CSC148 – Introduction to Computer Science

Lecture 9: Priority Queues

Sean Henderson

Priority Queue ADT

- insert(x)
  - Add x to the priority queue
- min()
  - Return the smallest element in the queue
- extract_min()
  - Return and remove the smallest element in the queue
- size()
  - How many elements are in the queue

Implementation: A List

- insert(x): my_list.append(x)
- min(): for loop to find the smallest element
- extract_min(): min(), plus remove from the list
- size(): len(my_list)
- What are the runtimes of these algorithms?

Implementation: A List

- insert(x): my_list.append(x)
- min(): for loop to find the smallest element
- extract_min(): min(), plus remove from the list
- size(): len(my_list)
- What are the runtimes of these algorithms?
- In order: O(1), O(n), O(n), O(1)
- Seems like this can be improved upon…

Implementation: A BST

- insert(x): insert x into the BST
- min(): return smallest element in tree (how?)
- extract_min(): remove the smallest element in the tree (how?)
- size(): return how many elements are in the tree
- Runtime?

Implementation: A BST

- insert, extract_min will both be O(height of tree); this is O(log n) with high probability, but in the worst case can be O(n)
- size() can be calculated in time O(1), if done correctly (i.e. using modification from A2)
- min() can also be done in constant time if we keep track of it every time we insert or extract (how?)

Priority Queue Goal

- Using a Balanced BST (guaranteed O(log n) height) we can do insert and extract_min in O(log n) worst case time
- This is a fair amount of work; would like to do something simpler, yet maintain a good runtime

Binary Heaps

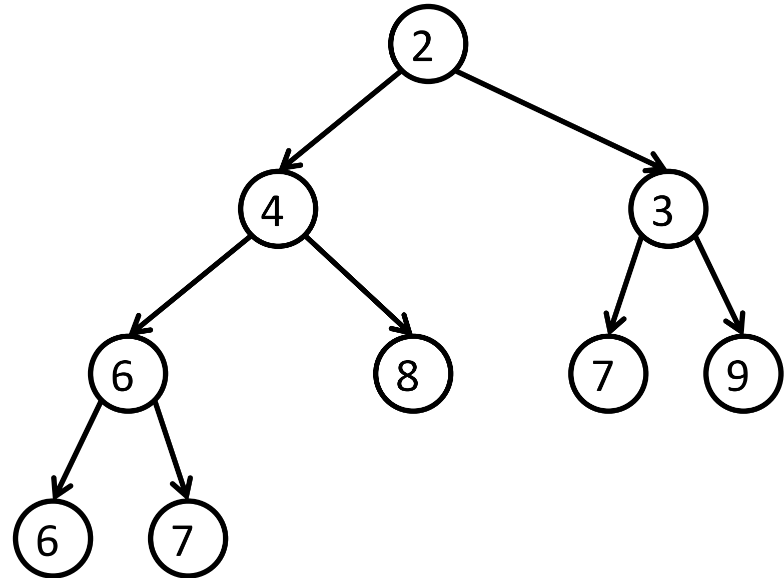
- A binary heap is a “complete” binary tree
  - Complete: each node was inserted to have the lowest possible depth; if multiple places to do this, then the leftmost position is chosen (we’ll see an example shortly…)
- Satisfies the heap property: the value stored on every node is less than (or equal to) the value stored on its children

Binary Heaps

- Easily proven fact: the value at the root must be the smallest value in the heap (why?)
- Easily proven fact: the height of a binary heap is O(log n) (it is by definition the smallest possible tree with n nodes, structurally)
- Clearly we can implement min and size rather simply
- Problem: what about extract_min and insert?

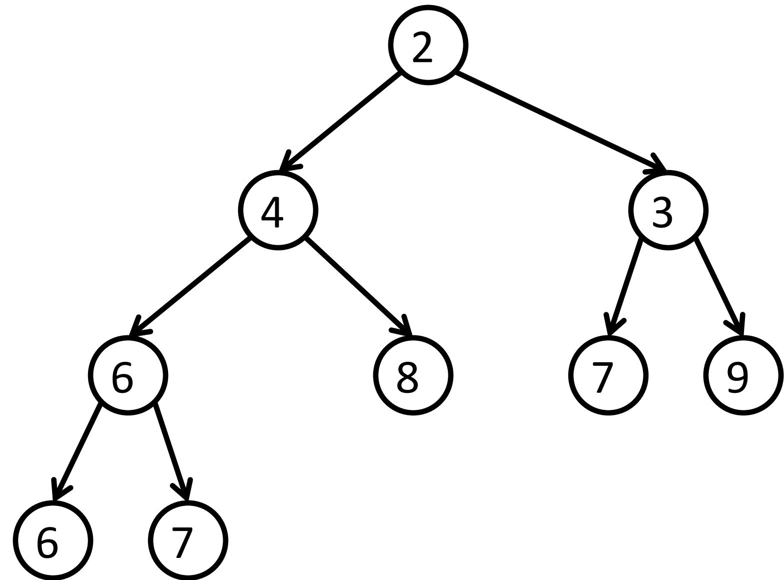
Binary Heap Example

- The figure on the right is an example of a binary heap with 9 nodes
- Notice: every row is “full” except the last one
- Min element at the root
- Each node <= children



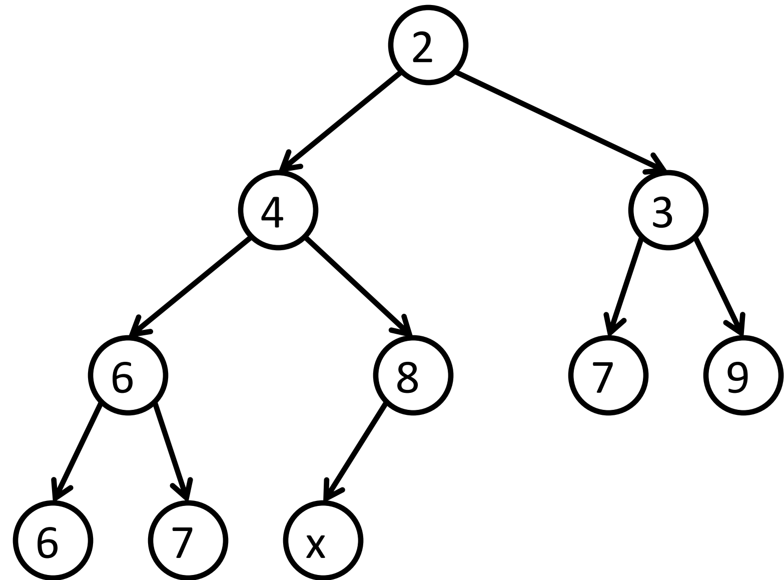
Insertion

- Say we want to insert the value x
- We will start by inserting it at the bottom of the tree as a leaf (such that the tree is still complete)



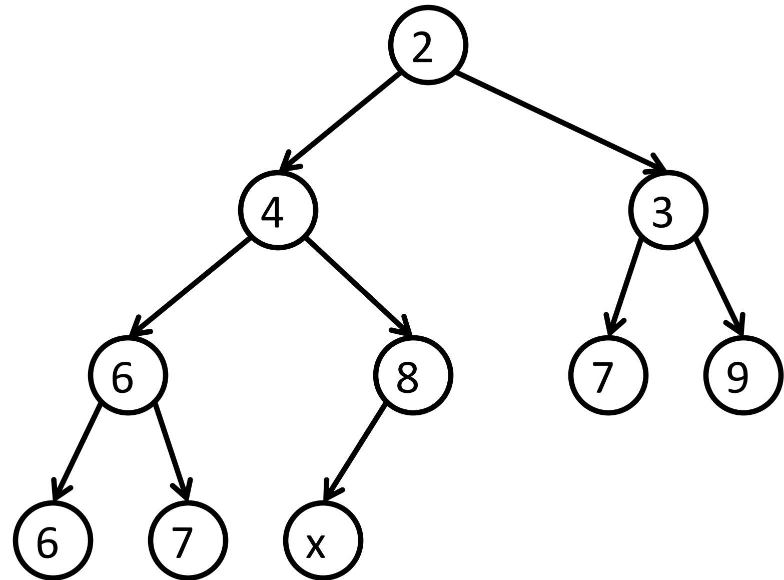
Insertion

- Say we want to insert the value x
- We will start by inserting it at the bottom of the tree as a leaf (such that the tree is still complete)



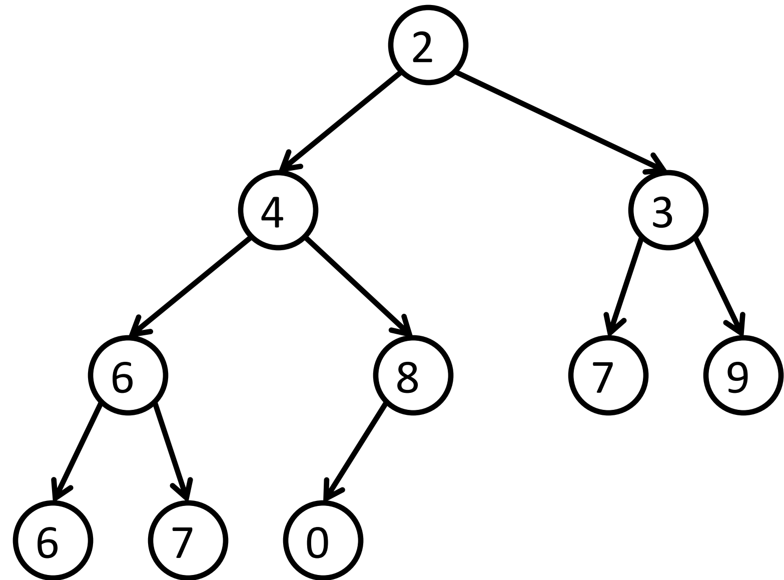
Insertion

- What if x >= 8?
- Then this is a heap, and we're done!
- Let's say it is not; choose x = 0



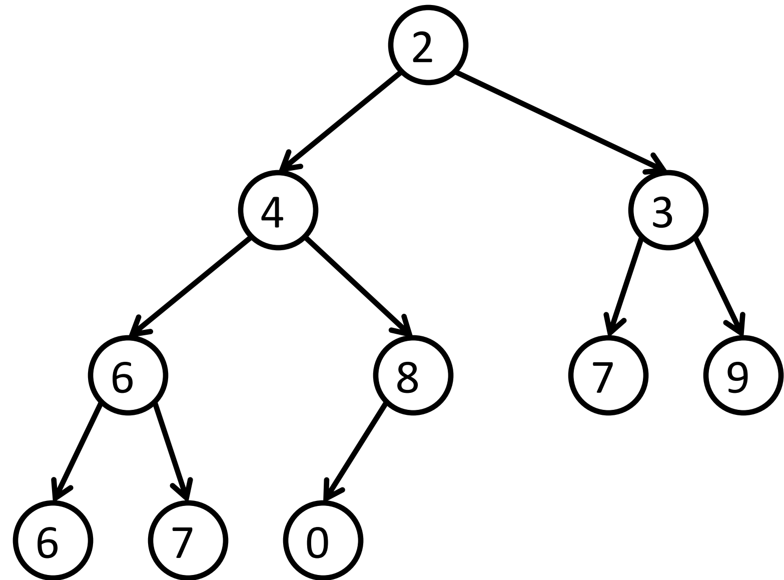
Insertion

- What if x >= 8?
- Then this is a heap, and we're done!
- Let's say it is not; choose x = 0
- Now we are violating the heap property



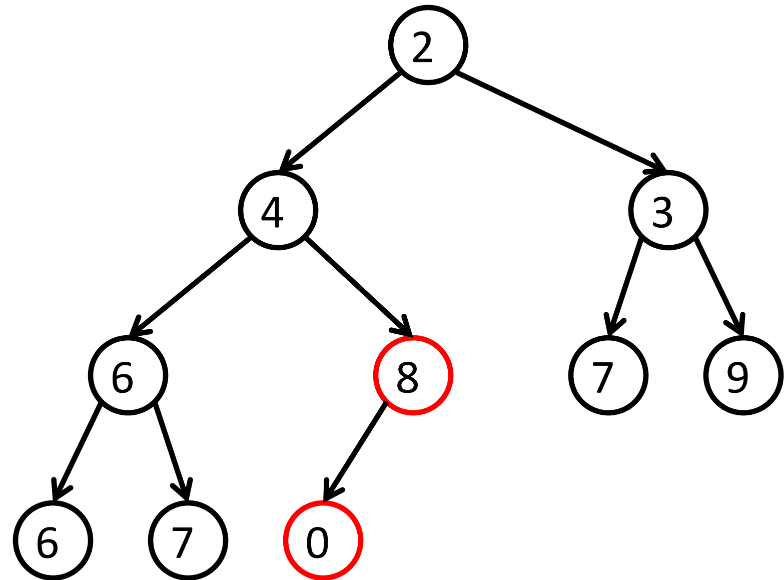
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



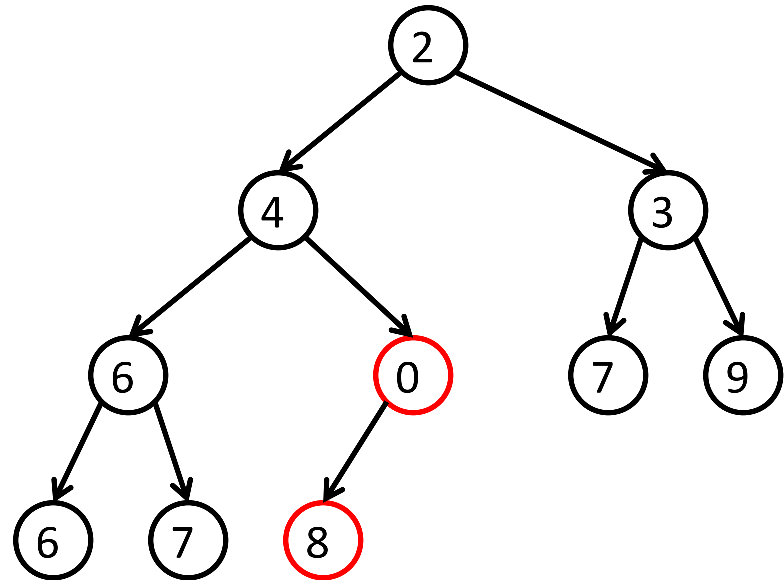
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



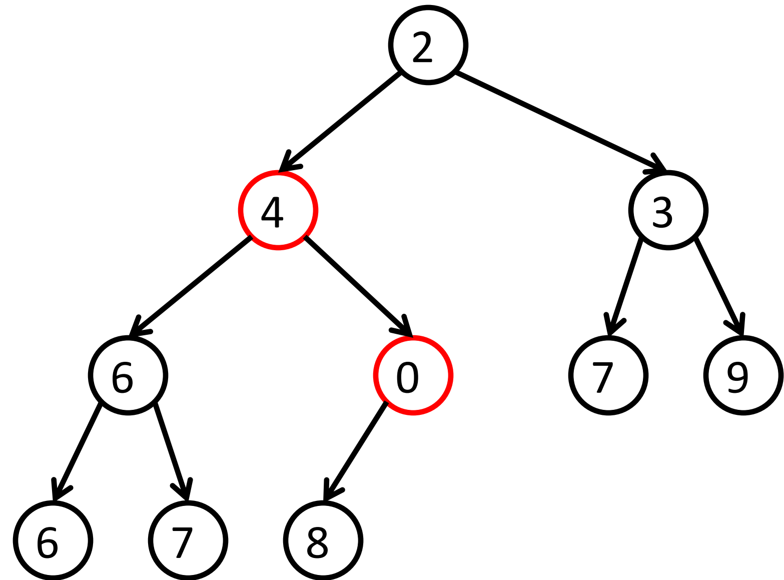
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



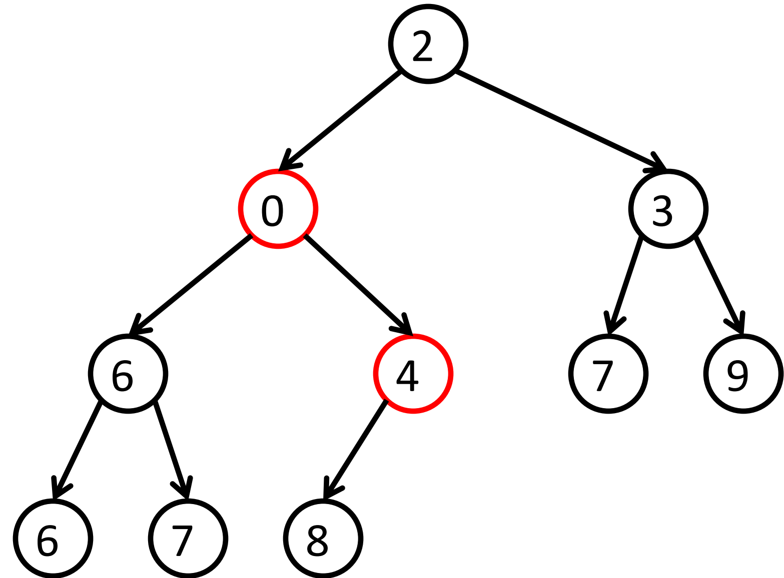
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



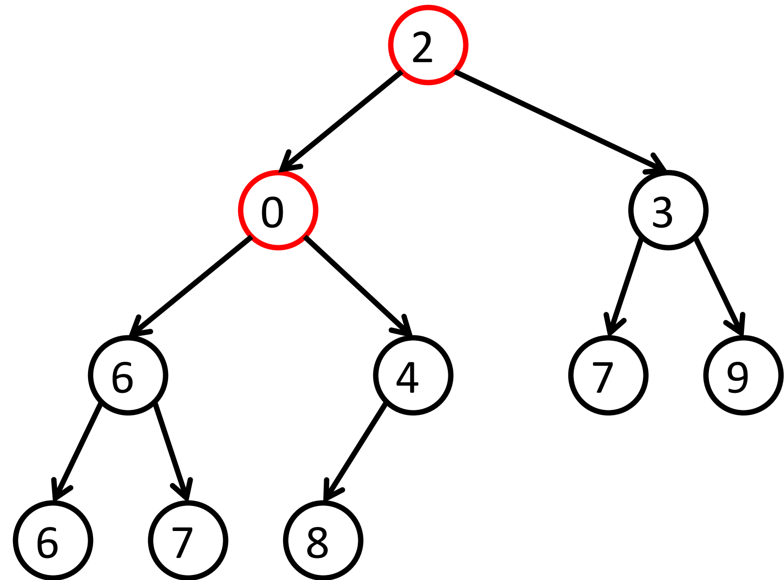
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



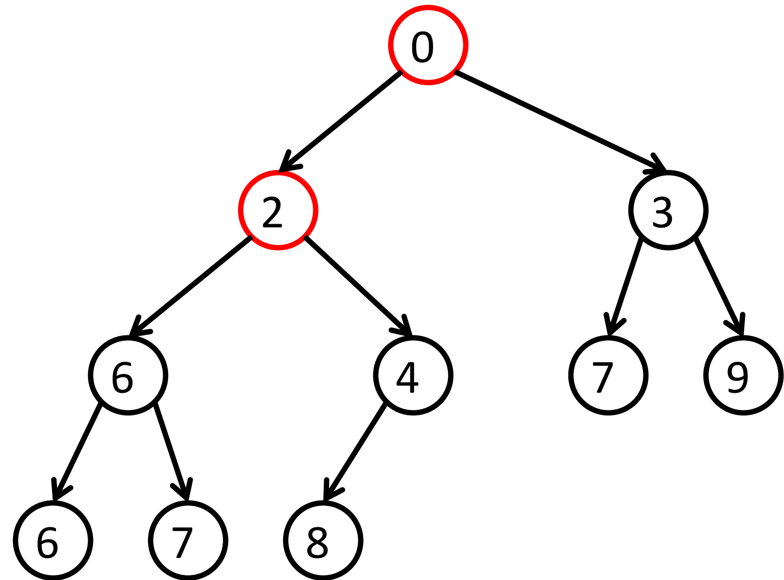
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



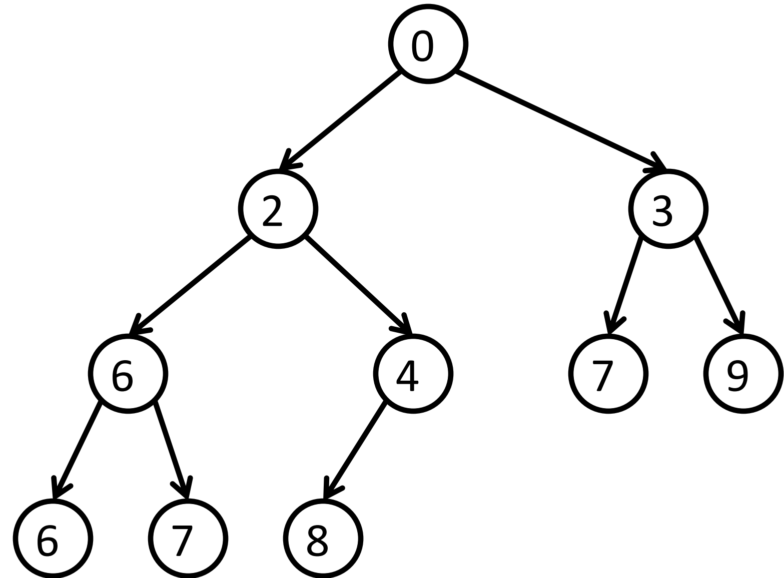
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



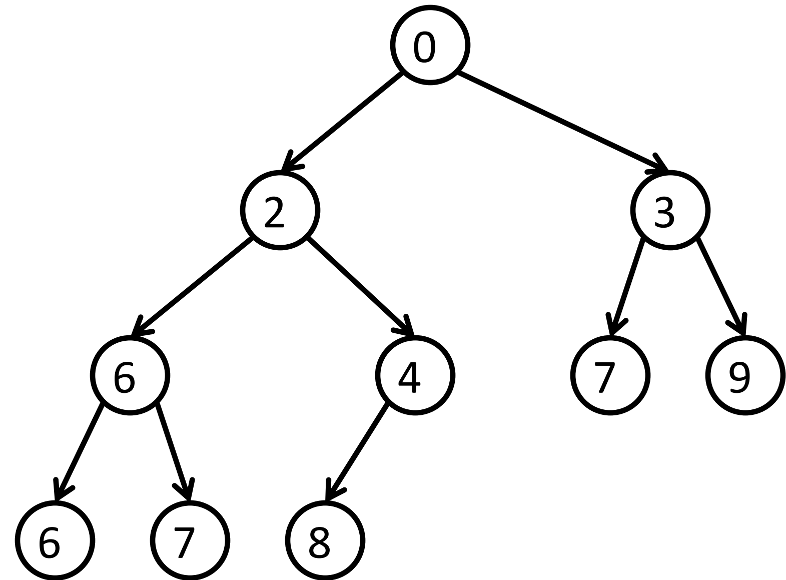
Insertion

- Algorithm: keep swapping it with its parent, until it is either the root, or greater than (or equal to) its parent



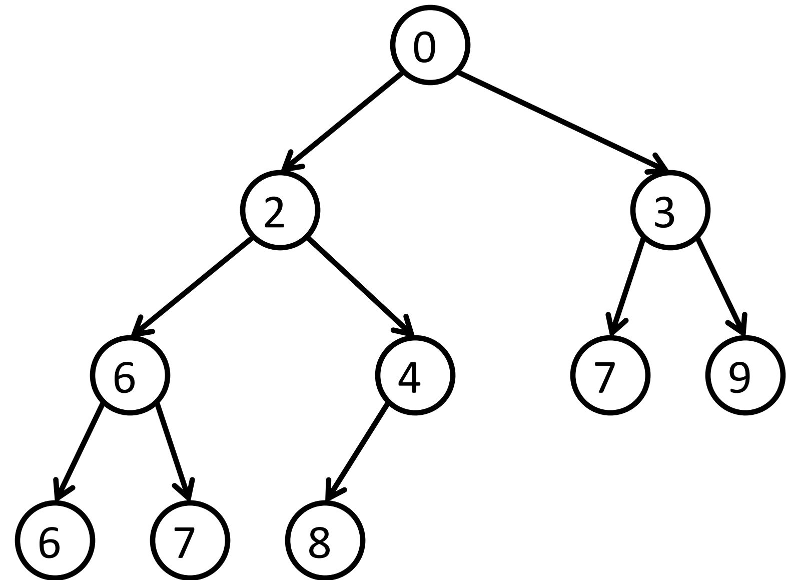
Insertion

- Why is this correct?
- Let’s say x is getting replaced by the smaller value at x.right
- We know x.left >= x, and that x > x.right, so it follows that x.right < x.left
- So the heap property is maintained after the swap



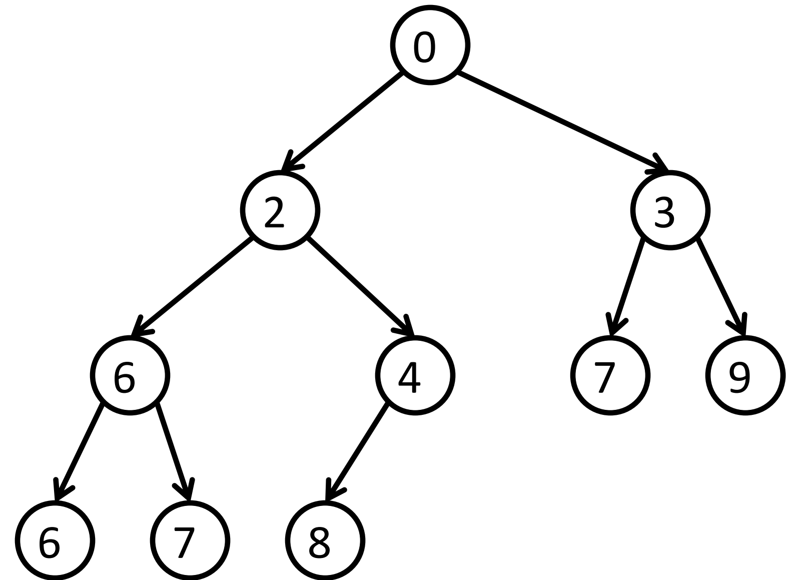
Insertion

- Can formally prove this algorithm is correct using induction; if x is the node we are swapping up the heap, than can show that at each step the subtree rooted at x is a heap



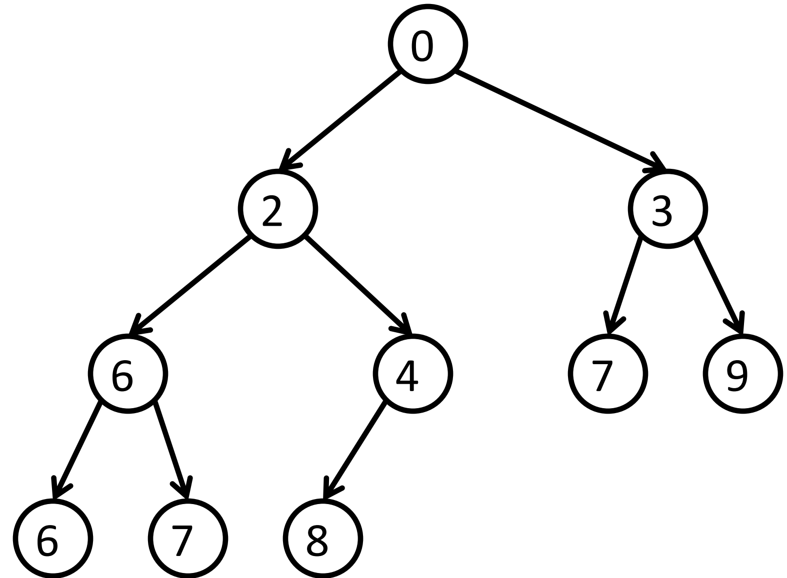
Extract Min

- Now that we can insert values, we would like to be able to remove the minimum element
- Can use a similar idea; except instead of swapping up the tree, we swap down…



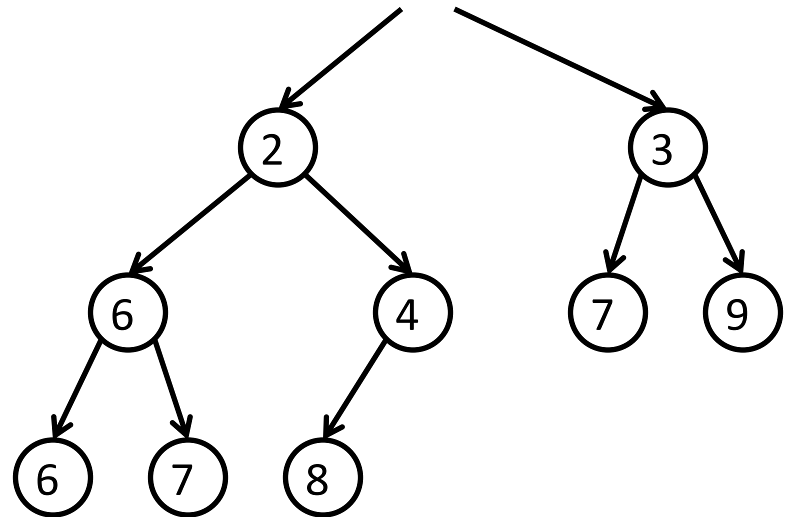
Extract Min

- Step 1: Remove the root



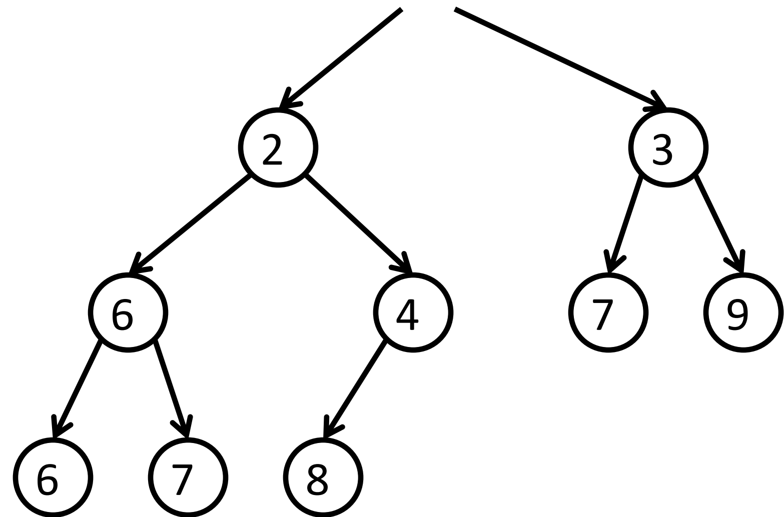
Extract Min

- Step 1: Remove the root



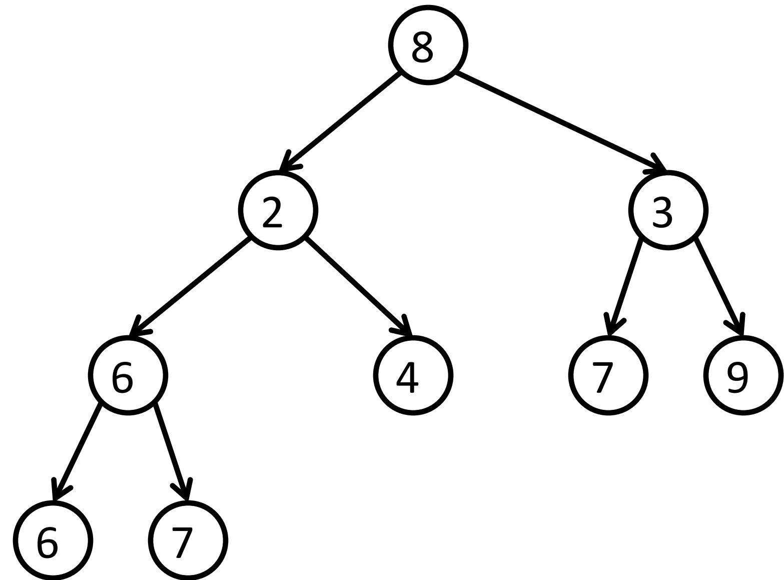
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row



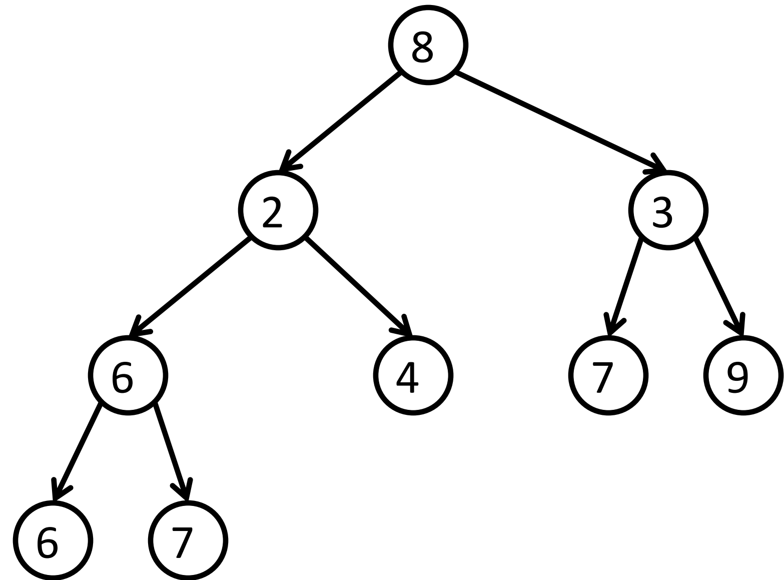
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row



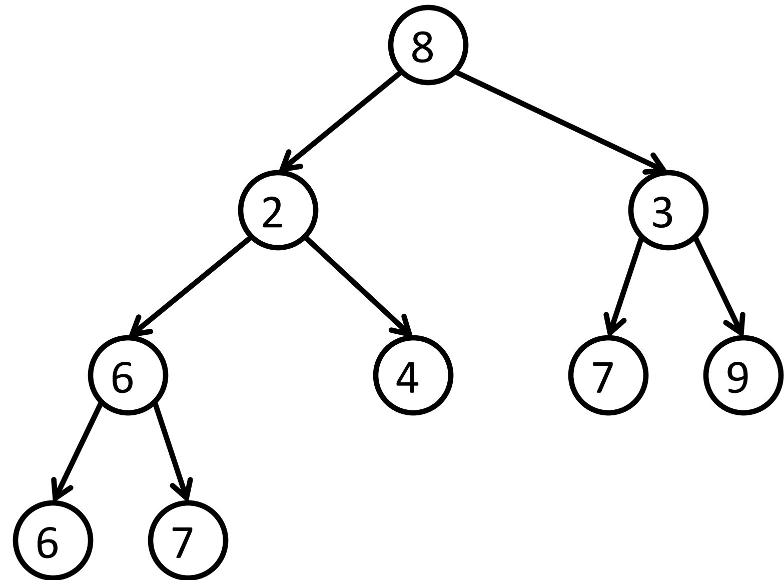
Extract Min

- Now at this point, the heap property is being violated by the root
- Want to fix this by swapping it with one of its children
- Which one?



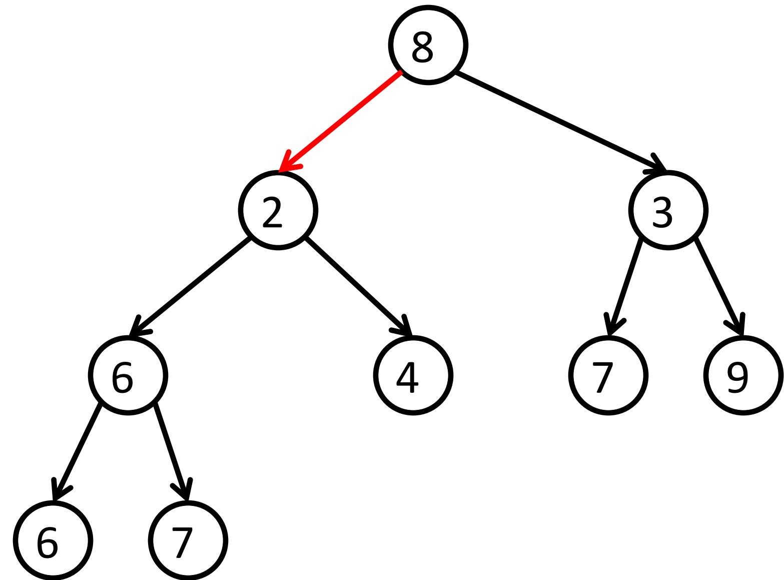
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row
- Step 3: Repeatedly swap the root node with its smallest child until the heap property is no longer violated



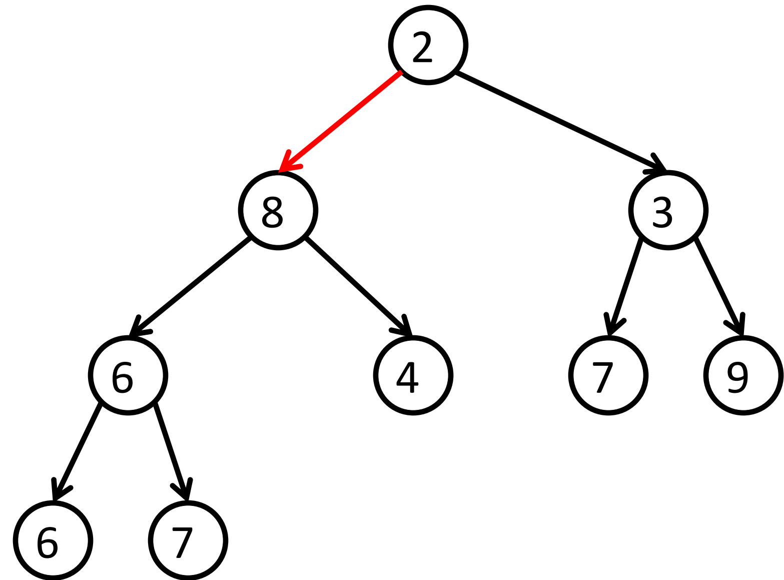
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row
- Step 3: Repeatedly swap the root node with its smallest child until the heap property is no longer violated



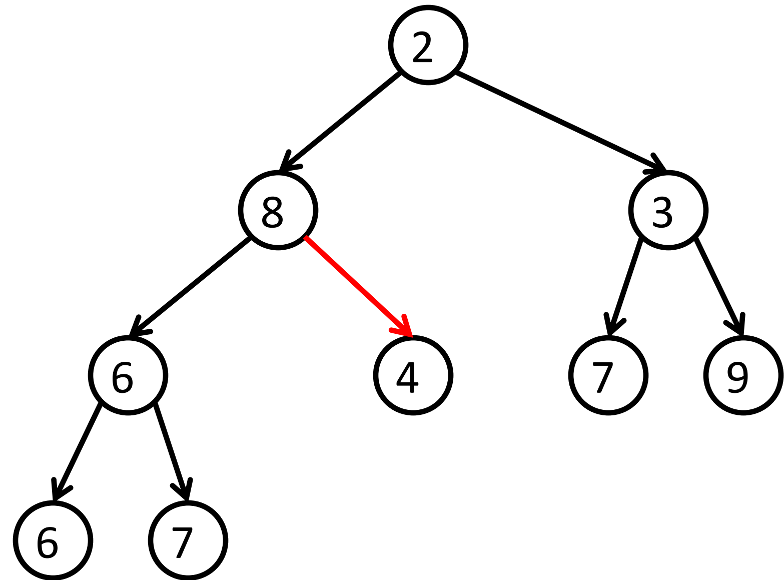
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row
- Step 3: Repeatedly swap the root node with its smallest child until the heap property is no longer violated



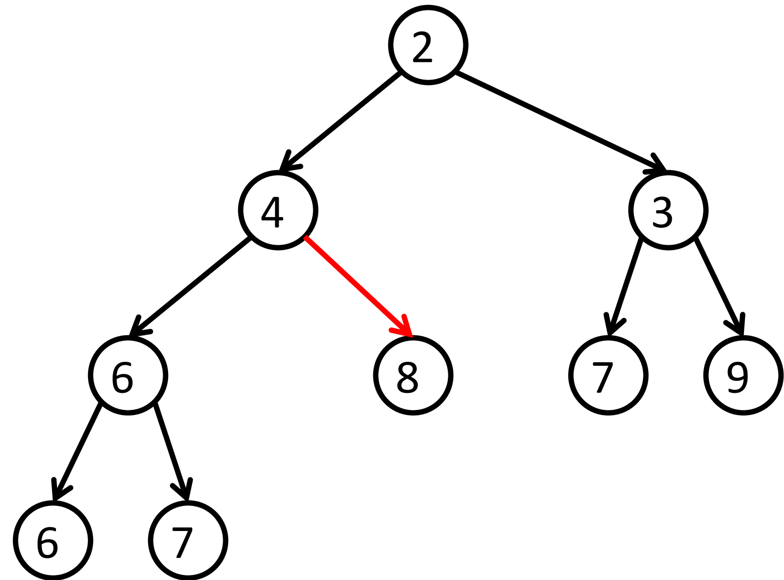
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row
- Step 3: Repeatedly swap the root node with its smallest child until the heap property is no longer violated



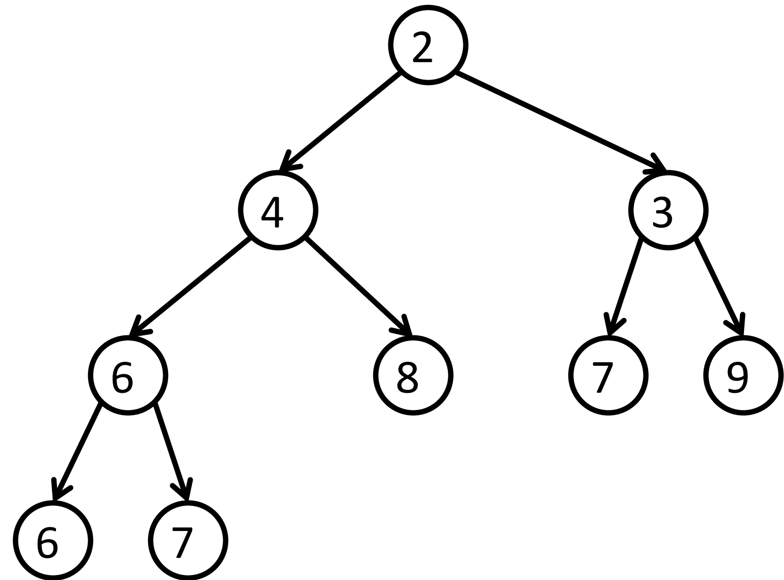
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row
- Step 3: Repeatedly swap the root node with its smallest child until the heap property is no longer violated



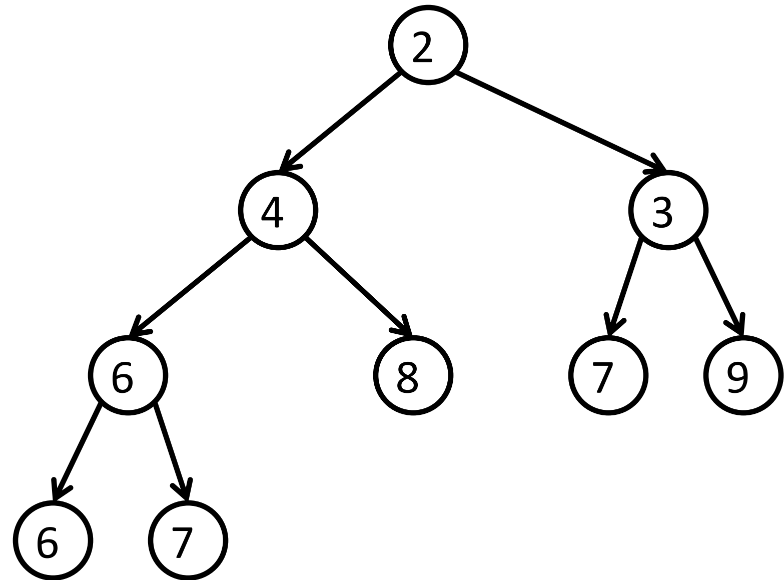
Extract Min

- Step 1: Remove the root
- Step 2: Replace it with the rightmost node in the last row
- Step 3: Repeatedly swap the root node with its smallest child until the heap property is no longer violated



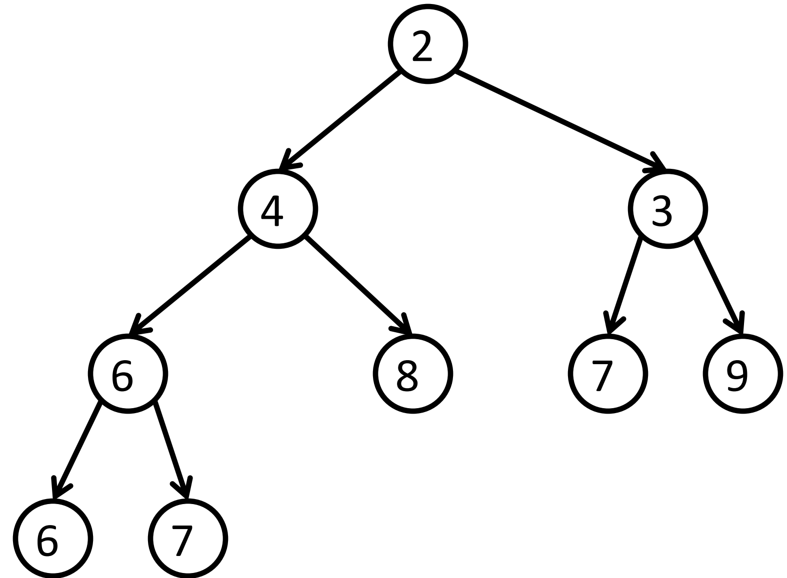
Extract Min

- Correctness: swapping with the minimum of the children maintains the heap property at that level
- Can formally prove this with induction



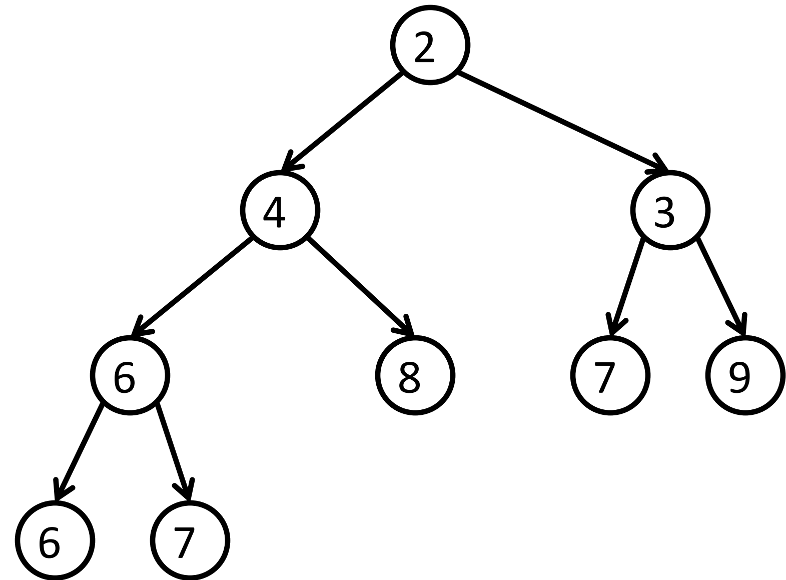
Implementation

- Working with trees and pointers not particularly simple
- Exploit the fact that the tree is complete
- Represent the tree as a list



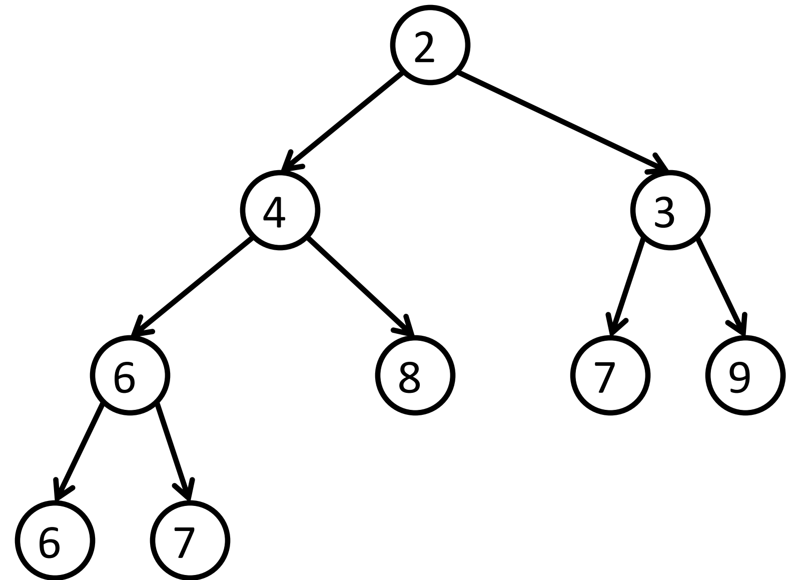
Implementation

- Working with trees and pointers not particularly simple
- Exploit the fact that the tree is complete
- Represent the tree as a list
- Represent tree in level order (this weeks lab)



Implementation

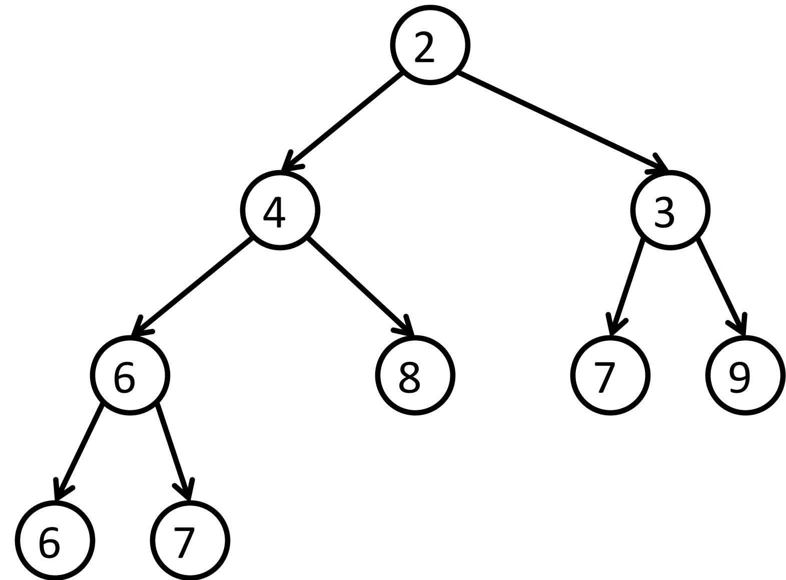
- Root is at index 0
- Left child of root at index 1
- Right child of root at index 2
- At what indexes are the children of the roots left child?



[2,4,3,6,8,7,9,6,7]

Implementation

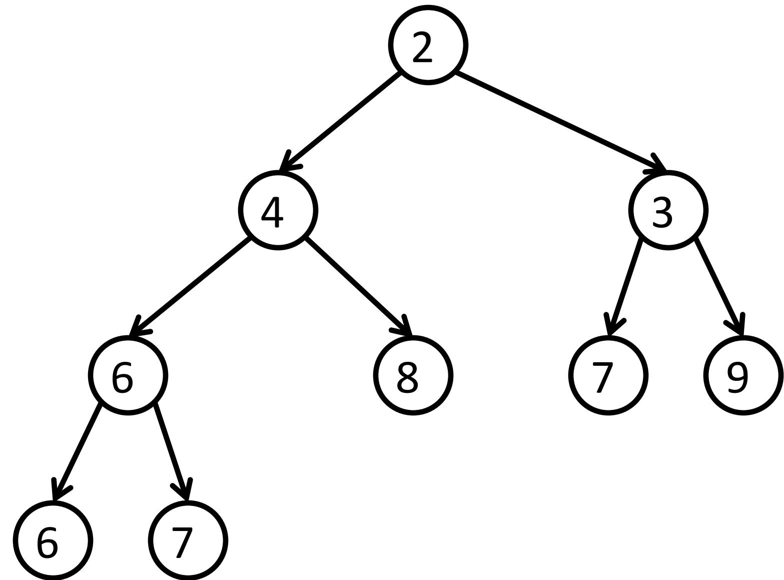
- Root is at index 0
- Left child of root at index 1
- Right child of root at index 2
- At what indexes are the children of the roots left child? 3,4



[2,4,3,6,8,7,9,6,7]

Implementation

- In general: at position i in the list, the left child is at position 2i+1 and the right child is at position 2i+2
- The parent of i is in position (i-1)/2



Runtime

- Worst case runtime of insert, extract_min: O(log n)
- Worst case runtime of size, min: O(1)
- Best case run times?
  - Always inserting the same value, everything is O(1)
  - Pretty boring case though…