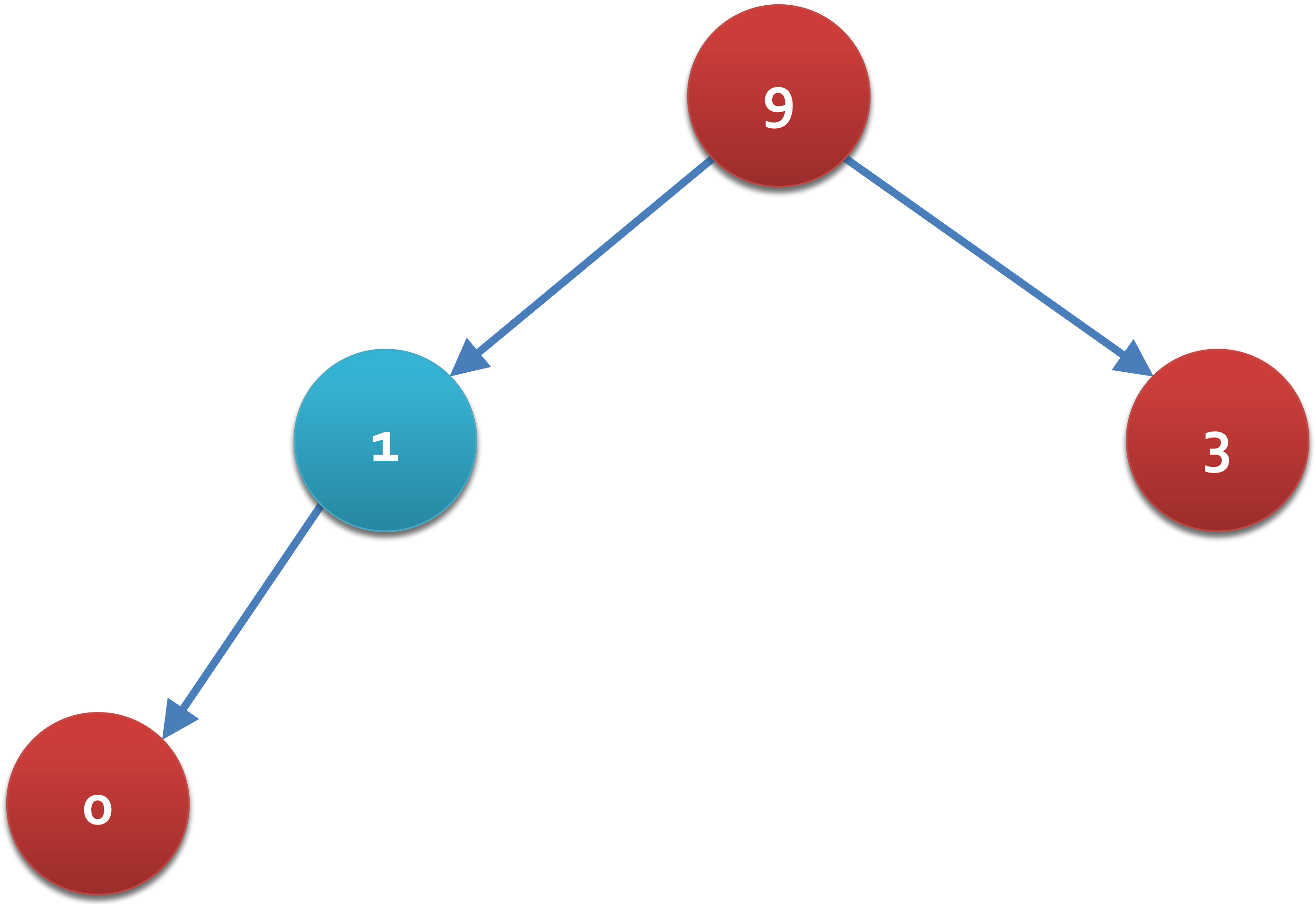
Only the root can be deleted



Replace it with the “last” node



Then, bubble down

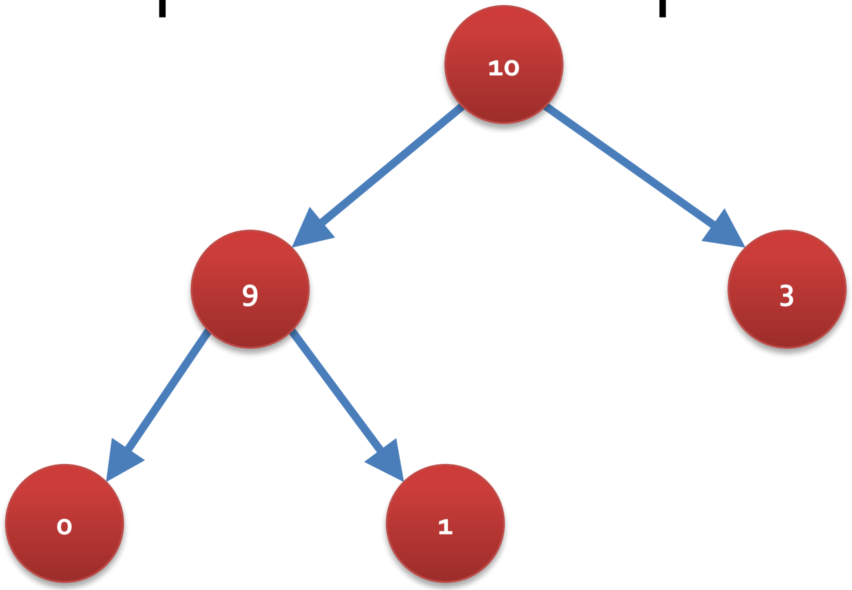


Operations

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

Array view

- We can implement a heap with a (dynamic) array



10	9	3	0	1

- The left child of element i is at $2i + 1$
- The right child of element i is at $2i + 2$

Tries

Storing strings (of anything)

- We can use a (non-binary) tree to record strings implicitly where each link corresponds to the next letter in the string
- Let's store:
  - 10
  - 102
  - 103
  - 10224
  - 305
  - 305678
  - 09

Upcoming

Next time…

- There is no next time!

Reminders

- Fill out course evaluations!
- Finish Project 4
  - Due tonight!
- Study for final exam
  - Friday, 12/13/2024 from 10:15 a.m. - 12:15 p.m.