CS2040S
Data Structures and Algorithms

Heaps and Priority Queues!

Today: Heaps and PQs

Priority Queue ADT:

– new API!

Binary Heap:

- new data structure!

Heapsort:

– new cool sorting algorithm!

Today: Heaps and PQs

Priority Queue ADT:

– new API!

Binary Heap:

- new data structure!

Heapsort:

– new cool sorting algorithm!

Priority Queues

A collection of (priority, id) pairs:

interface	**PriorityQueue<K>**	
void	insert(**int** p, **K** id)	*inserts id with priority p*
K	extractMax()	*return and remove id with maximum priority*
void	increaseKey(**int** p, **K** id)	*increase the priority of id to p*
void	decreaseKey(**int** p, **K** id)	*decrease the priority of id to p*
boolean	contains(**K** id)	*answers whether id is in the priority queue*
int	size(**K** id)	*returns the current number of pairs*

Priority Queues

Again, assume ids are unique

A collection of (priority, id) pairs:

interface	**PriorityQueue<K>**	
void	insert(**int** p, **K** id)	*inserts id with priority p*
K	extractMax()	*return and remove id with maximum priority*
void	increaseKey(**int** p, **K** id)	*increase the priority of id to p*
void	decreaseKey(**int** p, **K** id)	*decrease the priority of id to p*
boolean	contains(**K** id)	*answers whether id is in the priority queue*
int	size(**K** id)	*returns the current number of pairs*

Priority Queues

A collection of (priority, id) pairs:

interface	**PriorityQueue<K>**	
K	peekMaxId()	*returns the id with the max priority without removing it*
int	peekMaxPriority()	*returns the max priority without removing it*

Priority Queues

Again, assume ids are unique

A collection of (priority, id) pairs:

interface	**PriorityQueue<K>**	
K	peekMaxId()	*returns the id with the max priority without removing it*
int	peekMaxPriority()	*returns the max priority without removing it*

Java SE 21 Documentation:

Implementation note: this implementation provides O(log(n)) time for the <u>enqueuing and dequeuing</u> methods (`offer`, `poll`, `remove()` and `add`); <u>linear time</u> for the `remove(Object)` and `contains(Object)` methods; and constant time for the retrieval methods (`peek`, `element`, and `size`).

Priority Queues

Java lets people remove arbitrary elements too. But it costs linear time.

extractMax()

Java SE 21 Documentation:

`Implementation note: this implementation provides O(log(n)) time for the enqueuing and dequeuing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).`

Priority Queues

We will talk about supporting the `extractMax()` operation. But you can modify this to support `extractMin()` instead if you want to.

Priority Queues

Again, let’s first think about some straightforward implementations.

Priority Queues

Sorted Array:

- insert: O(n)
- increase/decrease key: O(n)
- extractMax: O(1)
- contains: O(n)
- peekMax: O(1)

Priority Queues

Sorted Array:

- insert: O(n)
- increase/decrease key: O(n)
- extractMax: O(1)
- contains: O(n)
- peekMax: O(1)

If we also used a hashtable, we can make this O(1) expected.

Priority Queues

Assuming our hashtable is happy*

Balanced BST + Hashtable:

- insert*: O(log n)
- increase/decrease key*: O(log n)
- extractMax*: O(log n)
- contains*: O(1)
- peekMax: O(1)

Idea:
The tree is keyed on the priorities and map to IDs. The hash table maps IDs to tree nodes. This way, given an ID, we know where it is in the tree.

Priority Queues

Assuming our hashtable is happy*

Balanced BST + Hashtable:

- insert*: O(log n)
- increase/decrease key*: O(log n)
- extractMax*: O(log n)
- contains*: O(1)
- peekMax: O(1)

Technically nothing wrong with this asymptotically. But we are going to try to accomplish the same complexities with **heaps** instead.

Today: Heaps and PQs

Priority Queue ADT:

– new API!

Binary Heap:

- new data structure!

Heapsort:

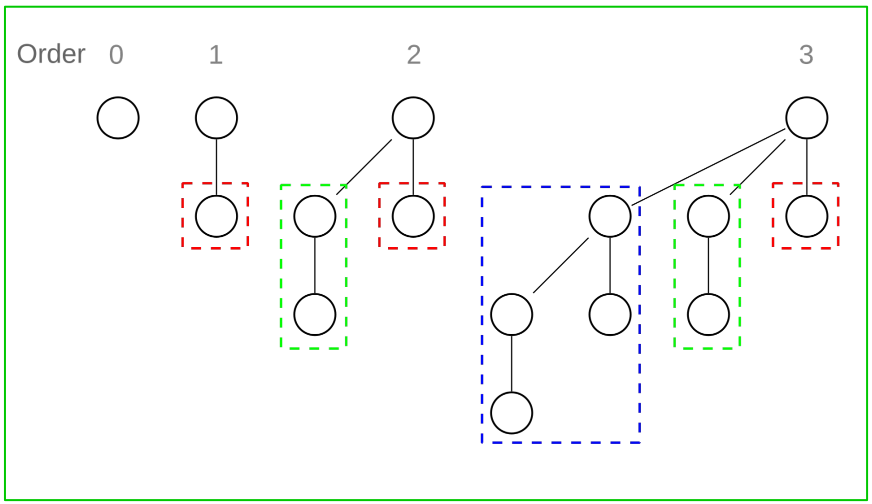
– new cool sorting algorithm!

Heaps

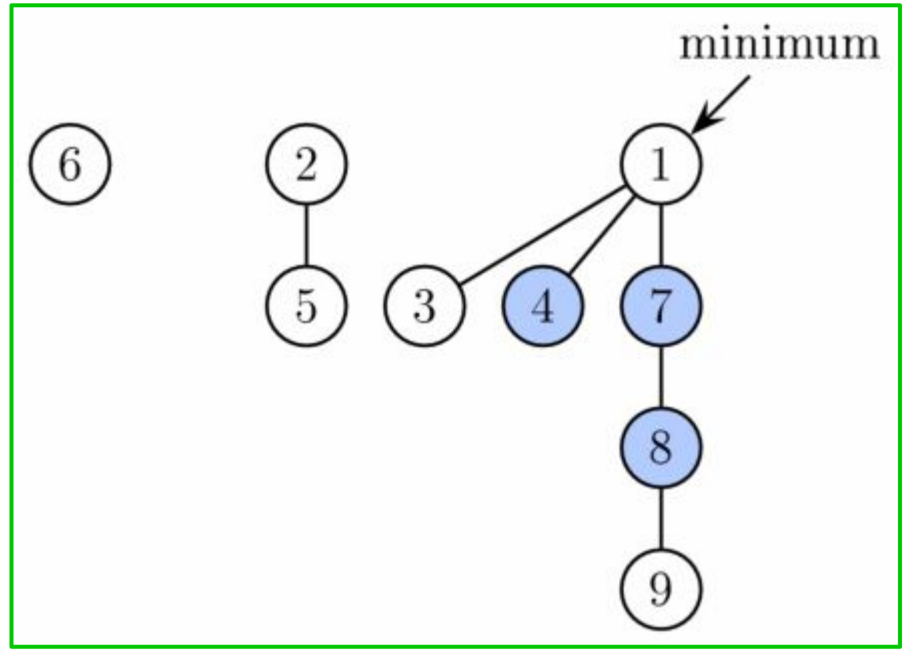
Why learn heaps if we have already a way to implement this ADT using trees and hashtables?

Heaps

Heaps actually come in many shapes and forms.



Binomial heaps



Fibonacci heaps

Heaps

Heaps actually come in many shapes and forms.

- 2–3 heap
- B-heap
- Beap
- Binary heap
- Binomial heap
- Brodal queue
- *d*-ary heap
- Fibonacci heap
- K-D Heap
- Leaf heap
- Leftist heap
- Skew binomial heap
- Strict Fibonacci heap
- Min-max heap
- Pairing heap
- Radix heap
- Randomized meldable heap
- Skew heap
- Soft heap
- Ternary heap
- Treap
- Weak heap

We will cover this one!

Heaps

Heaps actually come in many shapes and forms.

Different heaps actually provide

- Additional operations:
  - Splice
  - Merge

- Different complexities for existing operations:
  - E.g. Fibonacci Heaps do insert, decrease key, merge in amortised O(1)

Heaps

Heaps actually come in many shapes and forms.

Different heaps actually provide

- Additional operations:
  - Splice
  - Merge

But horribly impractical because of the huge amount of pointer chasing

- Different complexities for existing operations:
  - E.g. Fibonacci Heaps do insert, decrease key, merge in amortised O(1)

Heaps

Heaps actually come in many shapes and forms.

Comparison of theoretic bounds for variants [edit]

Here are time complexities[8] of various heap data structures. The abbreviation am. indicates that the given complexity is amortized, otherwise it is a worst-case complexity. For the meaning of "$O(f)$" and "$\Theta(f)$" see Big O notation. Names of operations assume a max-heap.

Operation	find-max	delete-max	increase-key	insert	meld	make-heap[b]
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(n)$ am.
Leftist[10]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Binomial[8][12]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ am.	$\Theta(\log n)$[c]	$\Theta(n)$
Skew binomial[13]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$[c]	$\Theta(n)$
2–3 heap[15]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$	$\Theta(1)$ am.	$O(\log n)$[c]	$\Theta(n)$
Bottom-up skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$ am.	$\Theta(n)$ am.
Pairing[16]	$\Theta(1)$	$O(\log n)$ am.	$o(\log n)$ am.[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Rank-pairing[19]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Fibonacci[8][20]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Strict Fibonacci[21][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Brodal[22][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$[23]

Heaps

Heaps actually come in many shapes and forms.

Comparison of theoretic bounds for variants [edit]

Here are time complexities[8] of various heap data structures. The abbreviation am. indicates that the given complexity is amortized, otherwise it is a worst-case complexity. For the meaning of "$O(f)$" and "$\Theta(f)$" see Big O notation. Names of operations assume a max-heap.

Operation	find-max	delete-max	increase-key	insert	meld	make-heap[b]
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(n)$ am.
Leftist[10]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Binomial[8][12]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ am.	$\Theta(\log n)$[c]	$\Theta(n)$
Skew binomial[13]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$[c]	$\Theta(n)$
2–3 heap[15]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$	$\Theta(1)$ am.	$O(\log n)$[c]	$\Theta(n)$
Bottom-up skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$ am.	$\Theta(n)$ am.
Pairing[16]	$\Theta(1)$	$O(\log n)$ am.	$o(\log n)$ am.[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Rank-pairing[19]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Fibonacci[8][20]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Strict Fibonacci[21][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Brodal[22][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$[23]

Heaps

Heaps actually come in many shapes and forms.

Comparison of theoretic bounds for variants [edit]

Here are time complexities[8] of various heap data structures. The abbreviation am. indicates that the given complexity is amortized, otherwise it is a worst-case complexity. For the meaning of "$O(f)$" and "$\Theta(f)$" see Big O notation. Names of operations assume a max-heap.

Operation	find-max	delete-max	increase-key	insert	meld	make-heap[b]
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(n)$ am.
Leftist[10]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Binomial[8][12]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ am.	$\Theta(\log n)$[c]	$\Theta(n)$
Skew binomial[13]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$[c]	$\Theta(n)$
2–3 heap[15]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$	$\Theta(1)$ am.	$O(\log n)$[c]	$\Theta(n)$
Bottom-up skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$ am.	$\Theta(n)$ am.
Pairing[16]	$\Theta(1)$	$O(\log n)$ am.	$o(\log n)$ am.[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Rank-pairing[19]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Fibonacci[8][20]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Strict Fibonacci[21][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Brodal[22][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$[23]

Heaps

Heaps actually come in many shapes and forms.

Comparison of theoretic bounds for variants [edit]

Here are time complexities[8] of various heap data structures. The abbreviation am. indicates that the given complexity is amortized, otherwise it is a worst-case complexity. For the meaning of "$O(f)$" and "$\Theta(f)$" see Big O notation. Names of operations assume a max-heap.

Operation	find-max	delete-max	increase-key	insert	meld	make-heap[b]
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(n)$ am.
Leftist[10]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Binomial[8][12]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ am.	$\Theta(\log n)$[c]	$\Theta(n)$
Skew binomial[13]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$[c]	$\Theta(n)$
2–3 heap[15]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$	$\Theta(1)$ am.	$O(\log n)$[c]	$\Theta(n)$
Bottom-up skew[9]	$\Theta(1)$	$O(\log n)$ am.	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$ am.	$\Theta(n)$ am.
Pairing[16]	$\Theta(1)$	$O(\log n)$ am.	$o(\log n)$ am.[d]	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Rank-pairing[19]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Fibonacci[8][20]	$\Theta(1)$	$O(\log n)$ am.	$\Theta(1)$ am.	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Strict Fibonacci[21][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Brodal[22][e]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$[23]

Heaps

The linux kernel uses heaps too.

Min Heap API

Author: Kuan-Wei Chiu <visitorckw@gmail.com>

Introduction

The Min Heap API provides a set of functions and macros for managing min-heaps in the Linux kernel. A min-heap is a binary tree structure where the value of each node is less than or equal to the values of its children, ensuring that the smallest element is always at the root.

This document provides a guide to the Min Heap API, detailing how to define and use min-heaps. Users should not directly call functions with **__min_heap_*()** prefixes, but should instead use the provided macro wrappers.

In addition to the standard version of the functions, the API also includes a set of inline versions for performance-critical scenarios. These inline functions have the same names as their non-inline counterparts but include an **_inline** suffix. For example, **__min_heap_init_inline** and its corresponding macro wrapper **min_heap_init_inline**. The inline versions allow custom comparison and swap functions to be called directly, rather than through indirect function calls. This can significantly reduce overhead, especially when CONFIG_MITIGATION_RETPOLINE is enabled, as indirect function calls become more expensive. As with the non-inline versions, it is important to use the macro wrappers for inline functions instead of directly calling the functions themselves.

Data Structures

Min-Heap Definition

The core data structure for representing a min-heap is defined using the **MIN_HEAP_PREALLOCATED** and **DEFINE_MIN_HEAP** macros. These macros allow you to define a min-heap with a preallocated buffer or dynamically al-

Heaps

So does C++

Defined in header <algorithm>

push_heap	adds an element to a max heap (function template)
ranges::push_heap (C++20)	adds an element to a max heap (algorithm function object)
pop_heap	removes the largest element from a max heap (function template)
ranges::pop_heap (C++20)	removes the largest element from a max heap (algorithm function object)
make_heap	creates a max heap out of a range of elements (function template)
ranges::make_heap (C++20)	creates a max heap out of a range of elements (algorithm function object)
sort_heap	turns a max heap into a range of elements sorted in ascending order (function template)
ranges::sort_heap (C++20)	turns a max heap into a range of elements sorted in ascending order (algorithm function object)
is_heap (C++11)	checks if the given range is a max heap (function template)
ranges::is_heap (C++20)	checks if the given range is a max heap (algorithm function object)
is_heap_until (C++11)	finds the largest subrange that is a max heap (function template)
ranges::is_heap_until (C++20)	finds the largest subrange that is a max heap (algorithm function object)

Heaps

- Reason #1
  - Knowing at at least the idea of a heap frees you up from only thinking about trees and search tree orderings. And after that, it's easier to learn the rest of the heaps.
  - This also means at some point you can either beat trees for operations like **`insert/extractMax`** or implement new operations like **`merge/split`**.

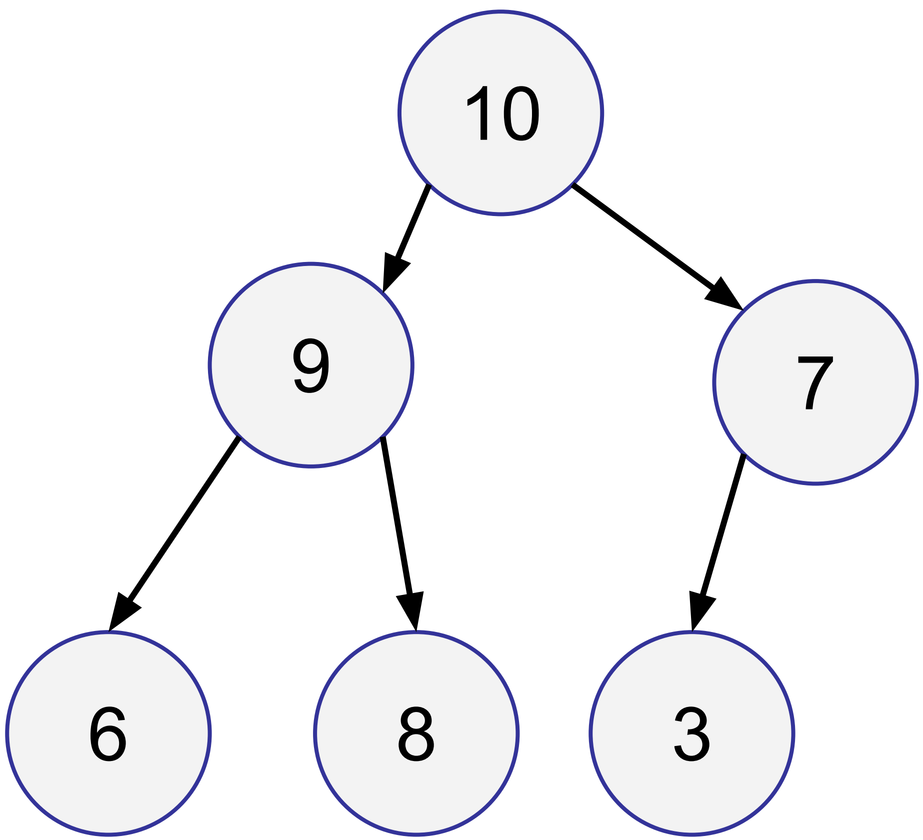
Heaps

- Reason #1
  - Knowing at at least the idea of a heap frees you up from only thinking about trees and search tree orderings. And after that, it's easier to learn the rest of the heaps.
  - This also means at some point you can either beat trees for operations like **`insert/extractMax`** or implement new operations like **`merge/split`**.

- Reason #2
  - Very much used in the wild. See linux/C++ libraries.

Binary Heaps:

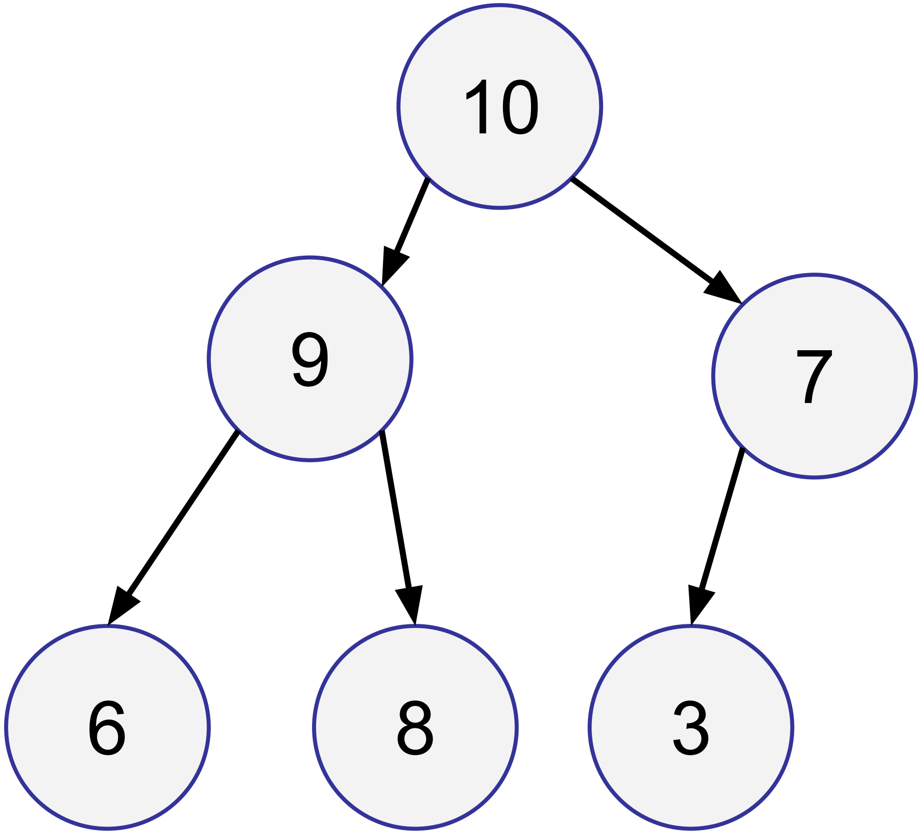
New data structure, new rules!



Binary Heaps:

Invariant/Property 1:

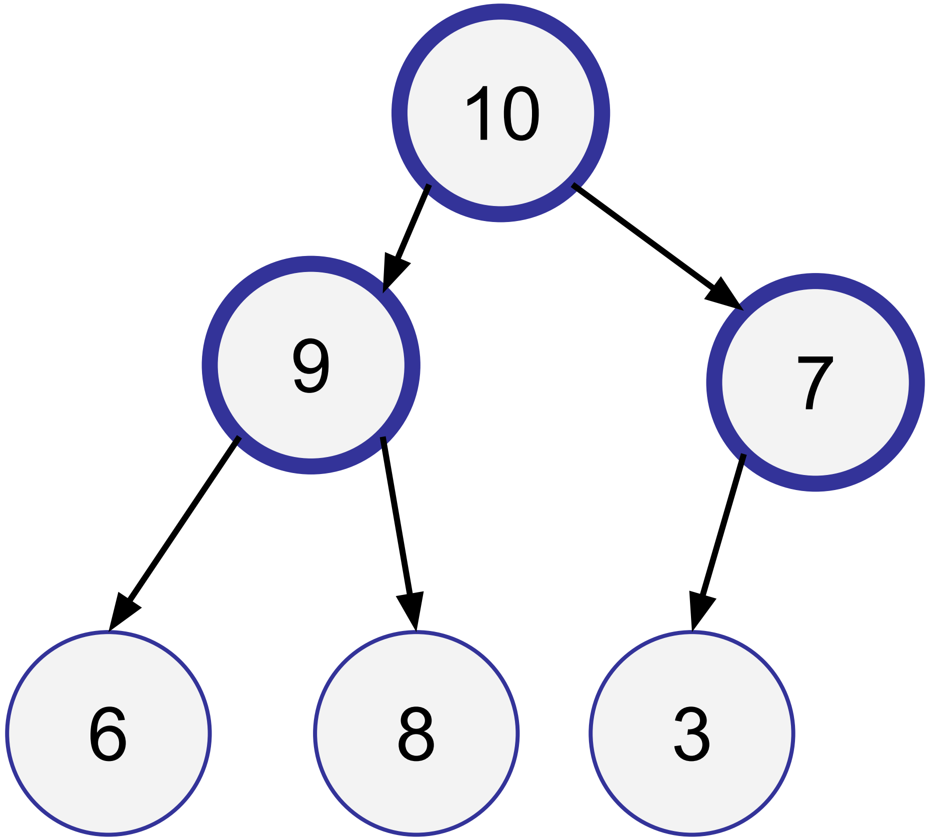
The priority at each node is < the priority at its parent



Binary Heaps:

Invariant/Property 1:

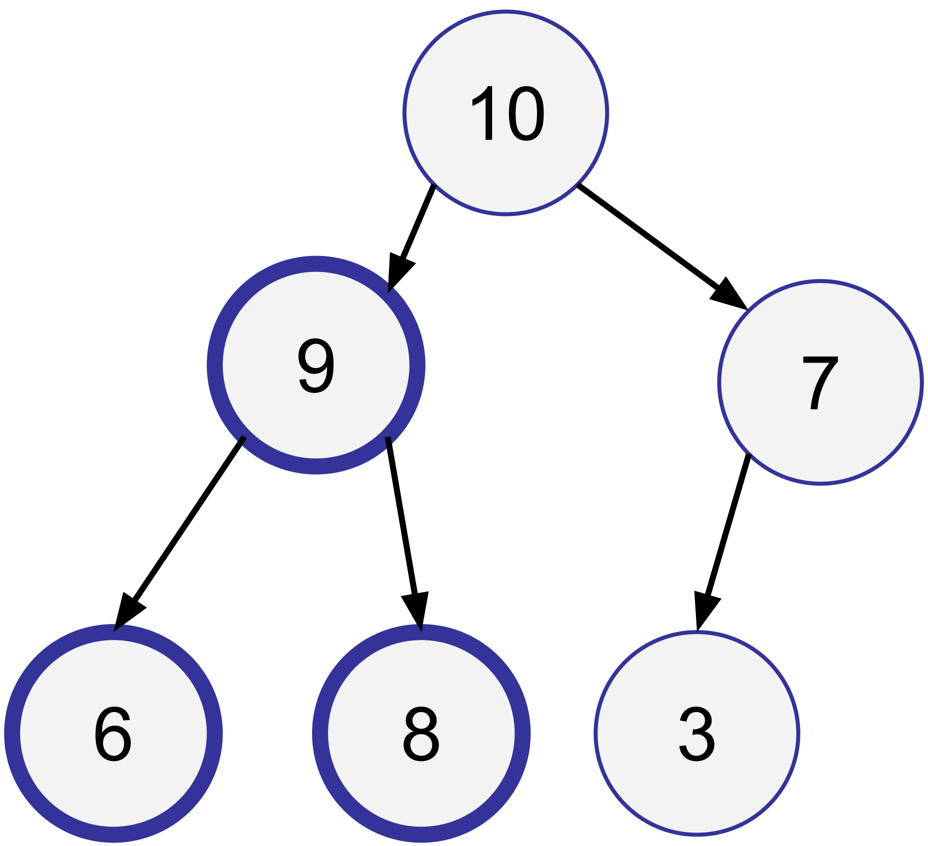
The priority at each node is < the priority at its parent



Binary Heaps:

Invariant/Property 1:

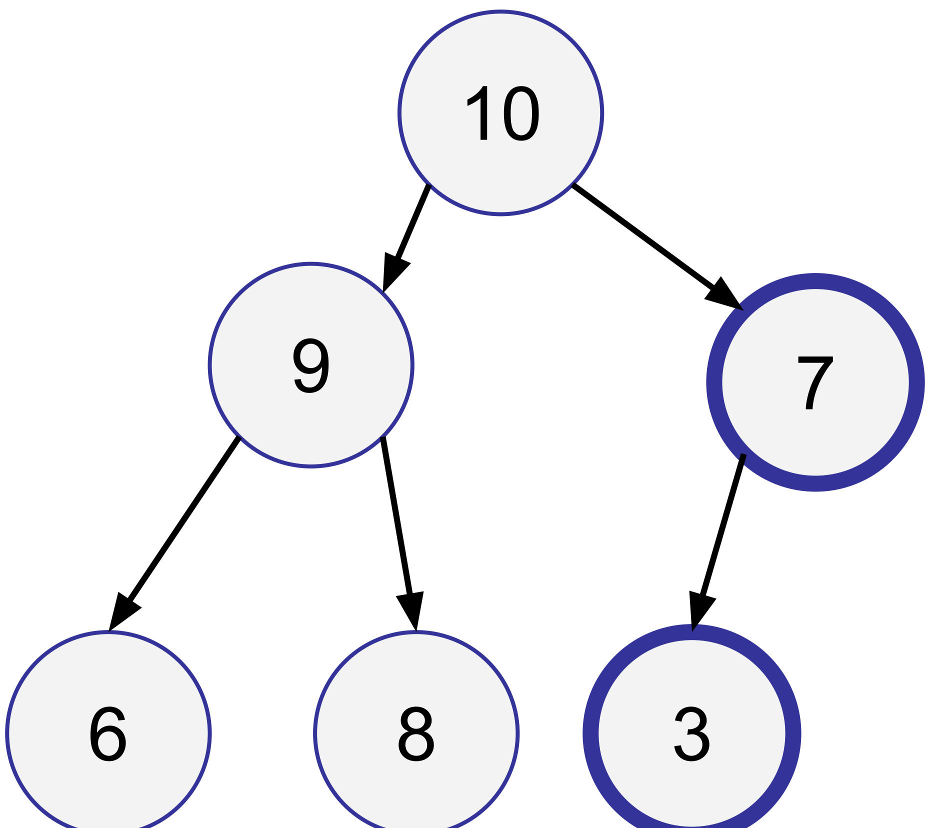
The priority at each node is < the priority at its parent



Binary Heaps:

Invariant/Property 1:

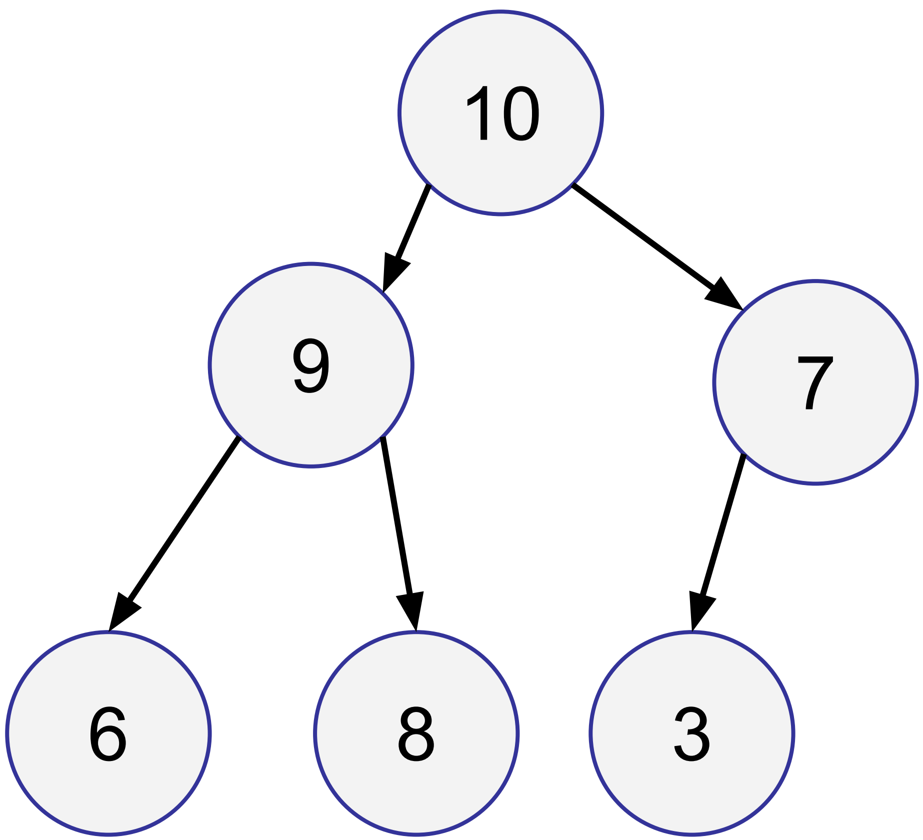
The priority at each node is < the priority at its parent



Binary Heaps:

Invariant/Property 2:

The heap tree is a complete binary tree

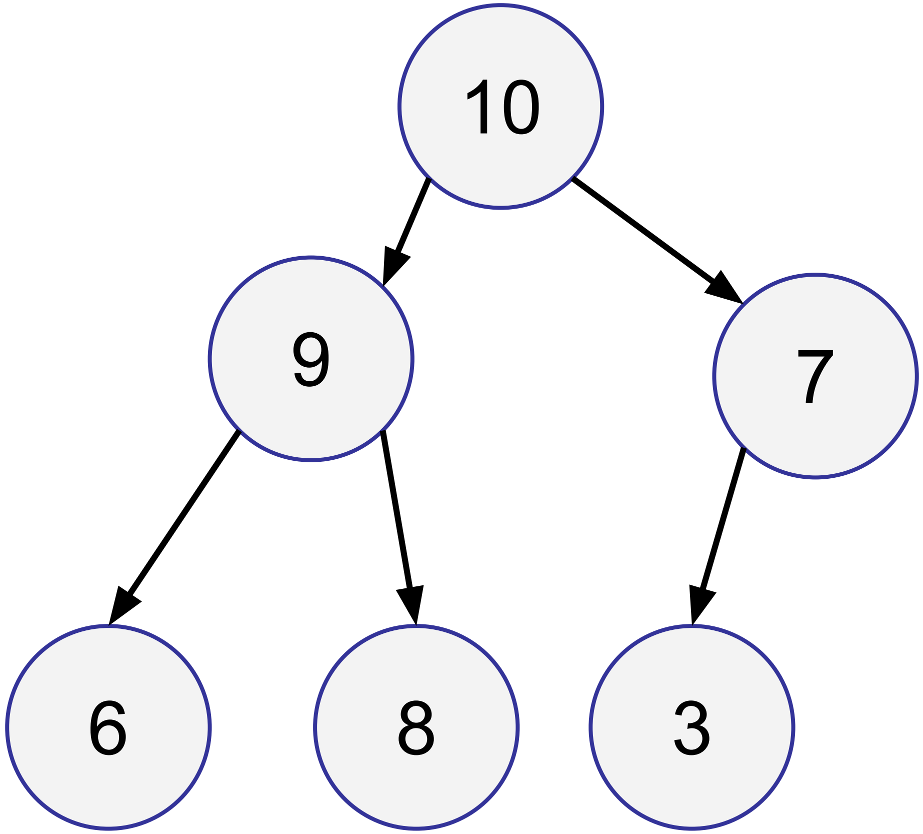


Binary Heaps:

Invariant/Property 2:

The heap tree is a <u>complete binary</u> tree

Every level is filled except for the last level, filled from left to right.

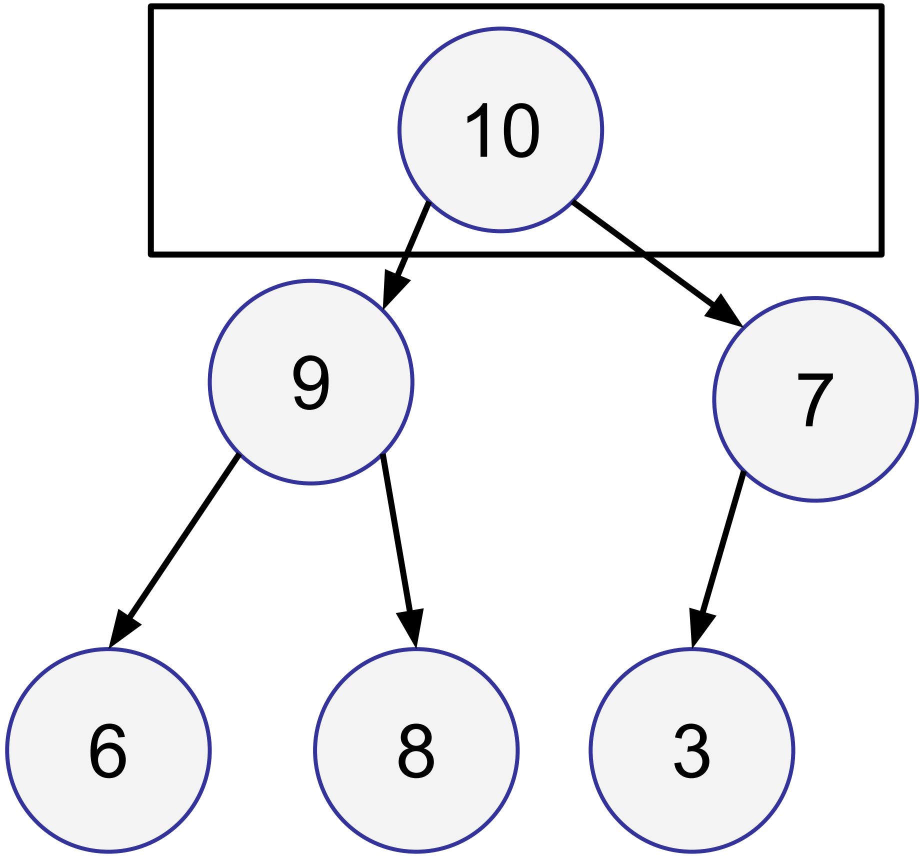


Binary Heaps:

Invariant/Property 2:

The heap tree is a complete binary tree

Every level is filled except for the last level, filled from left to right.

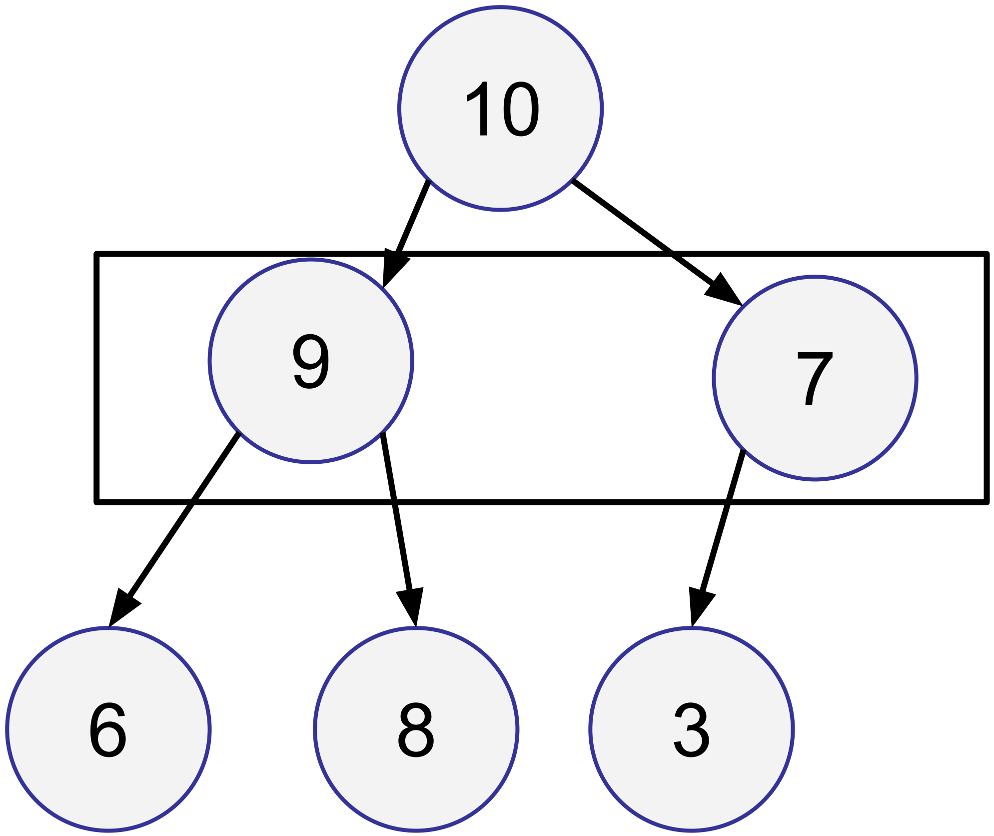


Binary Heaps:

Invariant/Property 2:

The heap tree is a complete binary tree

Every level is filled except for the last level, filled from left to right.

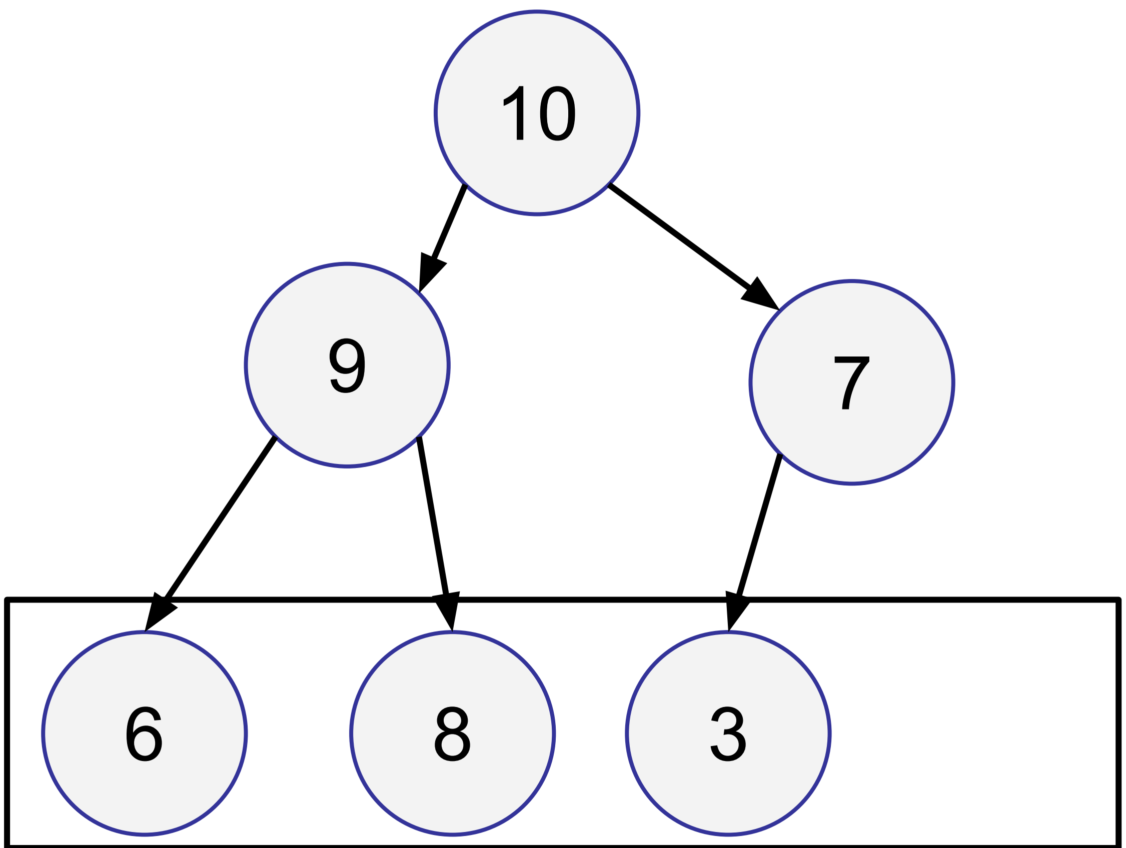


Binary Heaps:

Invariant/Property 2:

The heap tree is a complete binary tree

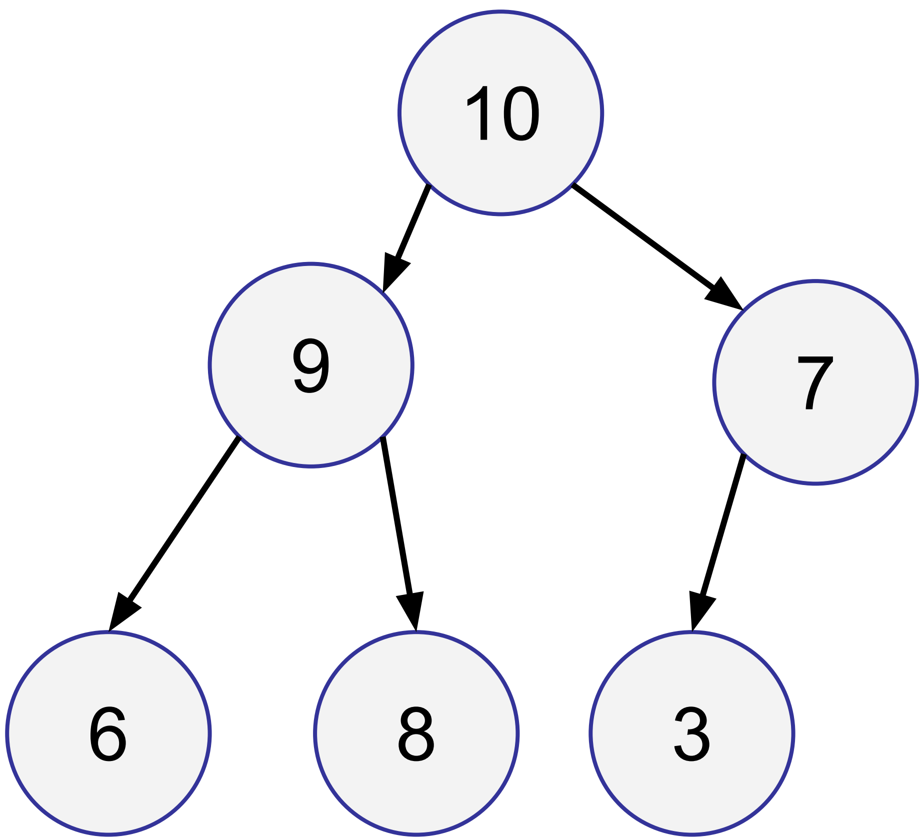
Every level is filled except for the last level, filled from left to right.



Binary Heaps:

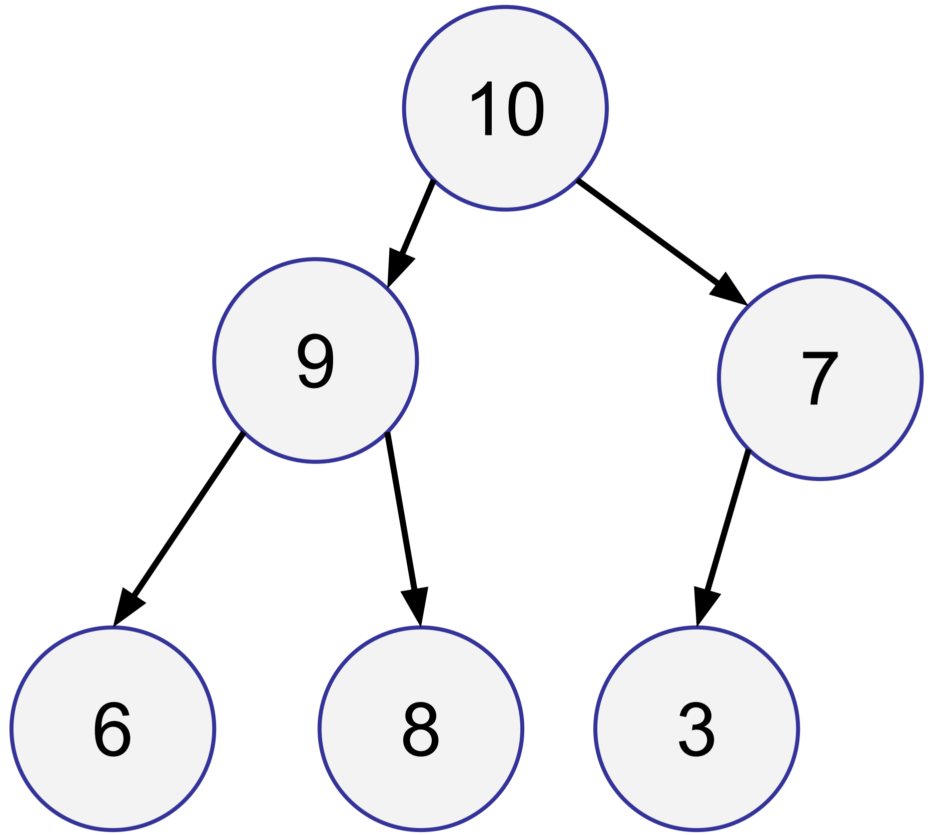
Invariant 1 (Heap ordering): Helps with finding max quickly!

Invariant 2 (Shape): Actually helps keep a O(log n) height. And it simplifies our node layout.



Binary Heaps:

What you see:

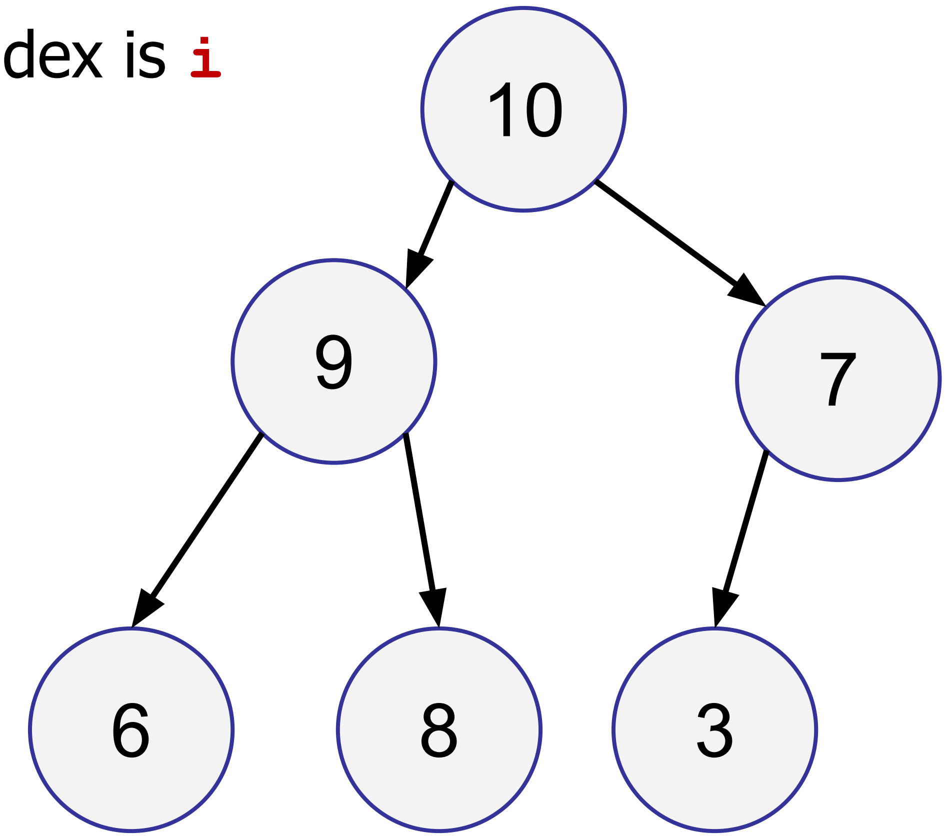


What is going on in memory:

6	10	9	7	6	8	3	

Binary Heaps:

If your current index is `i`



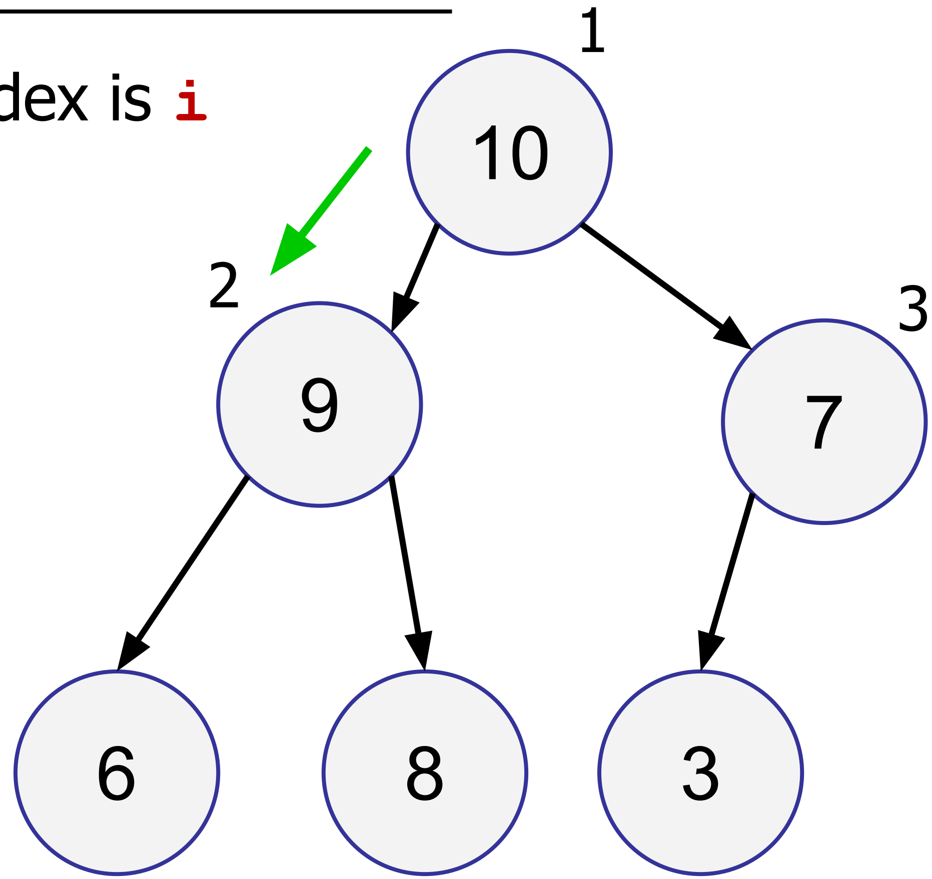
What is going on in memory:

6	10	9	7	6	8	3	

Binary Heaps:

If your current index is **i**

Left child is at **2i**



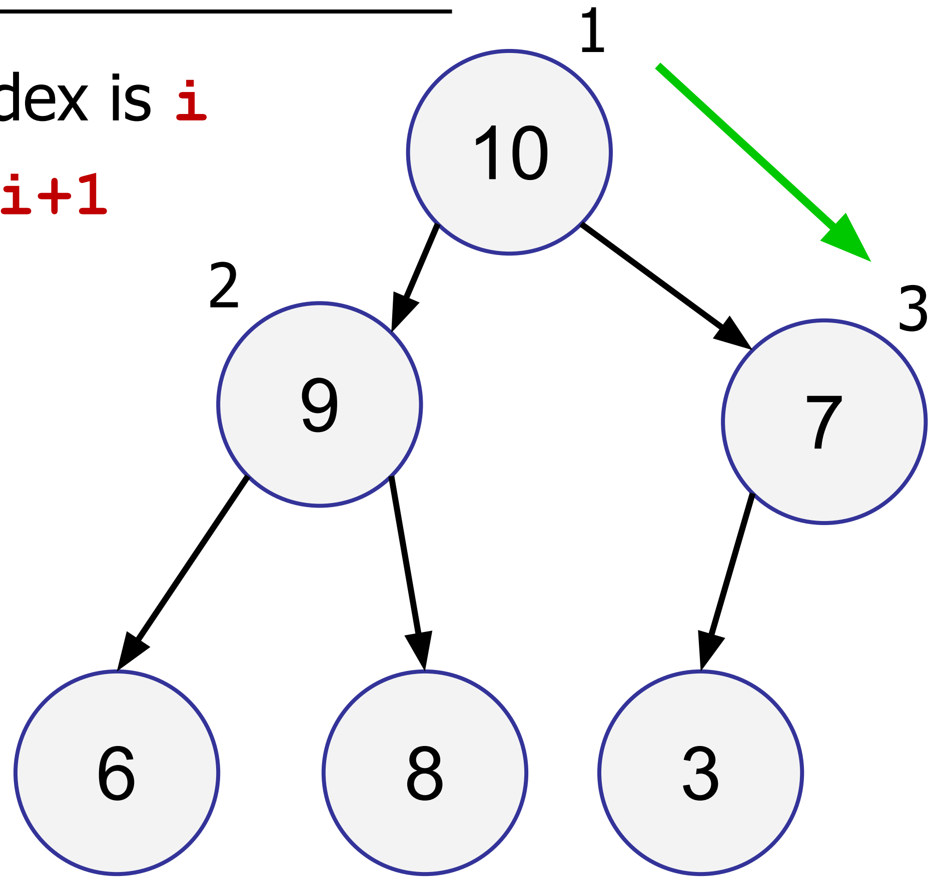
What is going on in memory:

6	10	9	7	6	8	3	

Binary Heaps:

If your current index is `i`

Right child is at `2i+1`



What is going on in memory:

6	10	9	7	6	8	3	

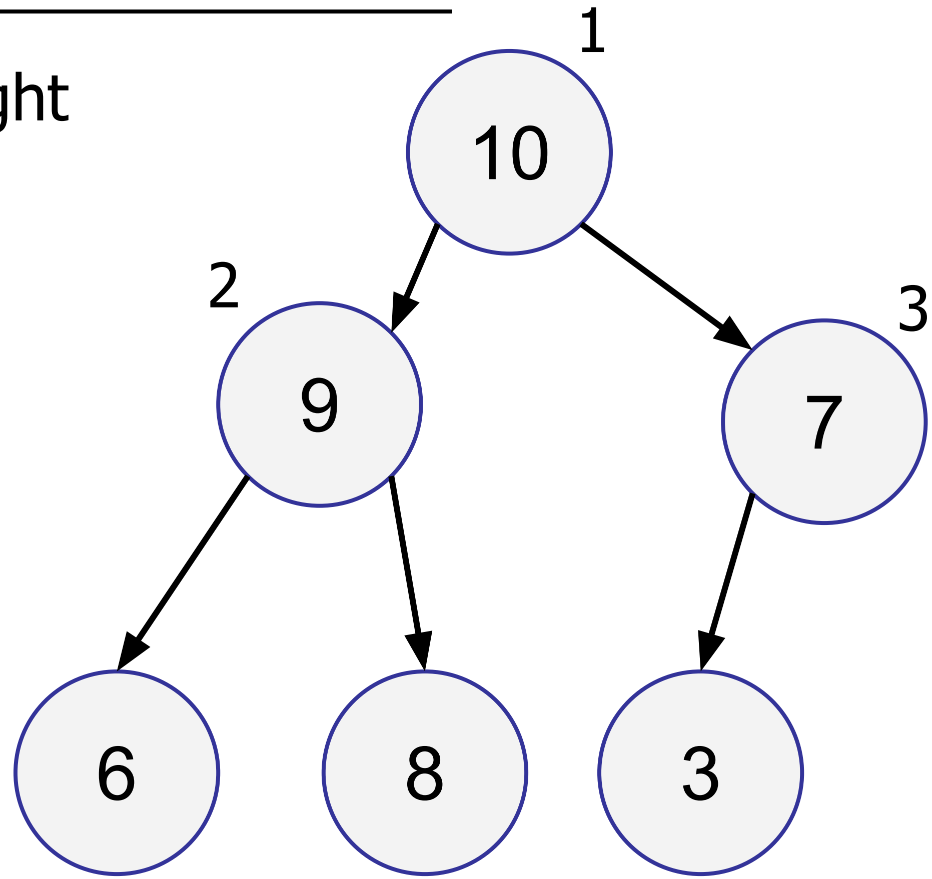
What is the height of the tree if we had n items?

1. O(log n)
2. O(n)
3. O(1)

Binary Heaps:

This tree has height

O(log n).

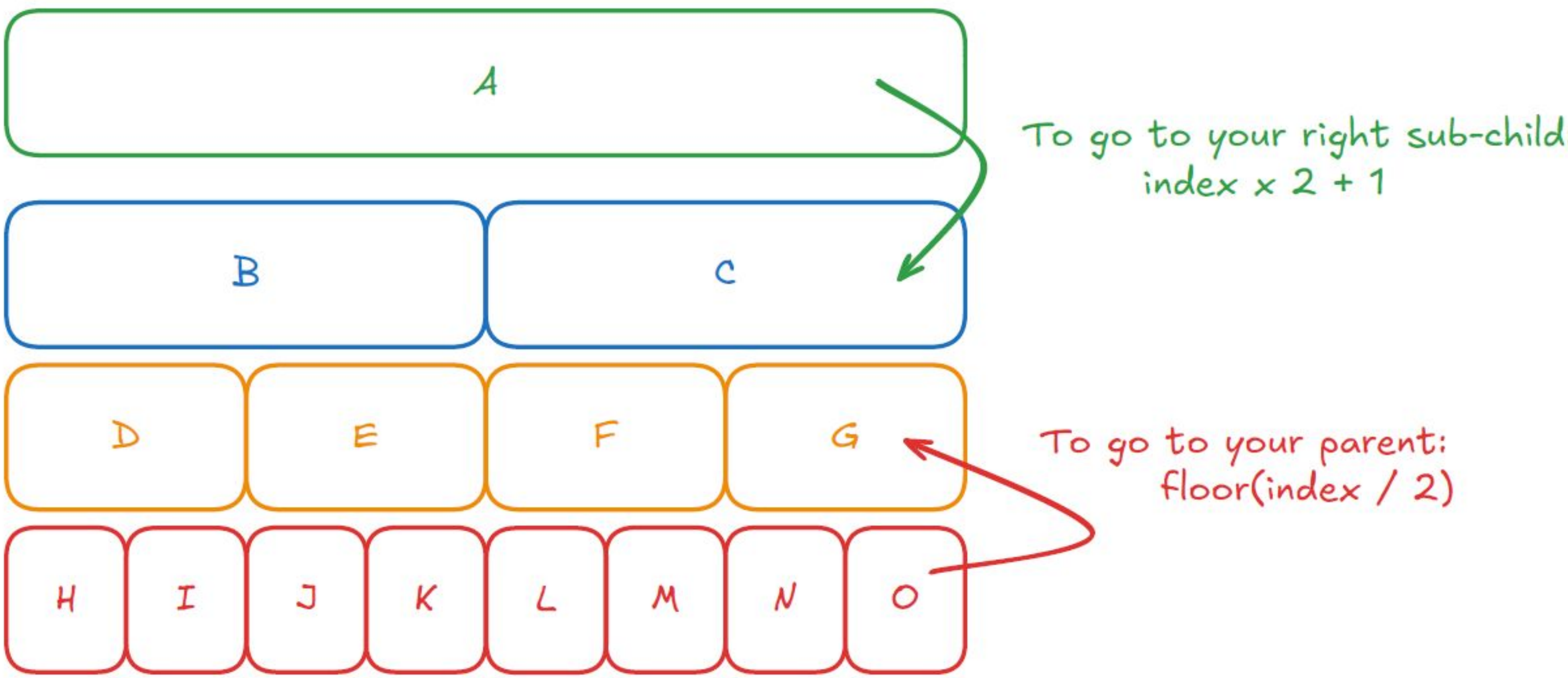


What is going on in memory:

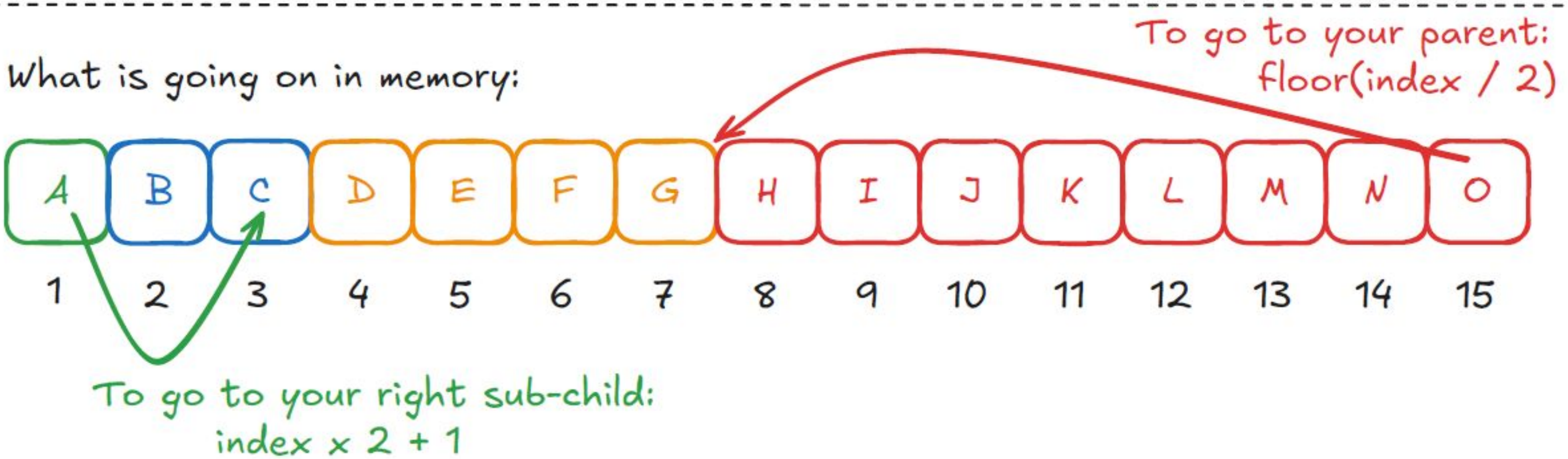
6	10	9	7	6	8	3	

Binary Heaps:

Conceptually what we see:

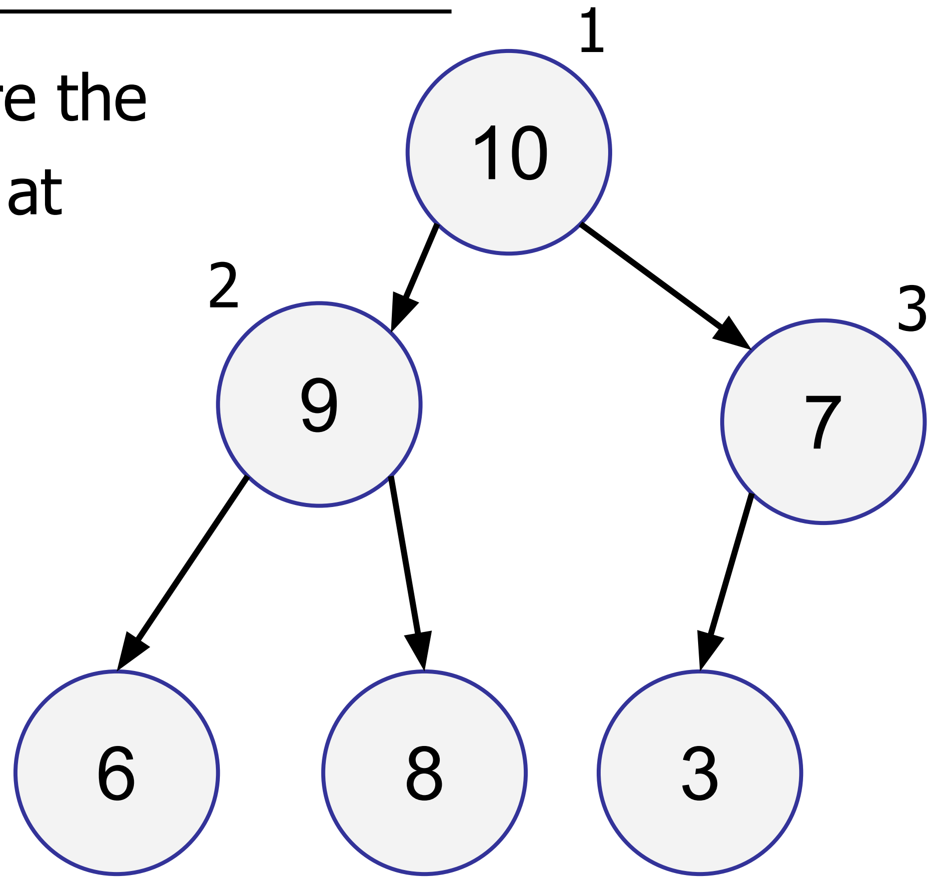


What is going on in memory:



Binary Heaps:

Might as well store the number of nodes at index 0

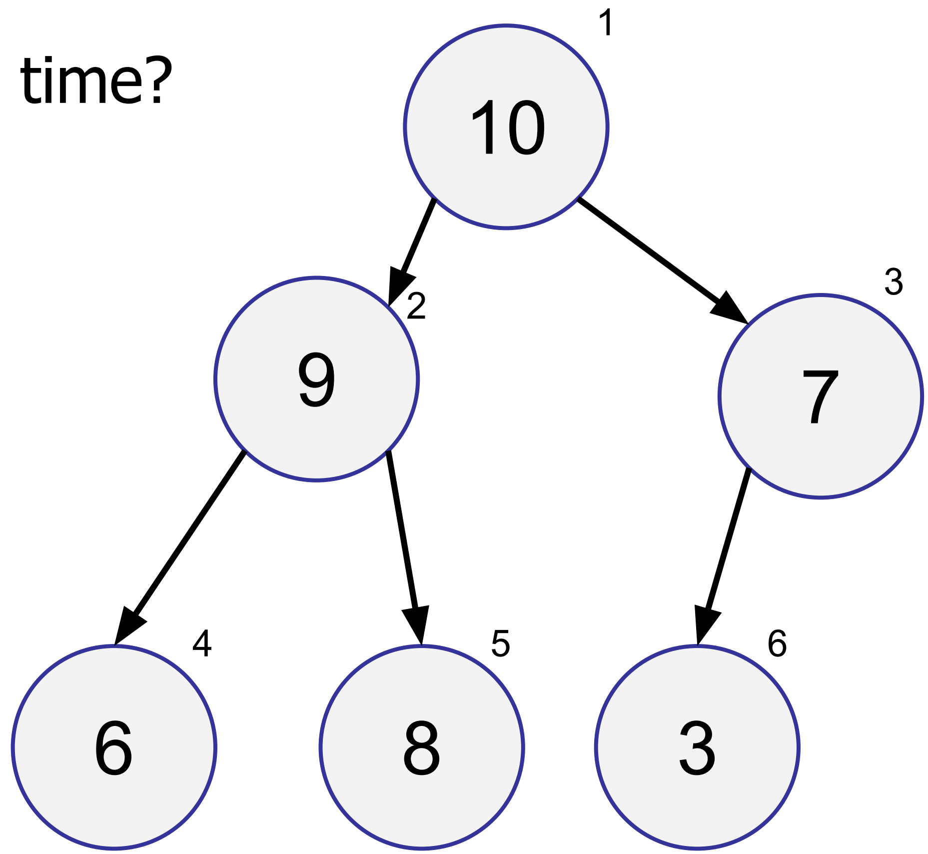


What is going on in memory:

6	10	9	7	6	8	3	

Binary Heaps:

How do we insert in O(log n) time?

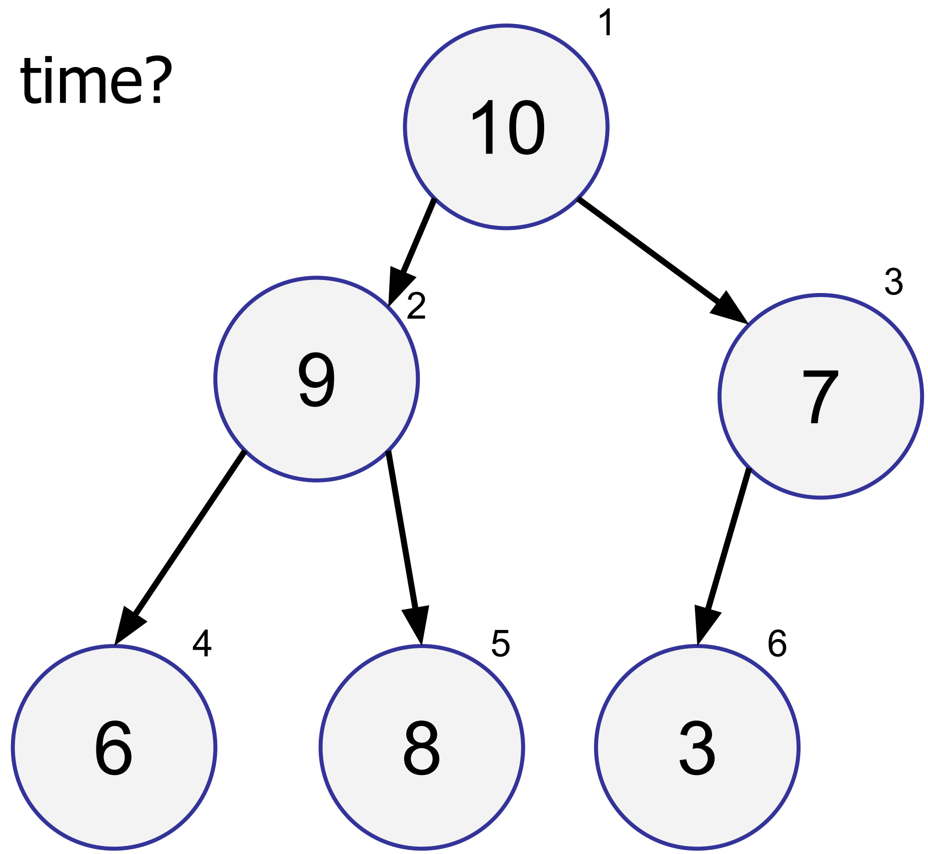


6	10	9	7	6	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

How do we insert in O(log n) time?

E.g. insert something with
priority 20

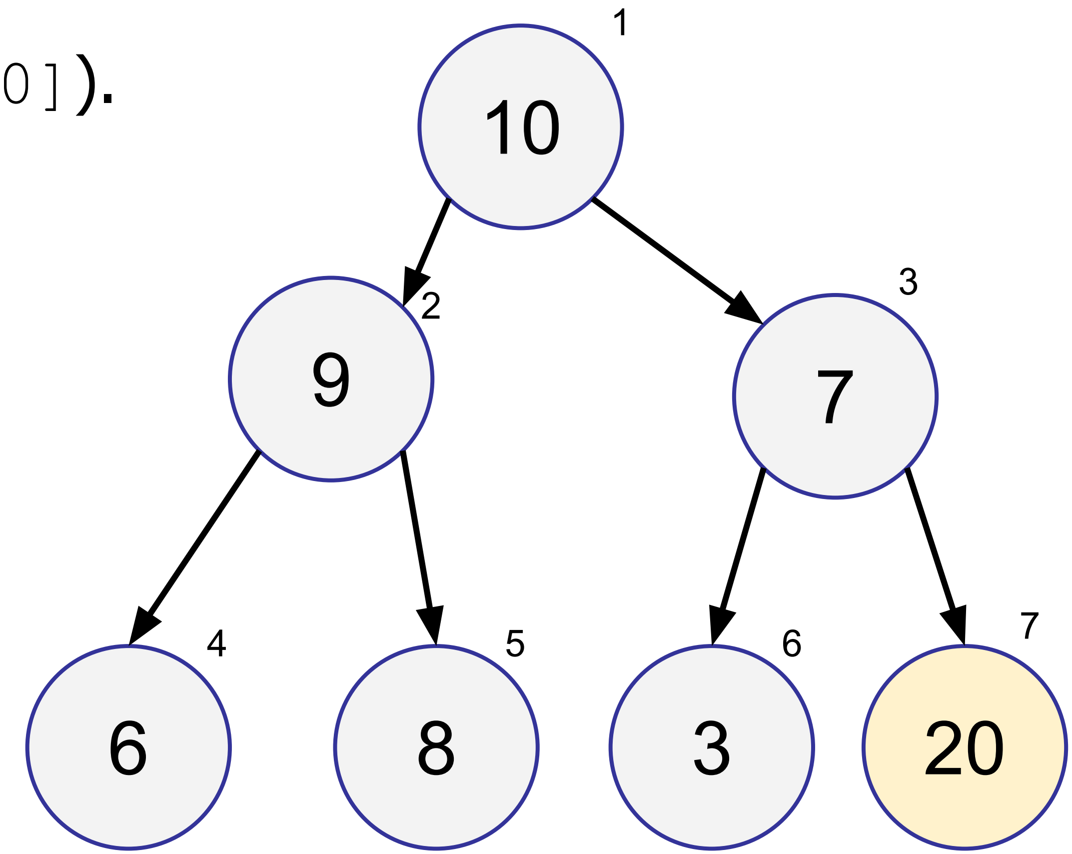


0	1	2	3	4	5	6	7
6	10	9	7	6	8	3	

Binary Heaps:

Increment our size (at `arr[0]`).

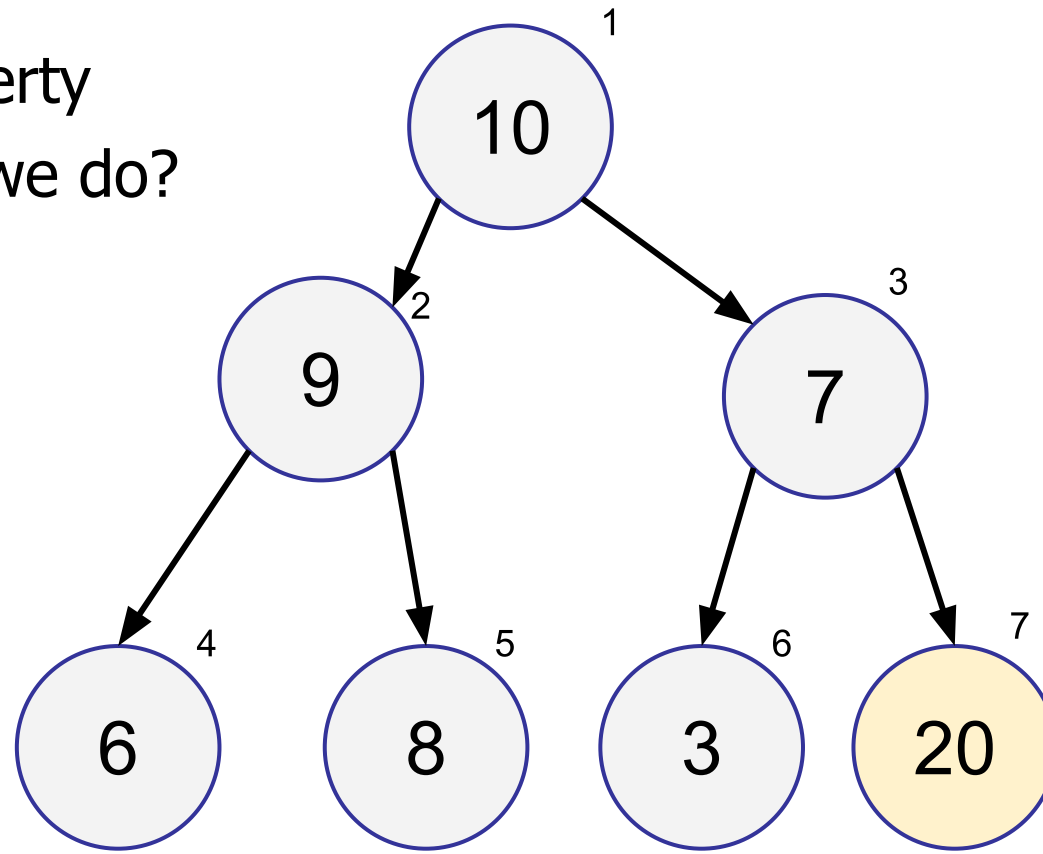
Place 20 at `arr[size]`



7	10	9	7	6	8	3	20
0	1	2	3	4	5	6	7

Binary Heaps:

Now the heap ordering property is violated! What should we do?

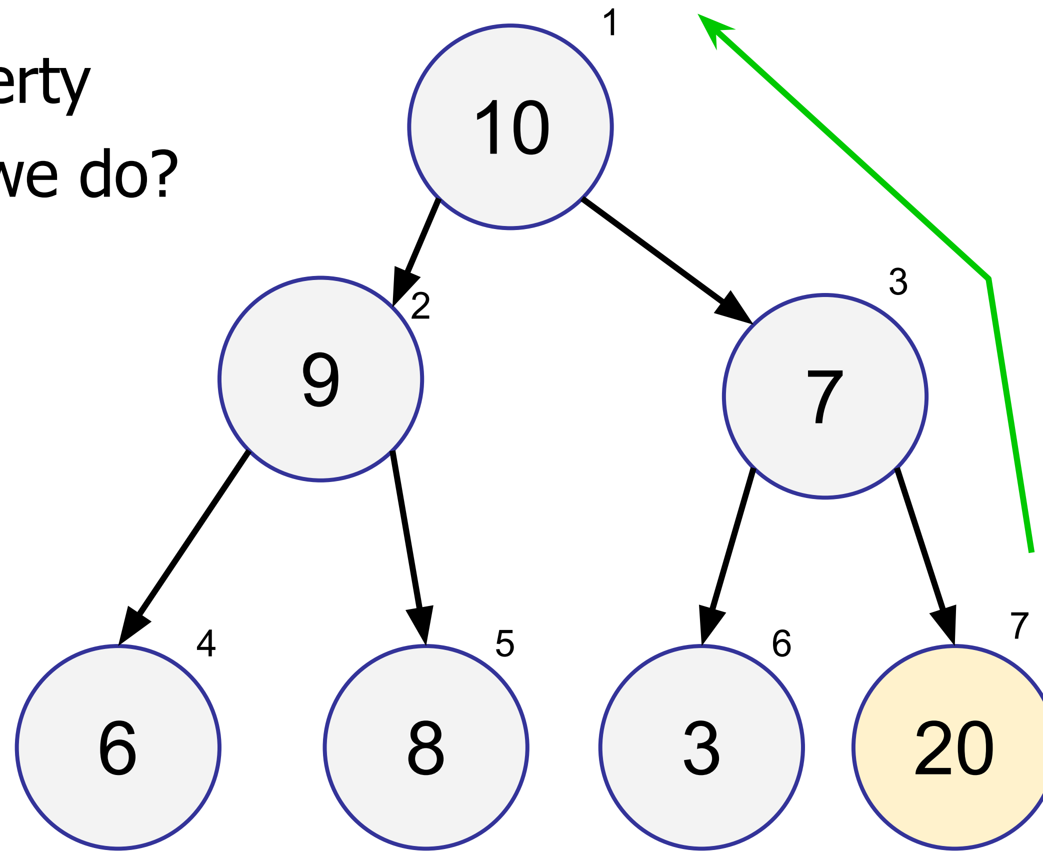


7	10	9	7	6	8	3	20
0	1	2	3	4	5	6	7

Binary Heaps:

Now the heap ordering property is violated! What should we do?

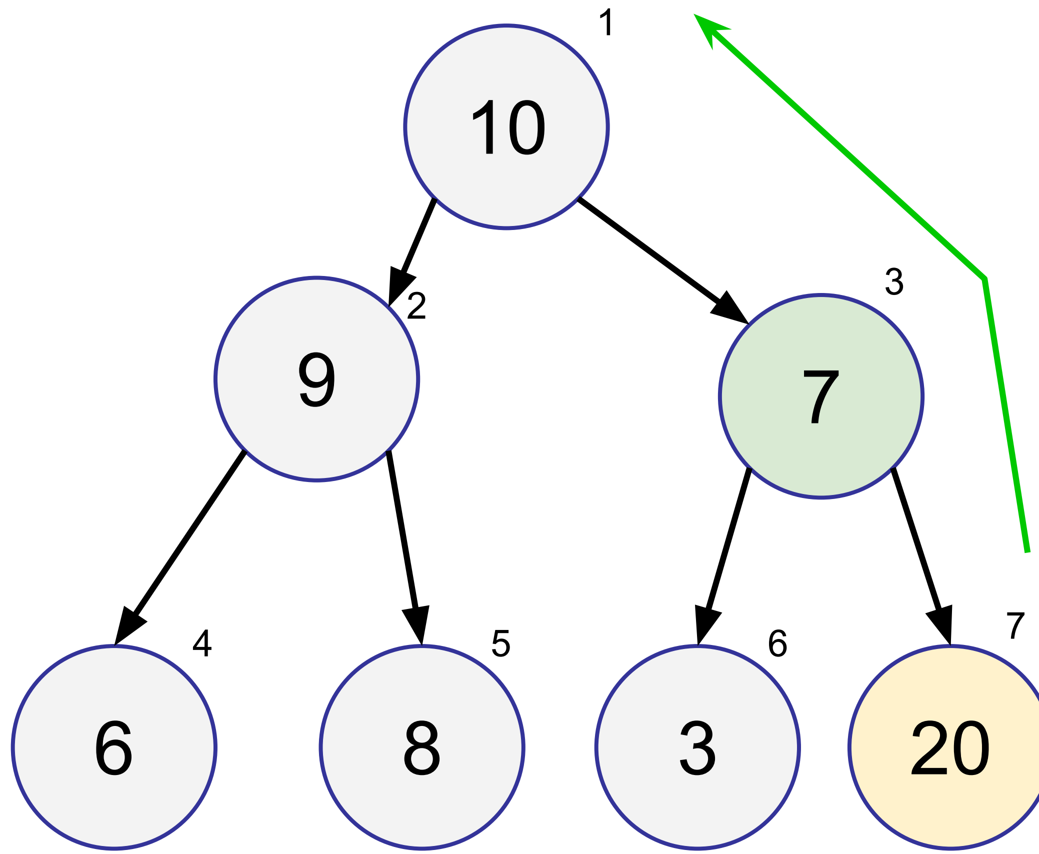
We can try fixing it by "bubbling" it up.



7	10	9	7	6	8	3	20
0	1	2	3	4	5	6	7

Binary Heaps:

Compare against the parent.



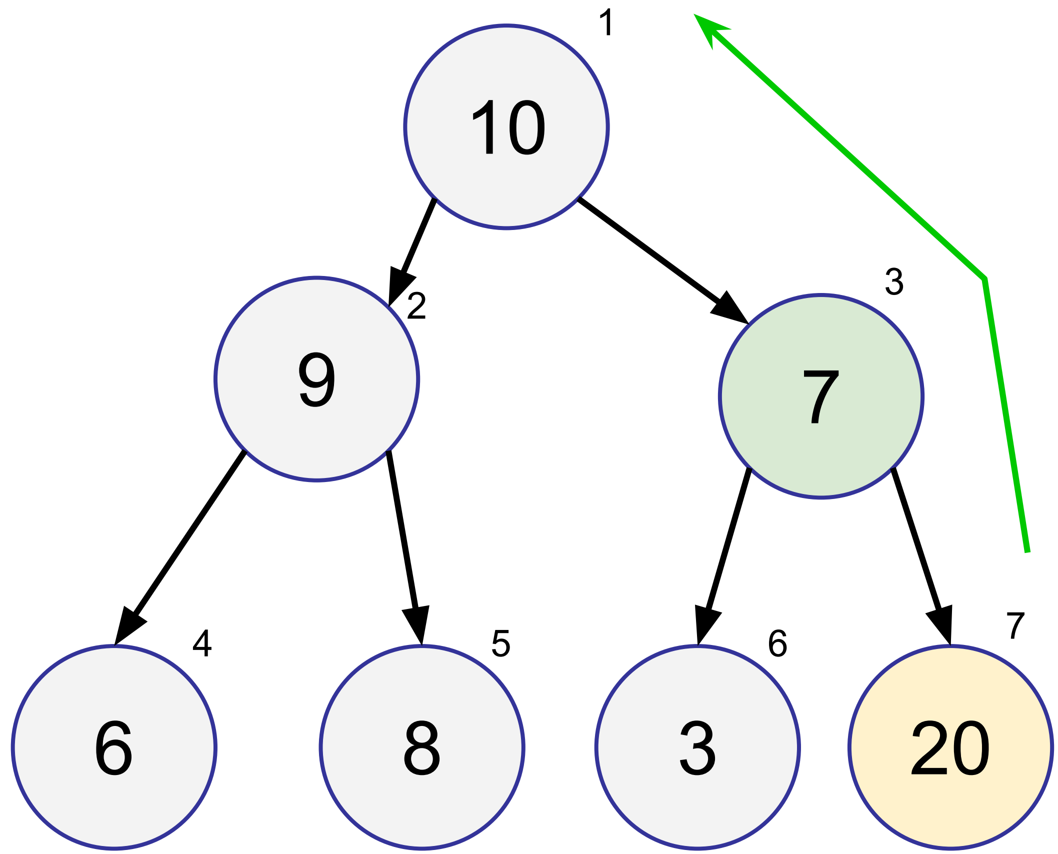
7	10	9	7	6	8	3	20
0	1	2	3	4	5	6	7

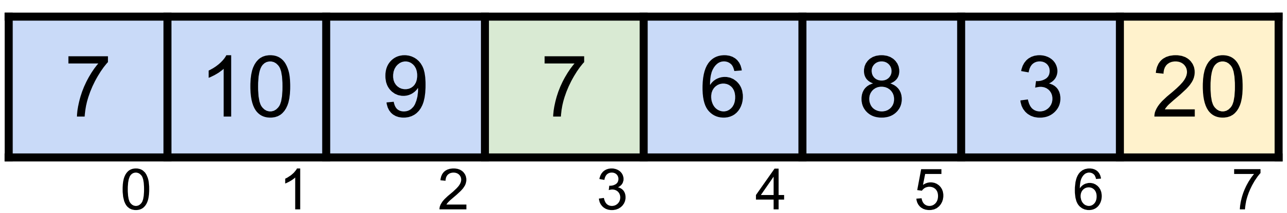
Binary Heaps:

Compare against the parent.

20 > 7

swap it!



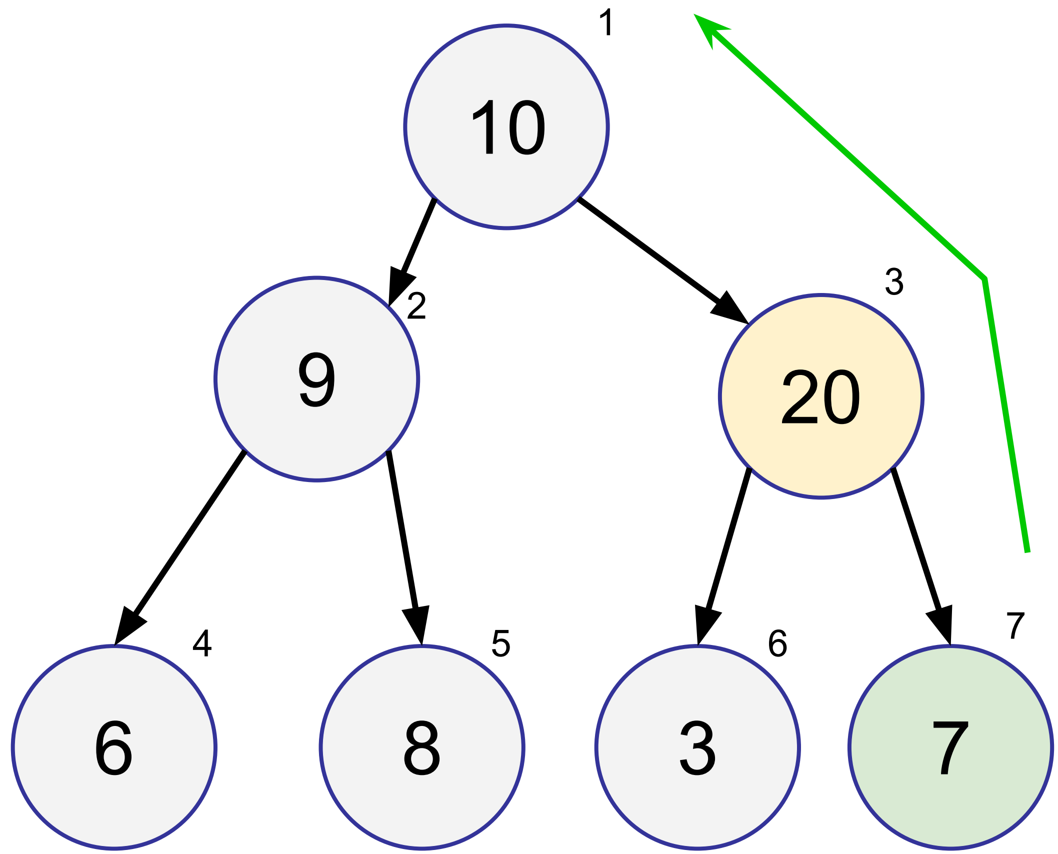


Binary Heaps:

Compare against the parent.

20 > 7

swap it!

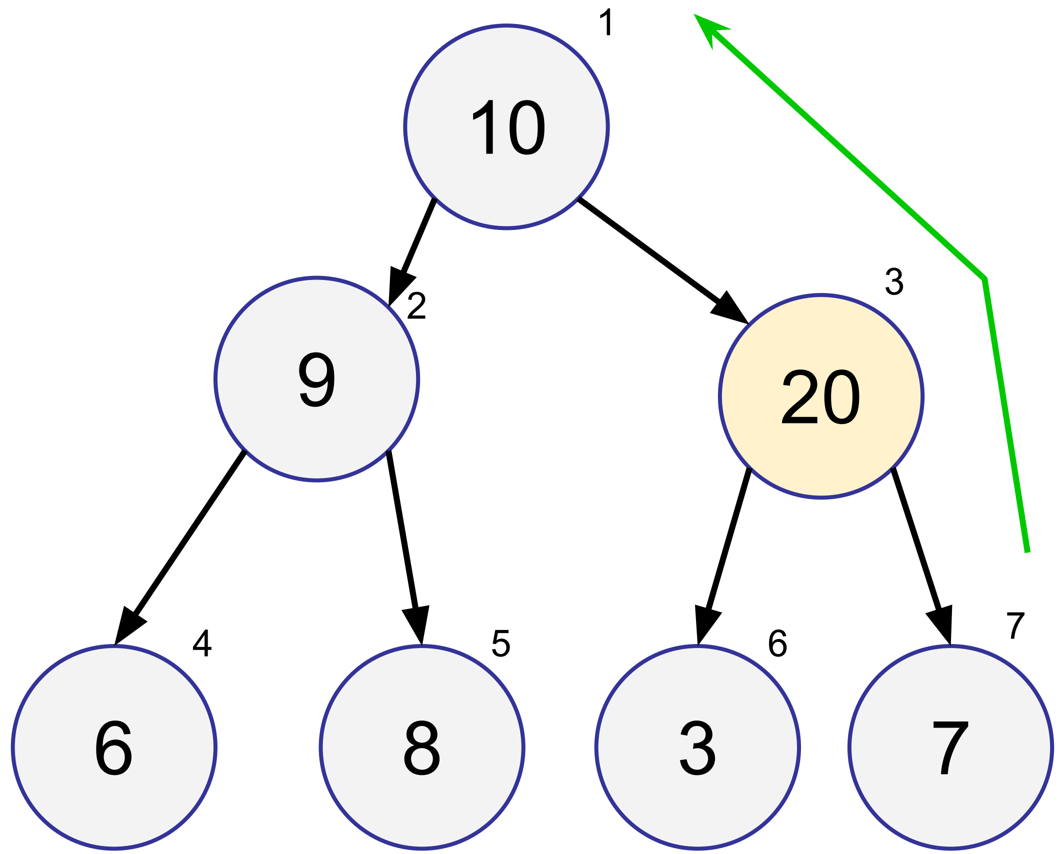


7	10	9	20	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

Repeat this until:

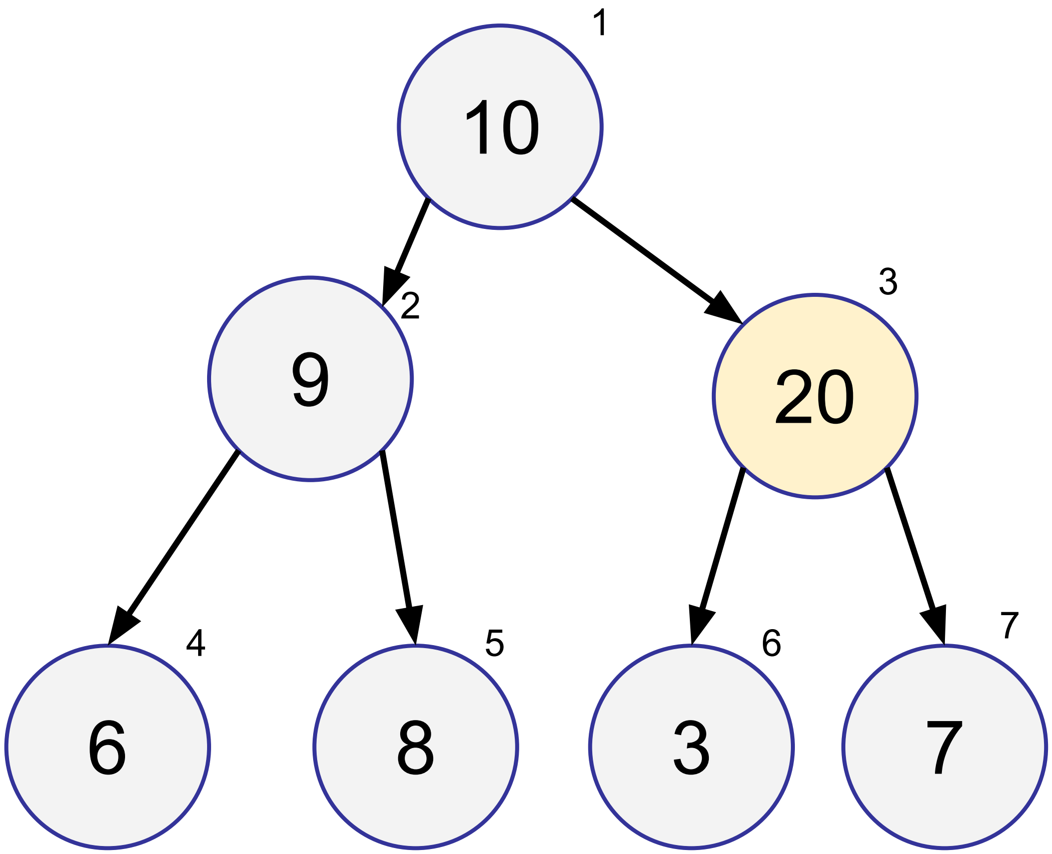
1. Either at root or
2. not larger than parent



7	10	9	20	6	8	3	7
0	1	2	3	4	5	6	7

Should we stop here or swap more?

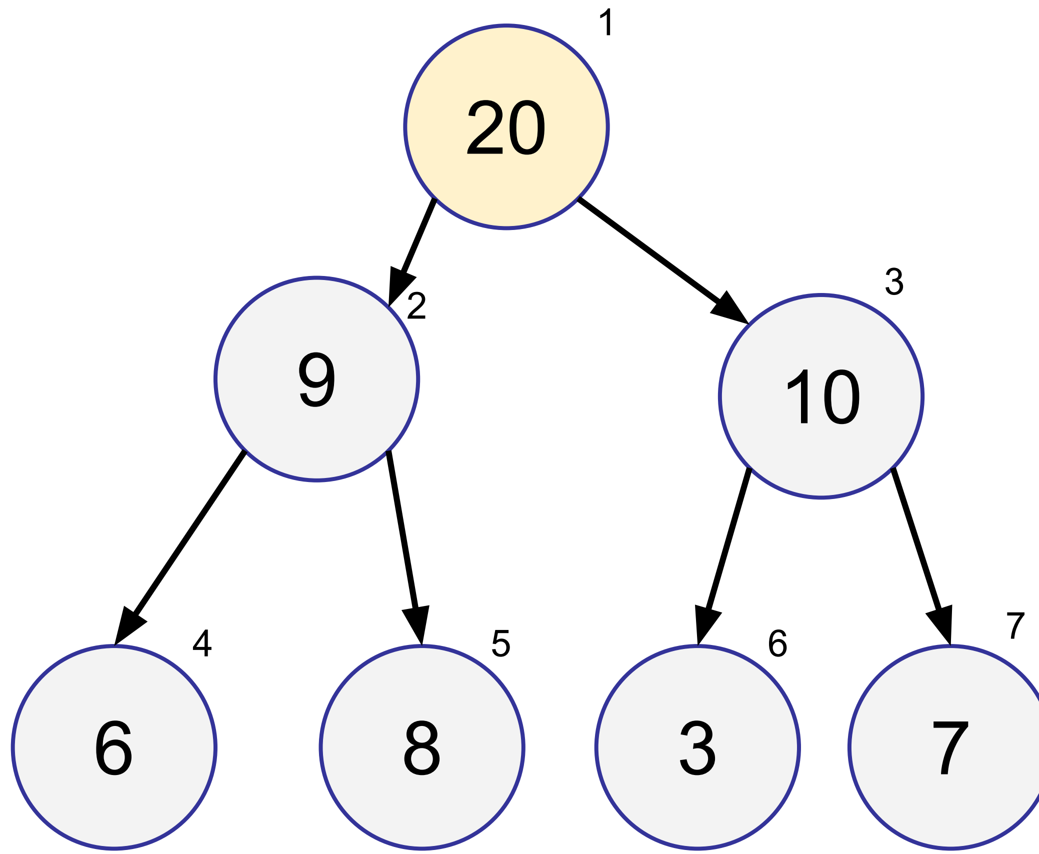
1. Swap more
2. Stop here
3. Yes



Binary Heaps:

Repeat this until:

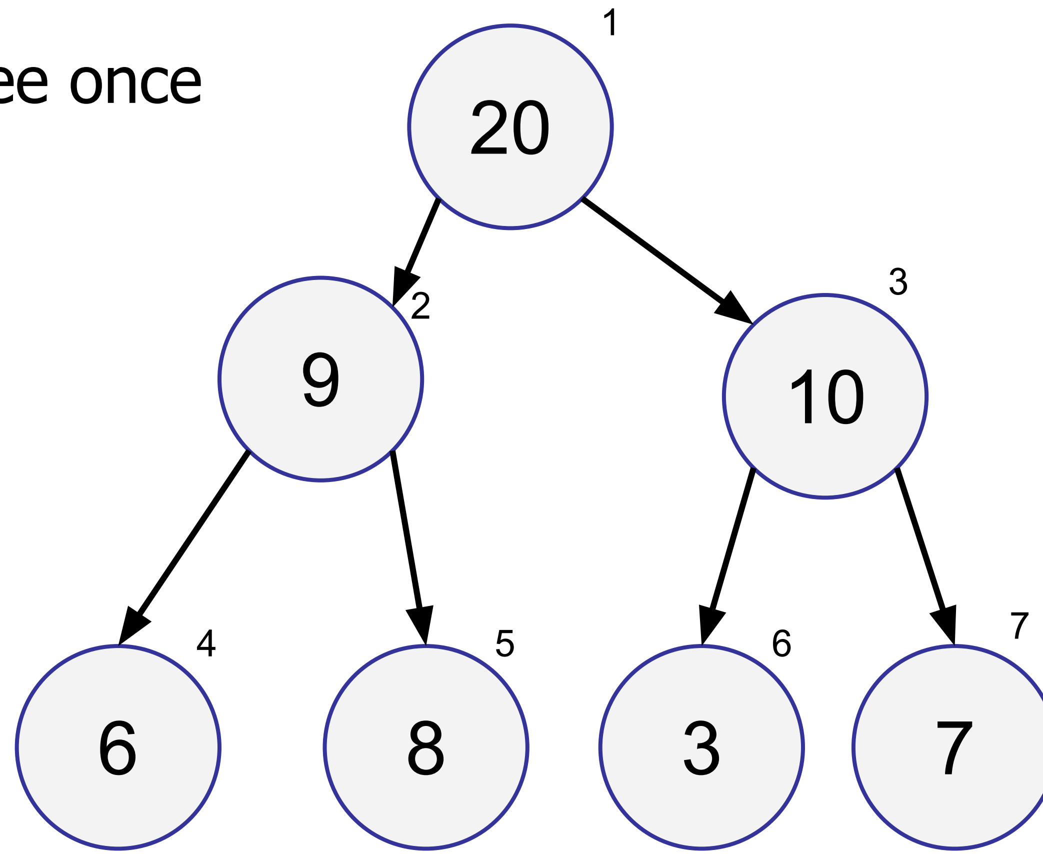
1. Either at root or
2. not larger than parent



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

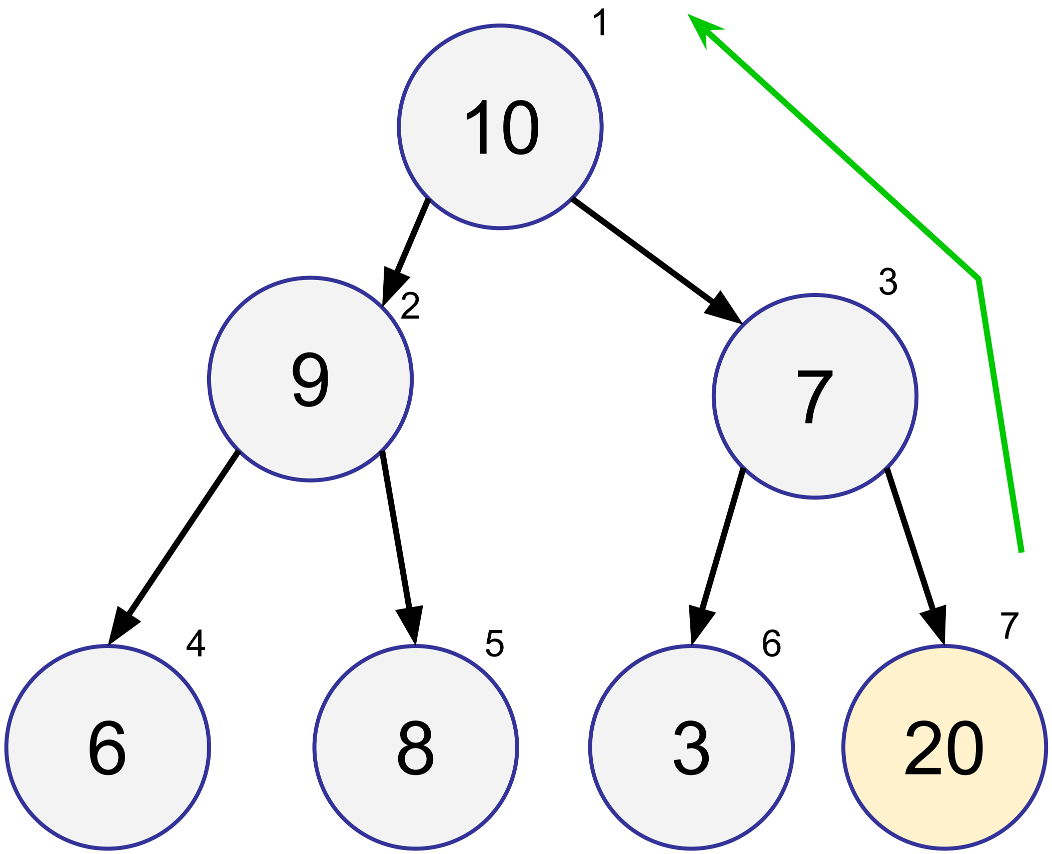
Since insert travels up the tree once
this costs O(log n) time.



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Is insert correct?

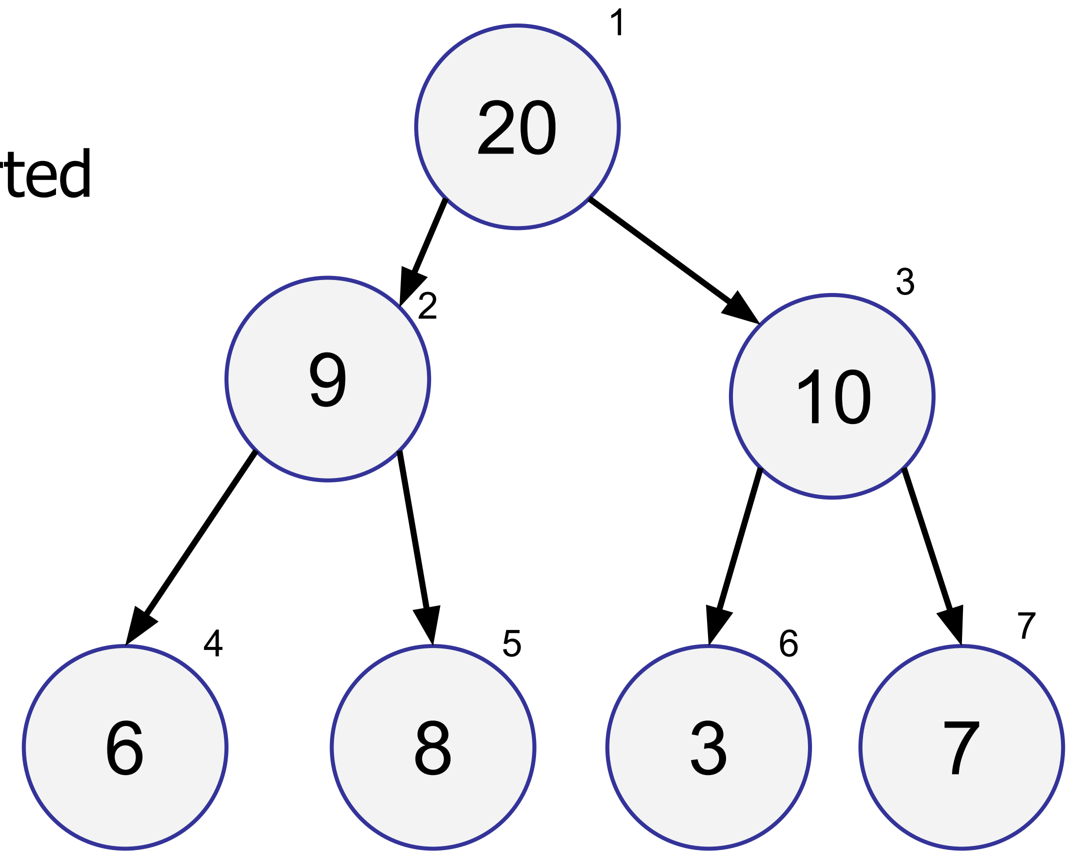
1. Yes
2. No
3. Yesn’t



Binary Heaps:

Is it possible that we need
to bubble the newly inserted
value back down?

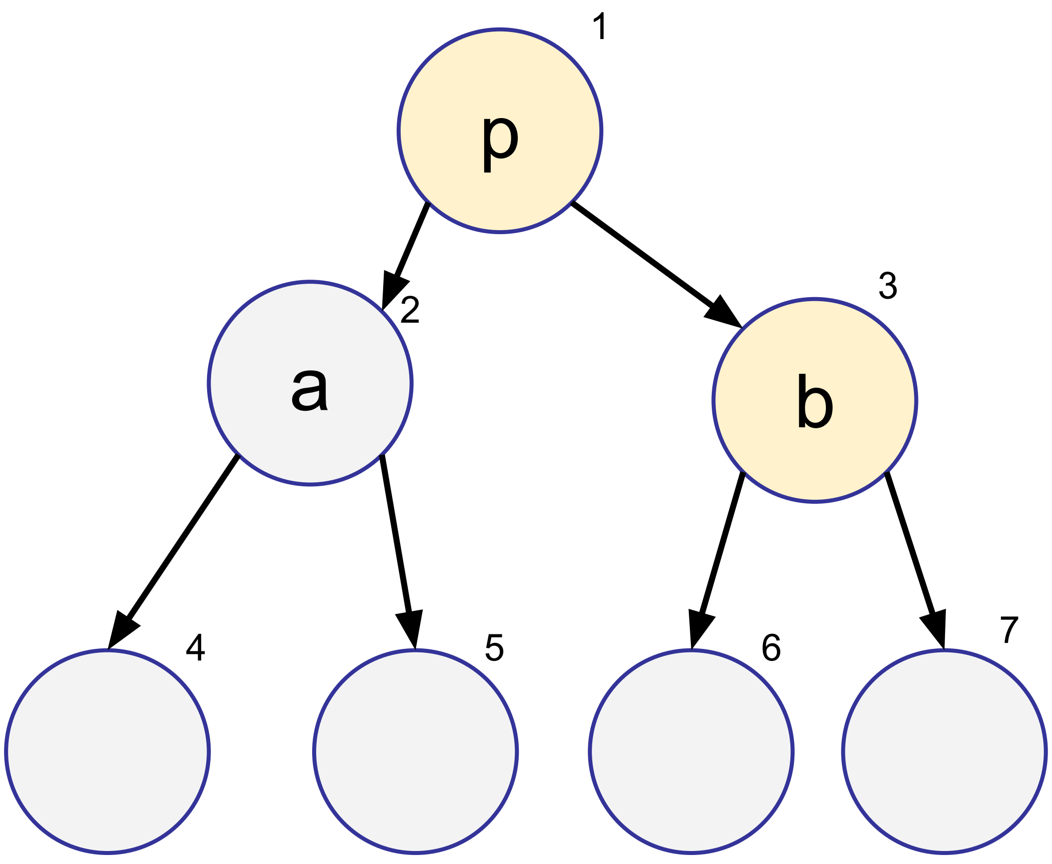
Why is bubbling up correct?



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

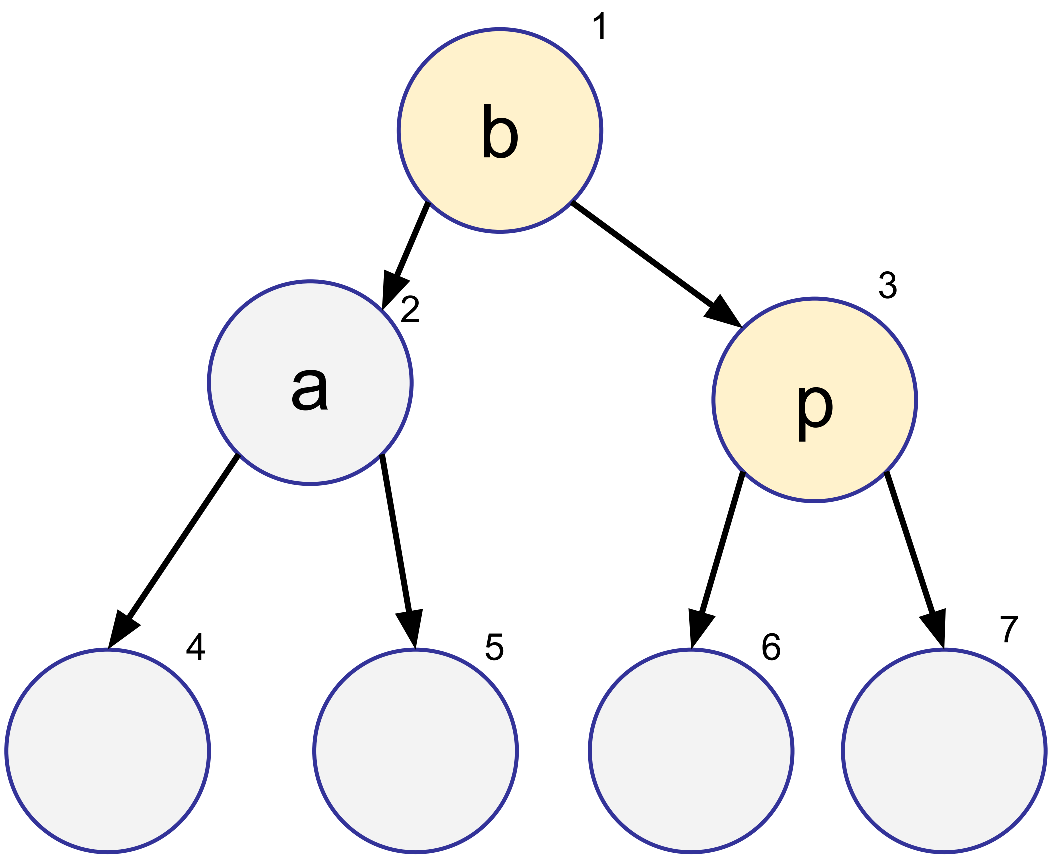
Say we had to bubble **b** up against **p**.



Binary Heaps:

Say we had to bubble **b** up against **p**.

Originally before insertion: p > a (due to our invariant)

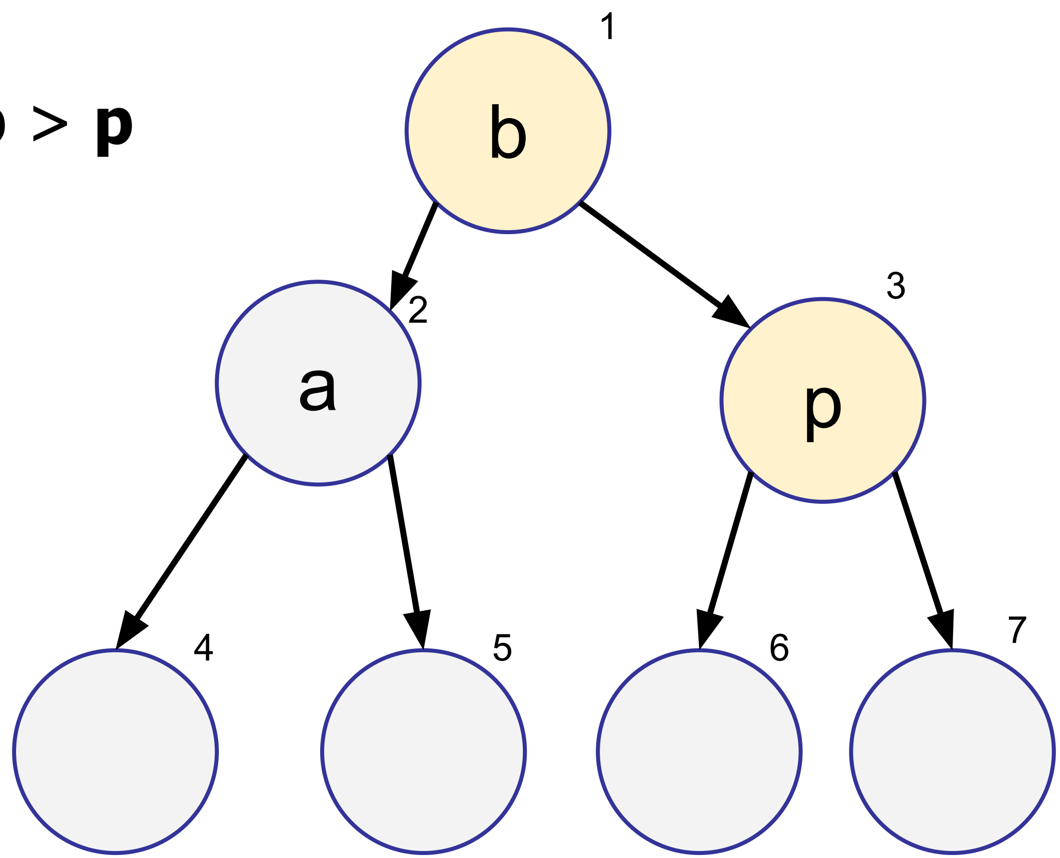


Binary Heaps:

Say we had to bubble **b** up against **p**.

Originally before insertion: **p** > **a** (due to our invariant)

bubbled **b** up because: **b** > **p**



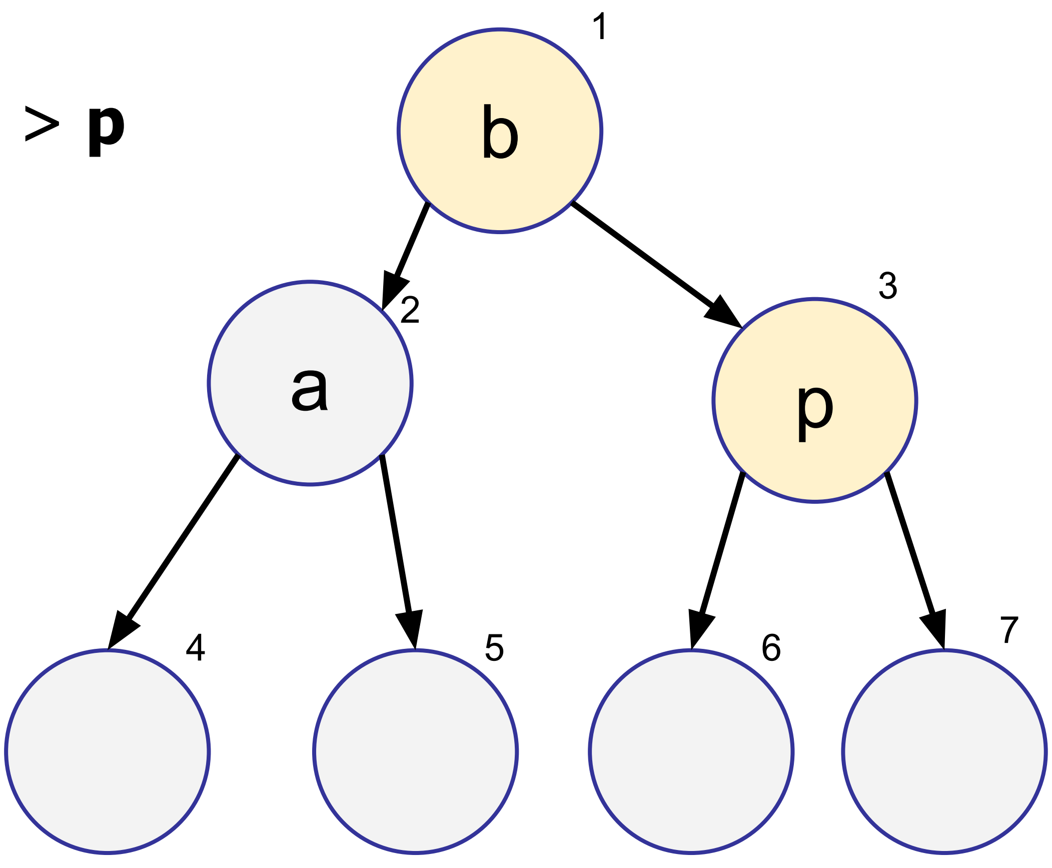
Binary Heaps:

Say we had to bubble **b** up against **p**.

Originally before insertion: **p** > **a** (due to our invariant)

bubbled **b** up because: **b** > **p**

So **b > p > a**



Binary Heaps:

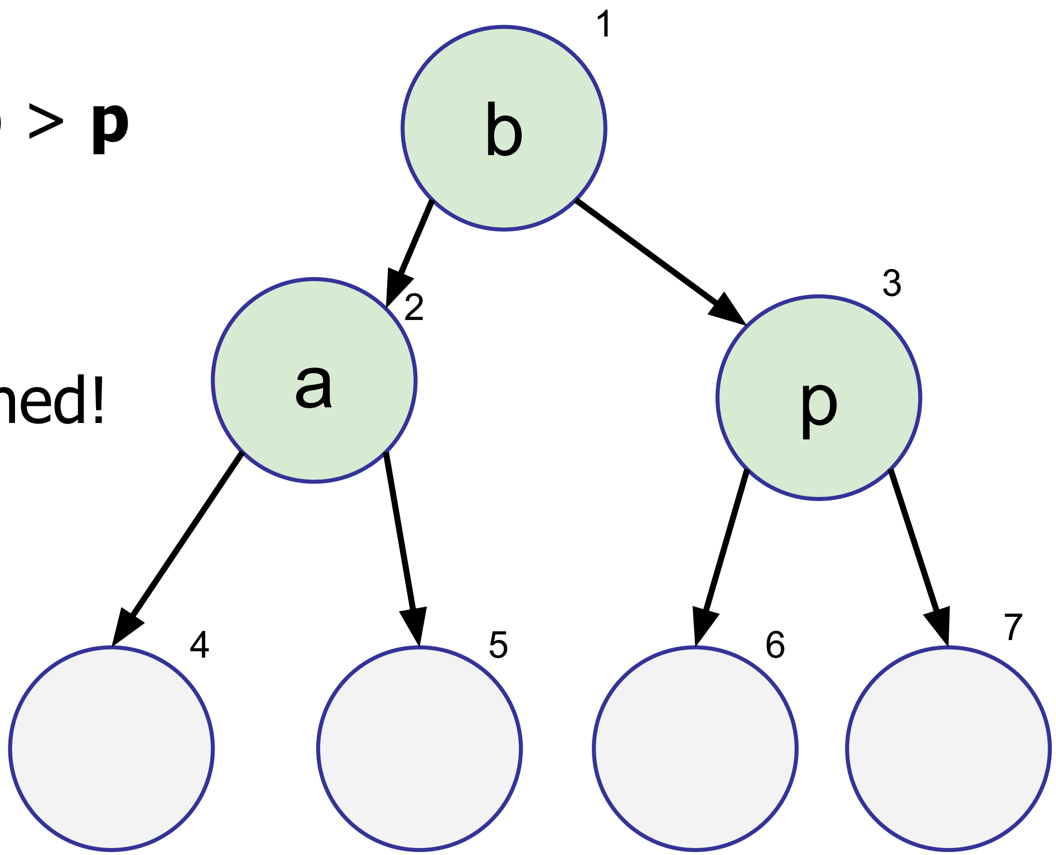
Say we had to bubble **b** up against **p**.

Originally before insertion: **p** > **a** (due to our invariant)

bubbled **b** up because: **b** > **p**

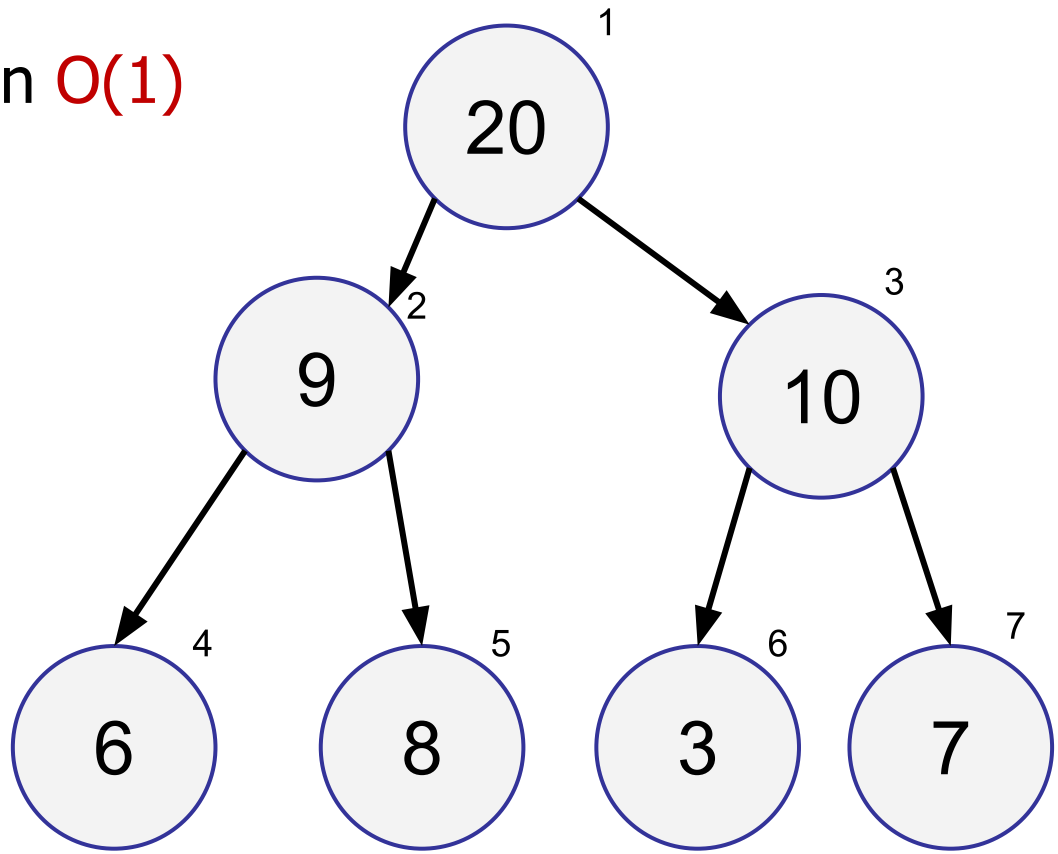
So **b > p > a**

Heap ordering is maintained!



Binary Heaps:

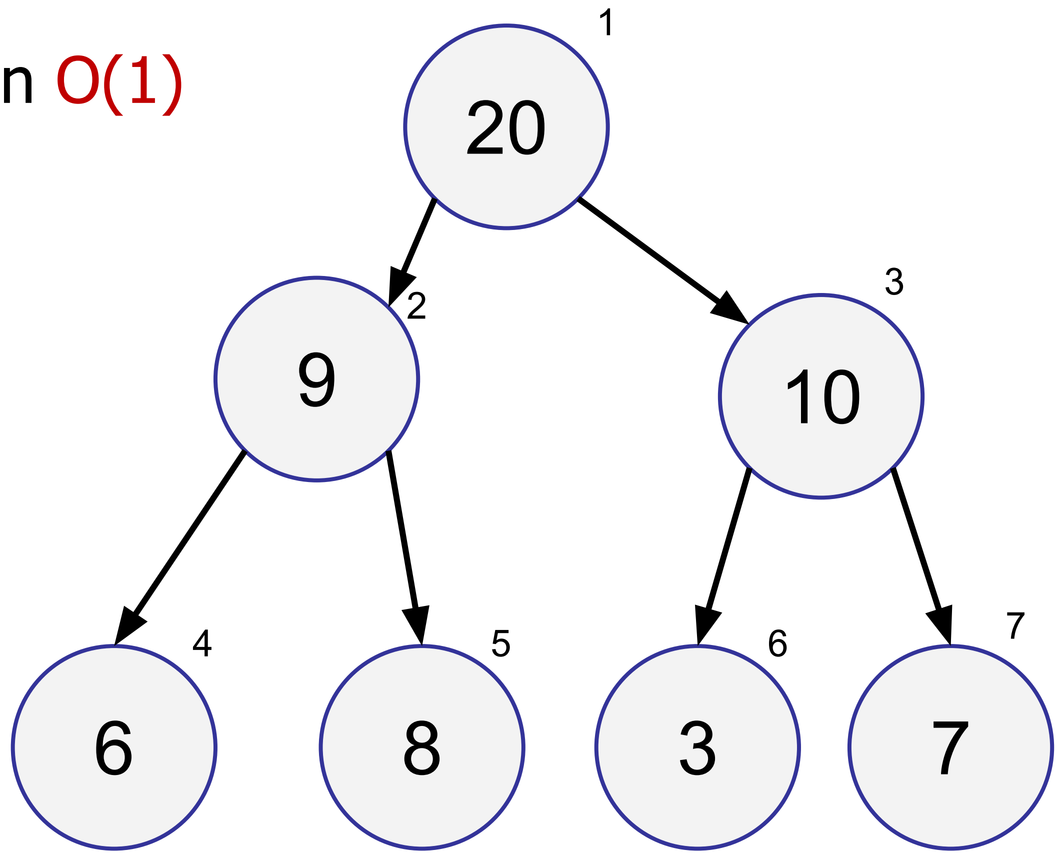
How do we peekMaxPriority in O(1) time?



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

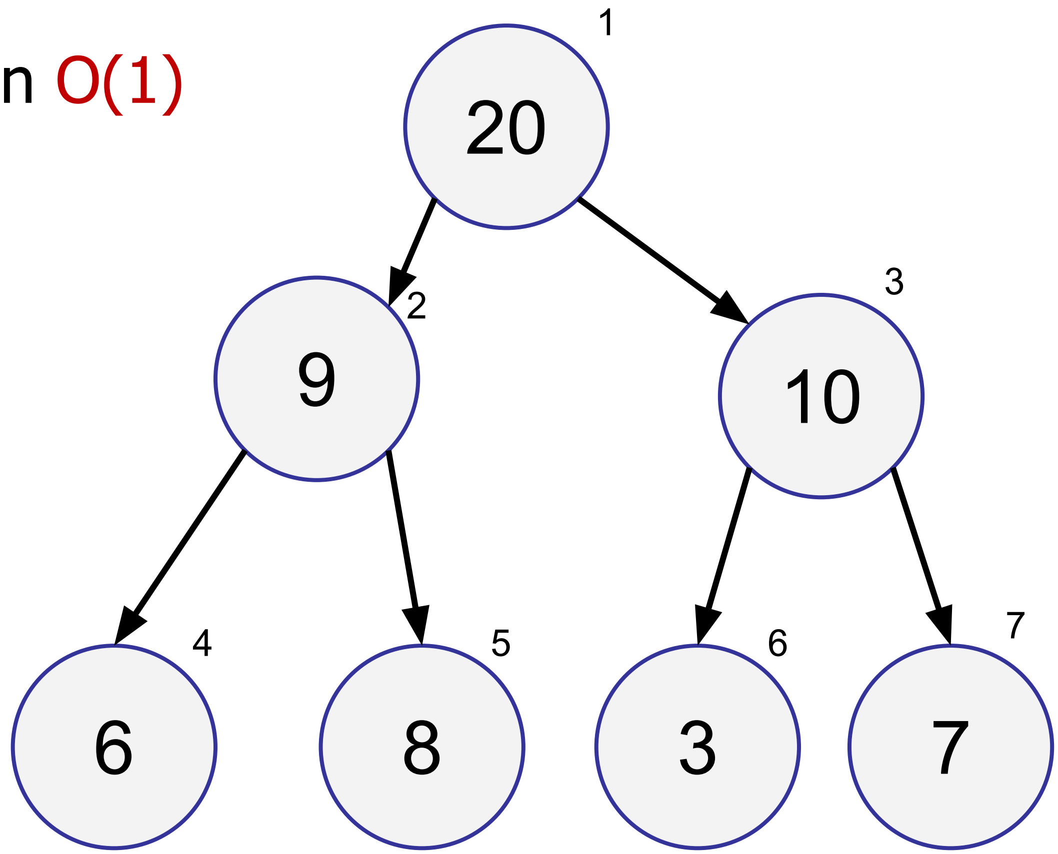
How do we peekMaxPriority in O(1) time?



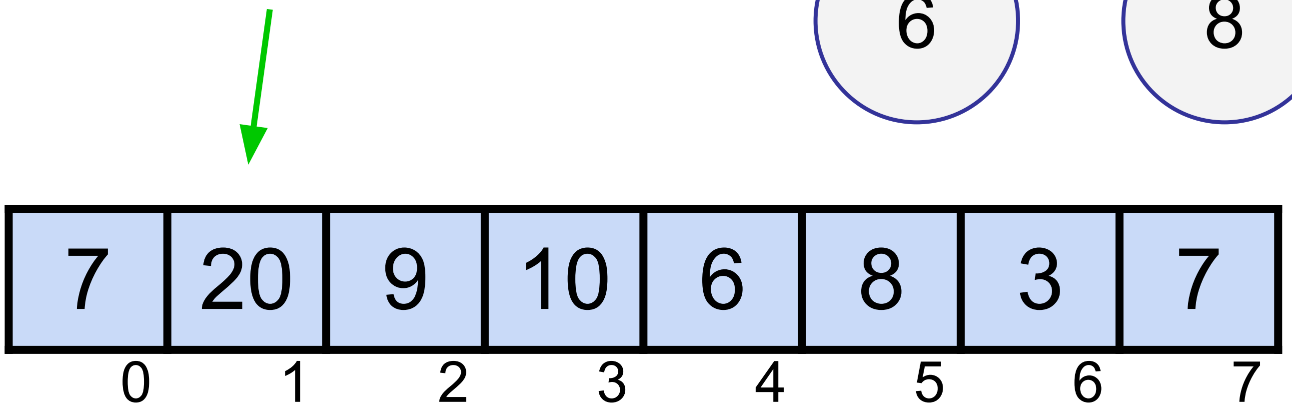
7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

How do we peekMaxPriority in O(1) time?



Return `arr[1]`

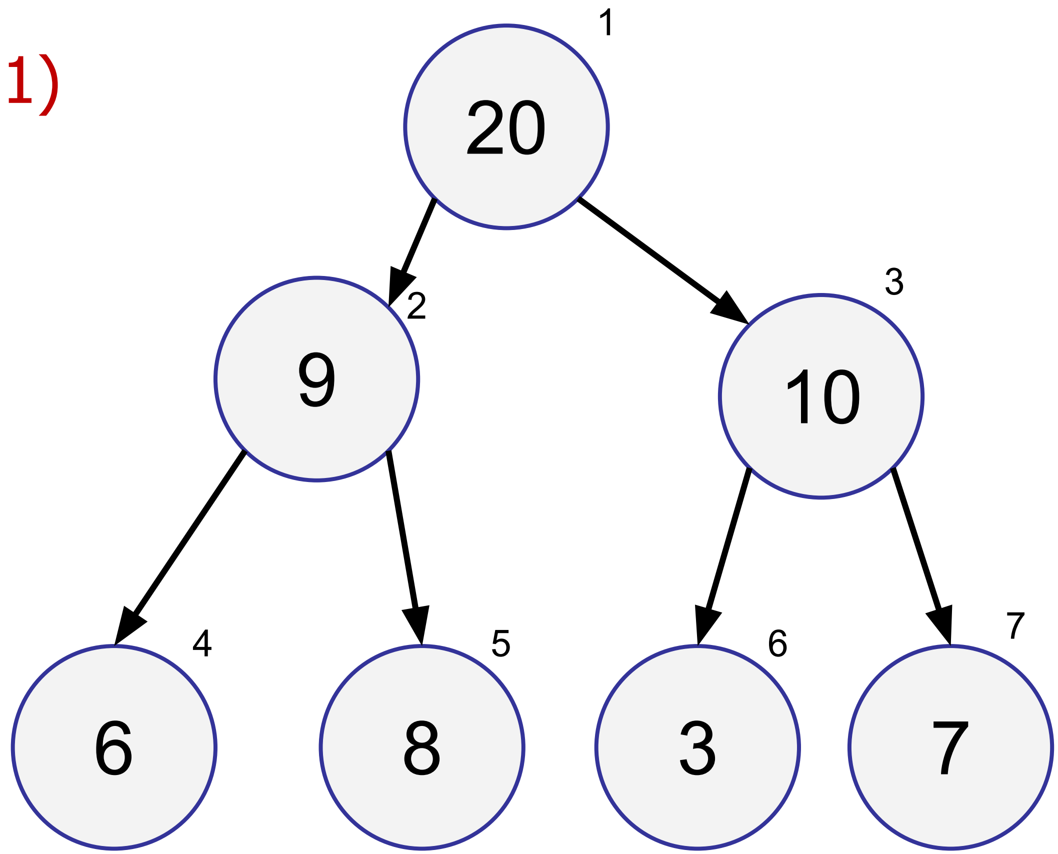


Binary Heaps:

How do we do contains in O(1) expected time?

Use a hashtable*not drawn here

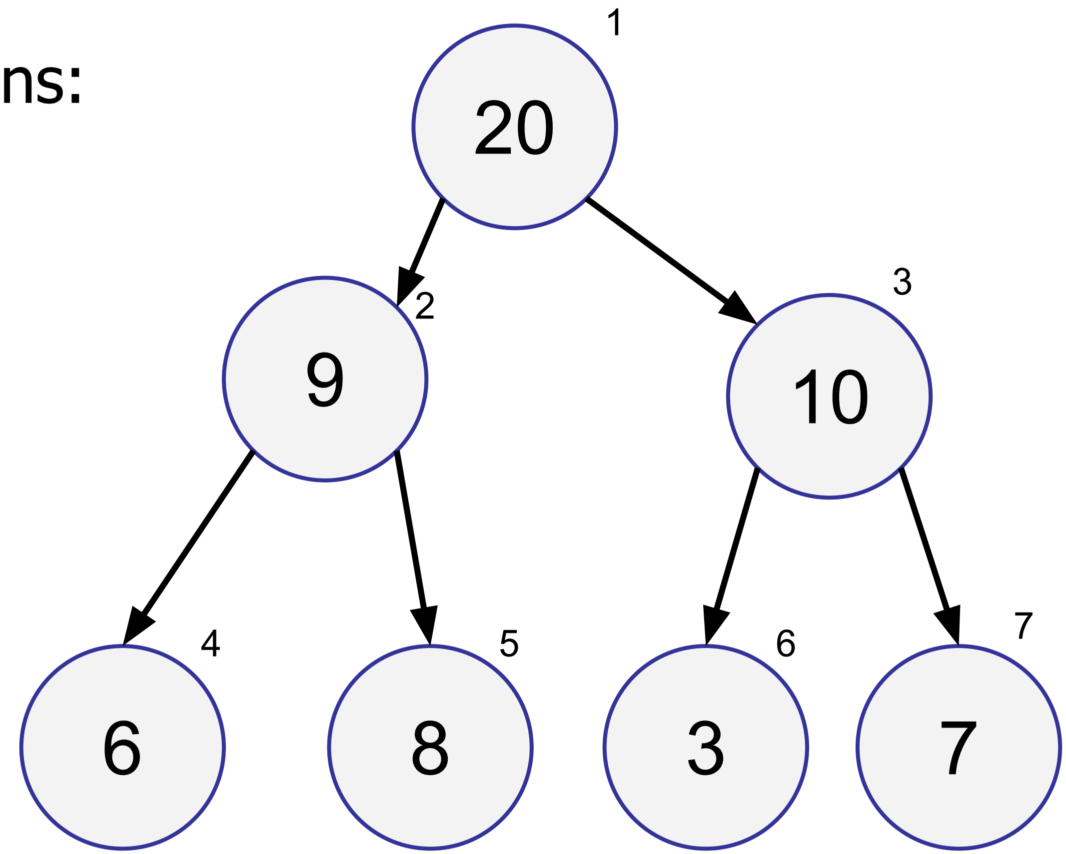
that stores the IDs



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

The only interesting operations:
increaseKey/decreaseKey in
O(log n) time.

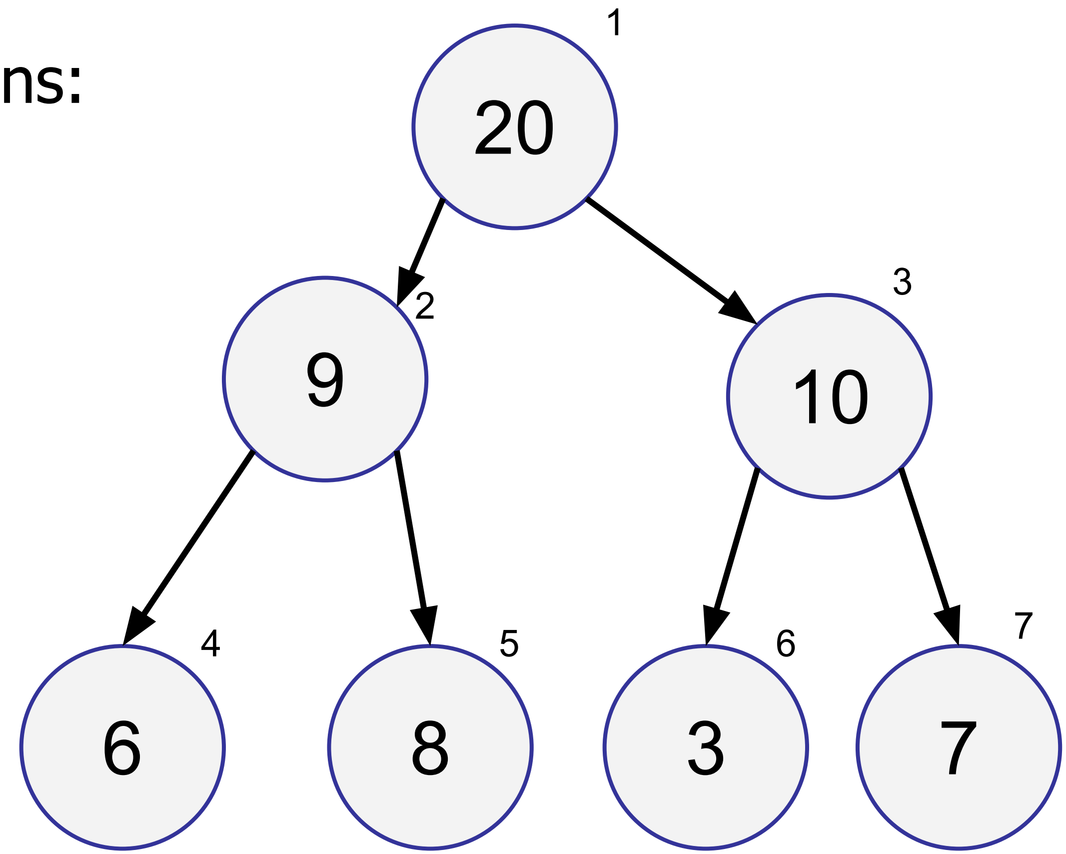


7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

The only interesting operations:
increaseKey/decreaseKey in
O(log n) time.

This needs us to map IDs
to nodes.

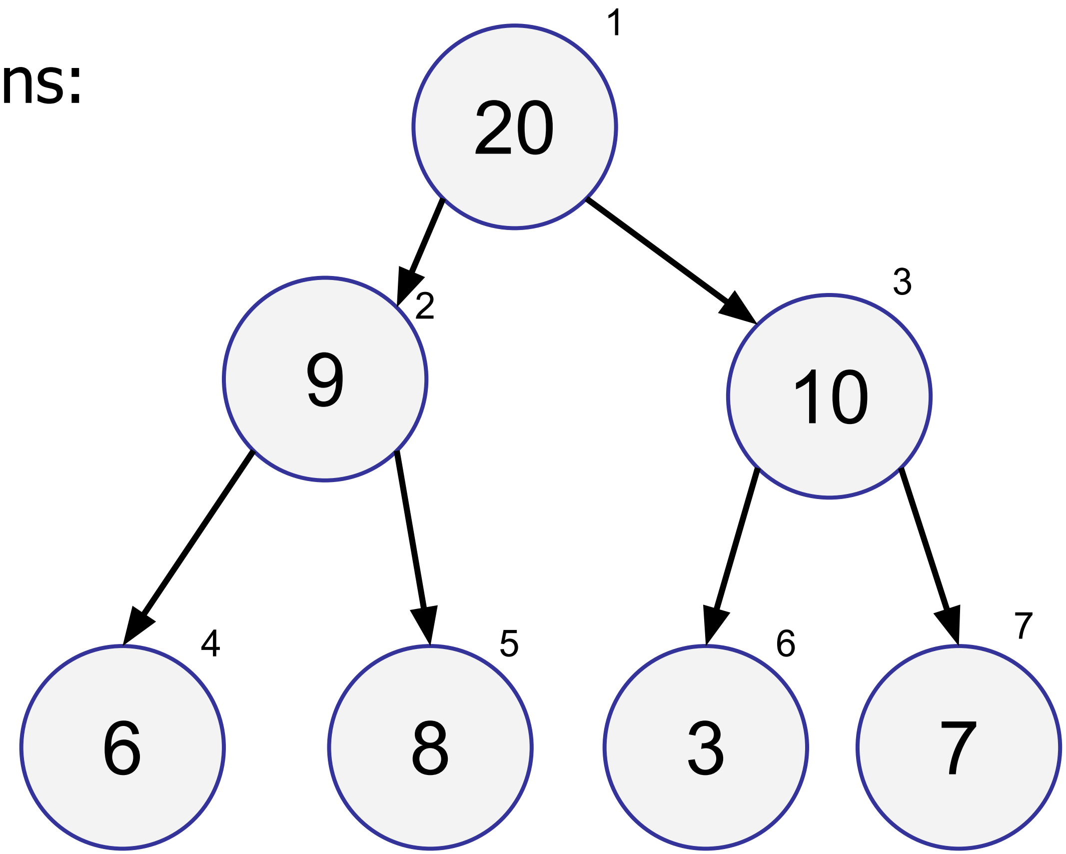


7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

The only interesting operations:
increaseKey/decreaseKey in
O(log n) time.

This needs us to map IDs
to nodes.



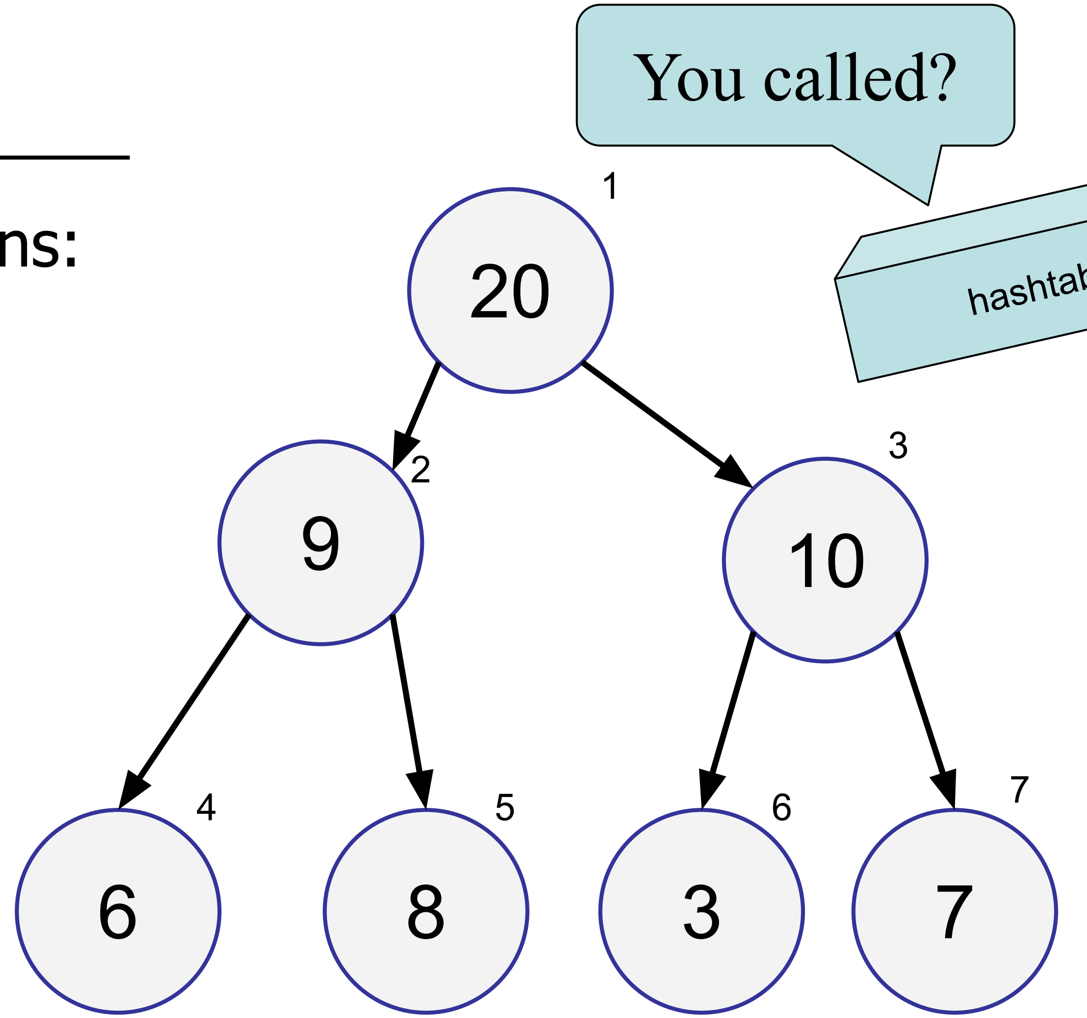
7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

The only interesting operations: increaseKey/decreaseKey in O(log n) time.

This needs us to map IDs to nodes.

Would be nice we knew the index of the node for a given ID.

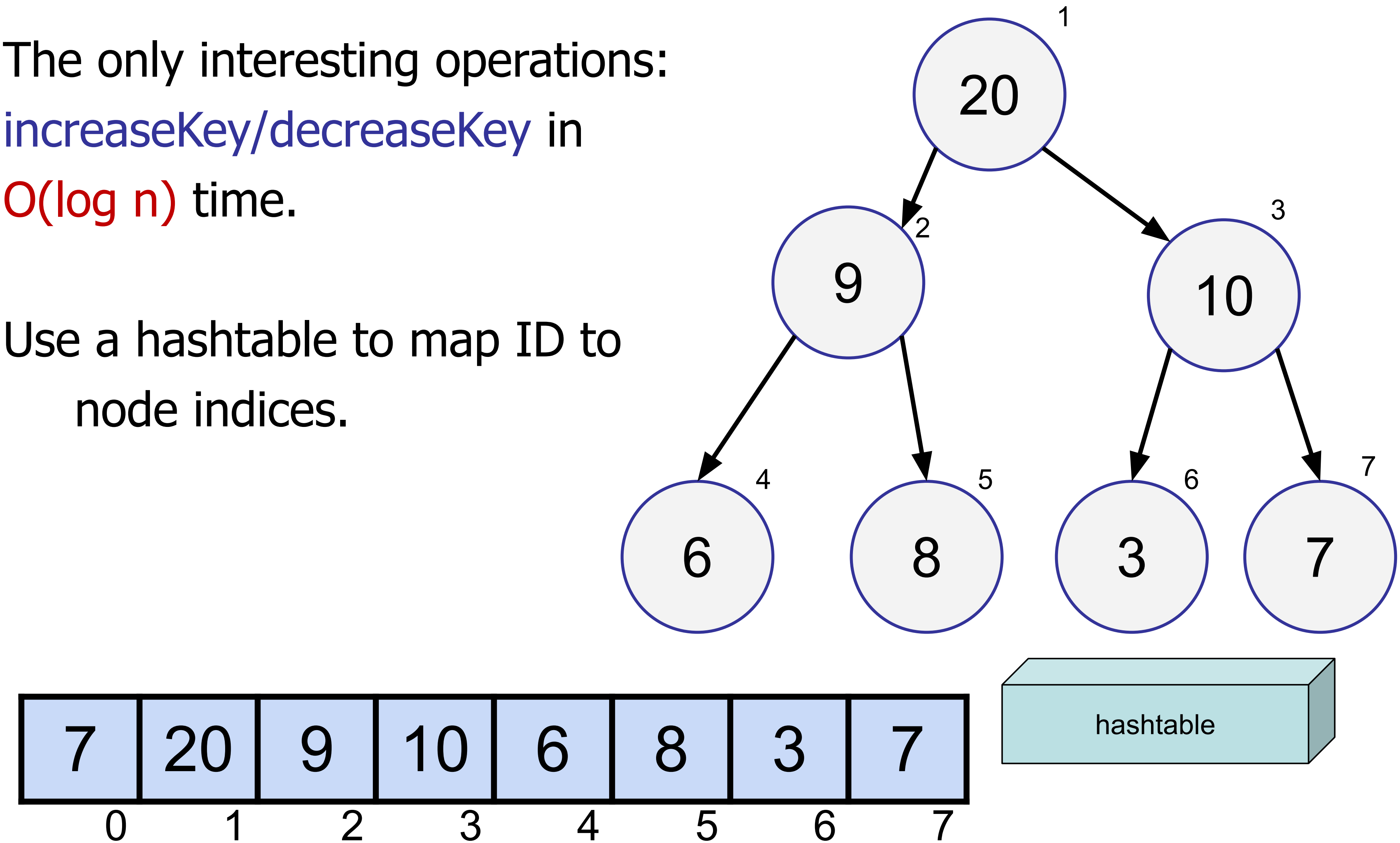


7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

Binary Heaps:

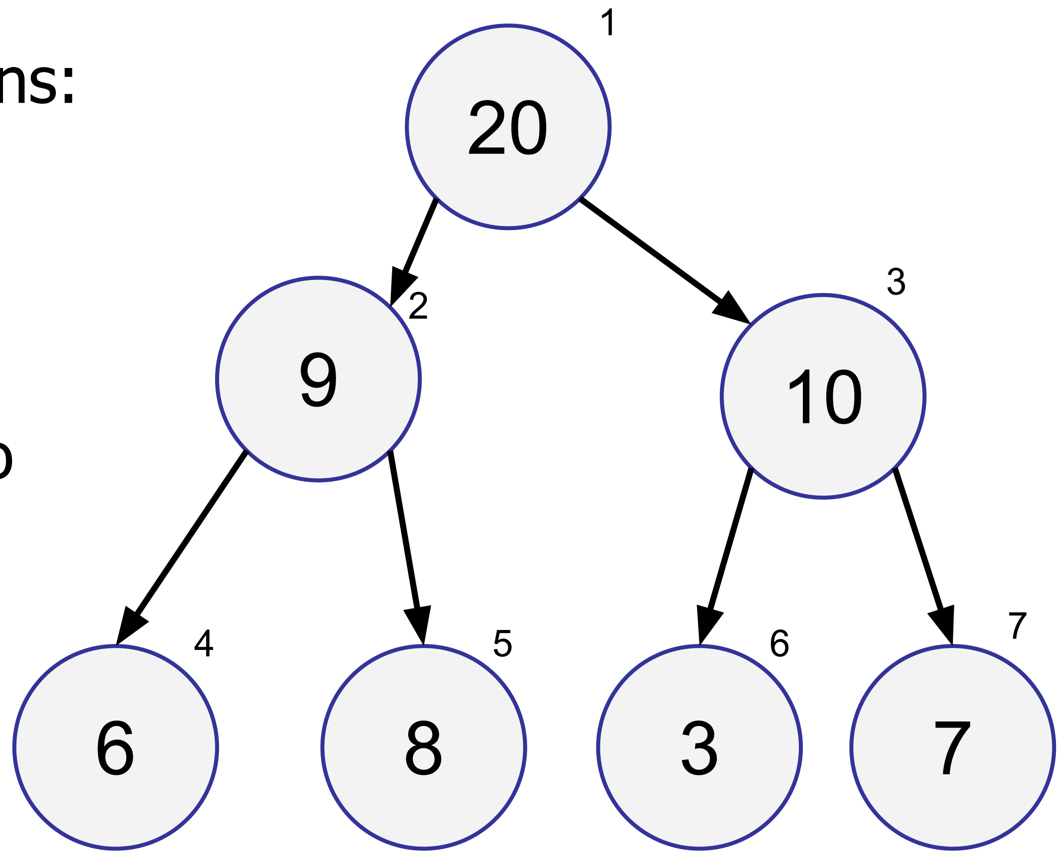
The only interesting operations:
increaseKey/decreaseKey in
O(log n) time.

Use a hashtable to map ID to
node indices.



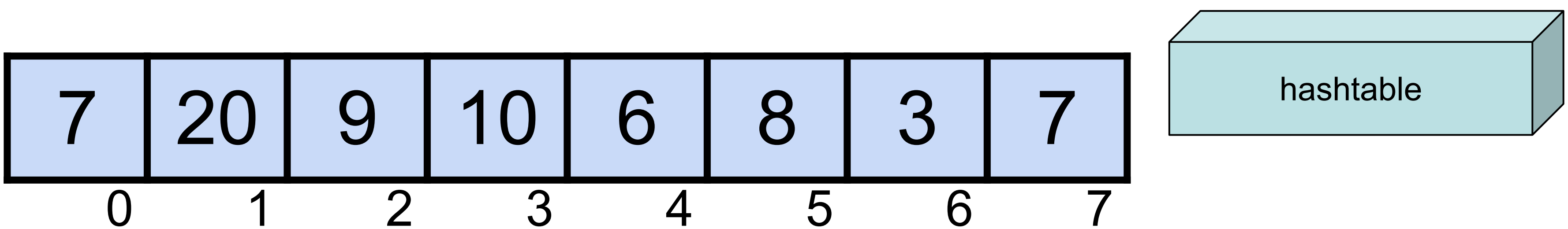
Binary Heaps:

The only interesting operations:
increaseKey/decreaseKey in
O(log n) time.



Use a hashtable to map ID to node indices.

E.g. ID 61 has priority 8

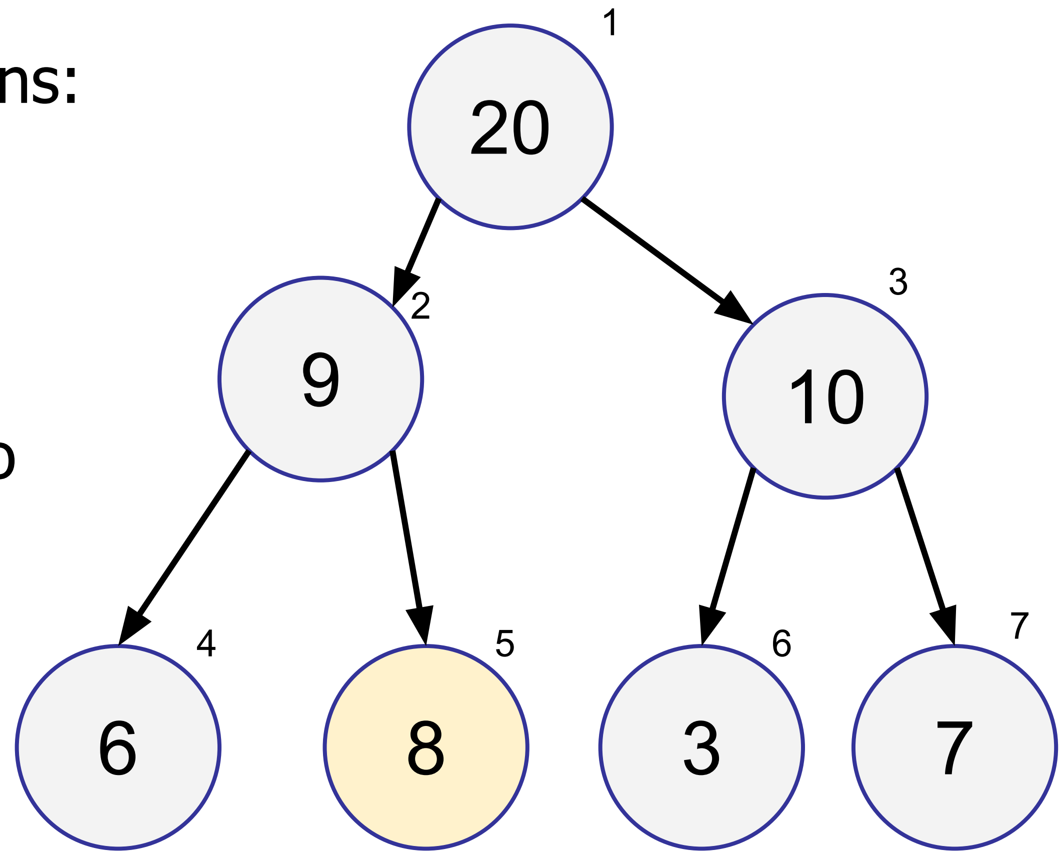


Binary Heaps:

The only interesting operations:

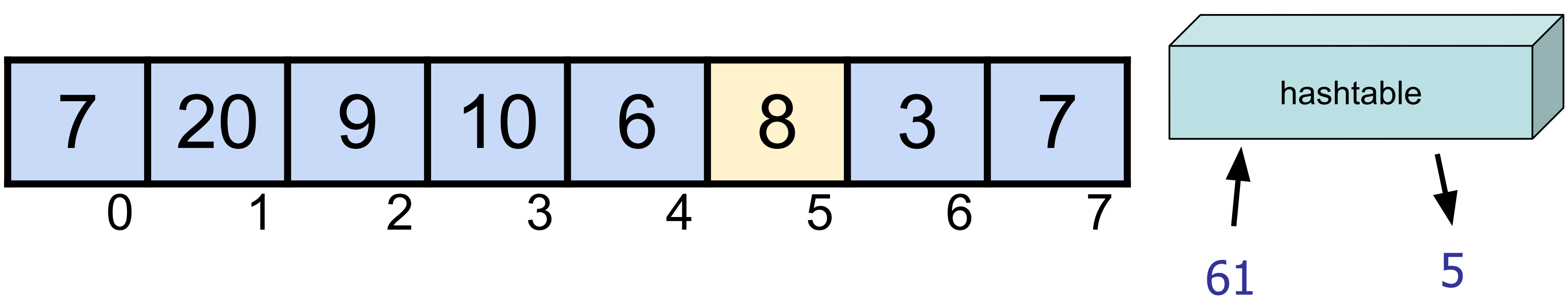
increaseKey/decreaseKey in

O(log n) time.



Use a hashtable to map ID to
node indices.

E.g. ID 61 has priority 8



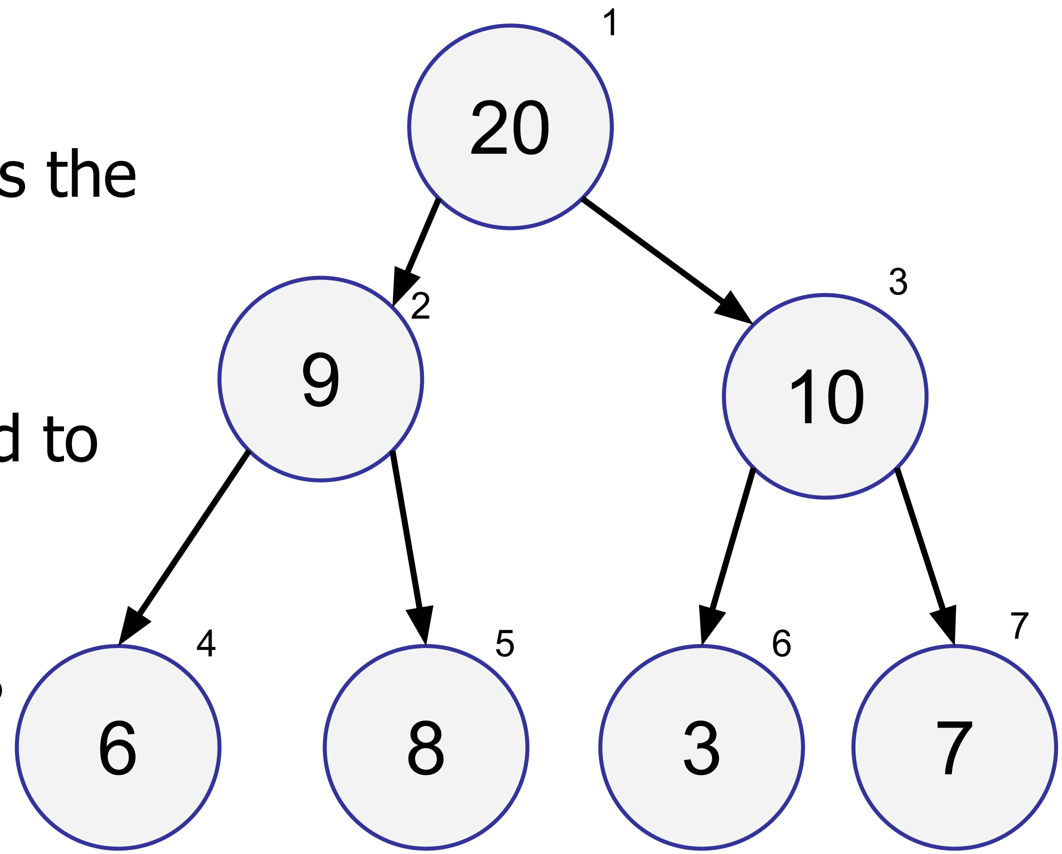
Binary Heaps:

Question:

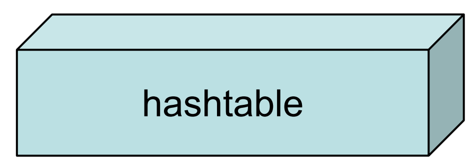
Let's say the array only stores the priorities.

What happens when we need to swap nodes around?

What do we need to update?



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

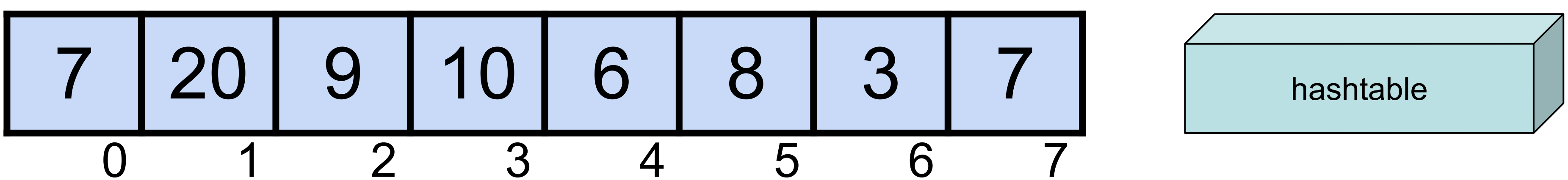


When we swap nodes around, we only need to update:

1. The array that stores only priorities

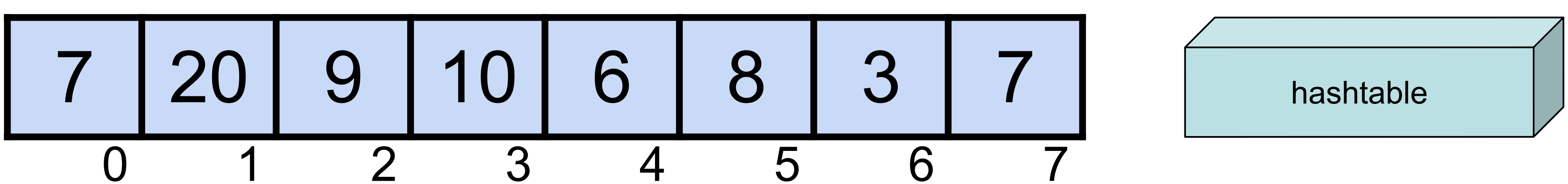
2. The hashtable

3. Both the array and the hashtable



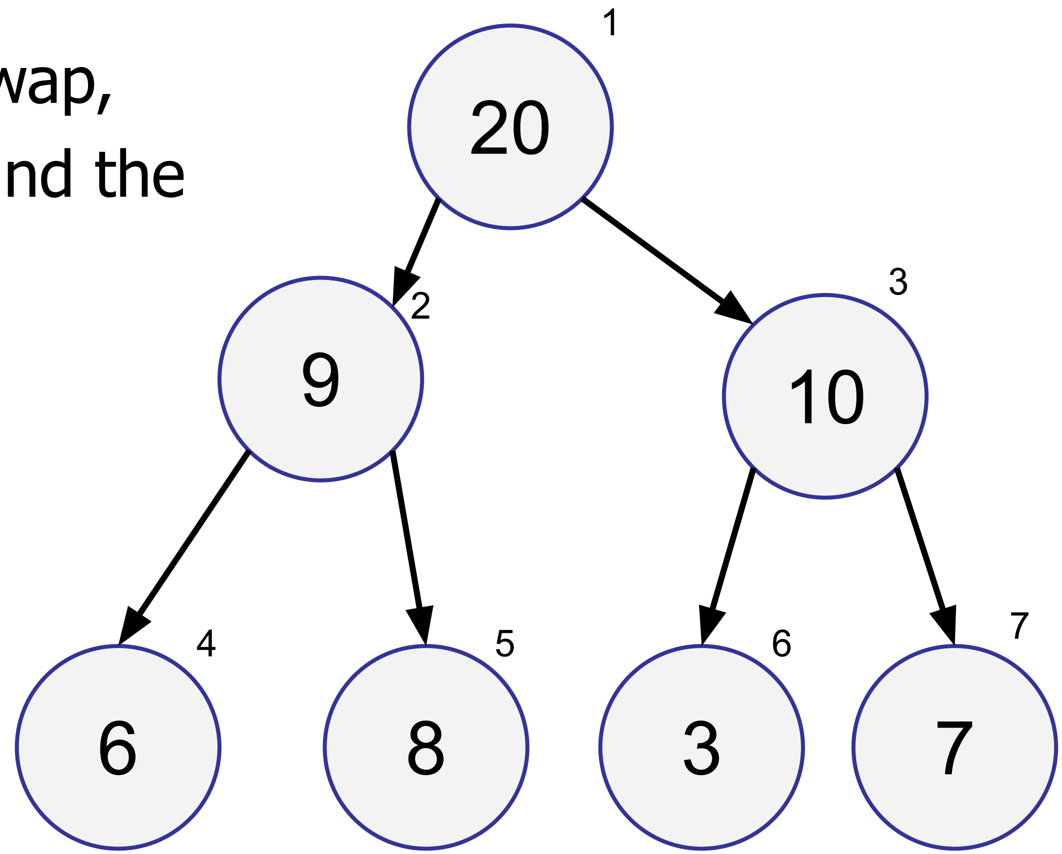
When we swap nodes around, we <u>only</u> need to update:

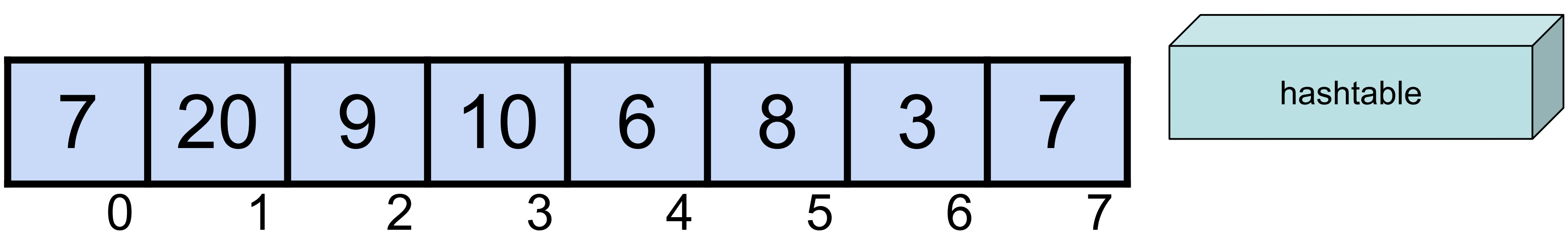
- The array that stores only priorities
- The hashtable
- Both the array and the hashtable



Binary Heaps:

Answer: When we want to swap, we only know the index and the priority of our parent.

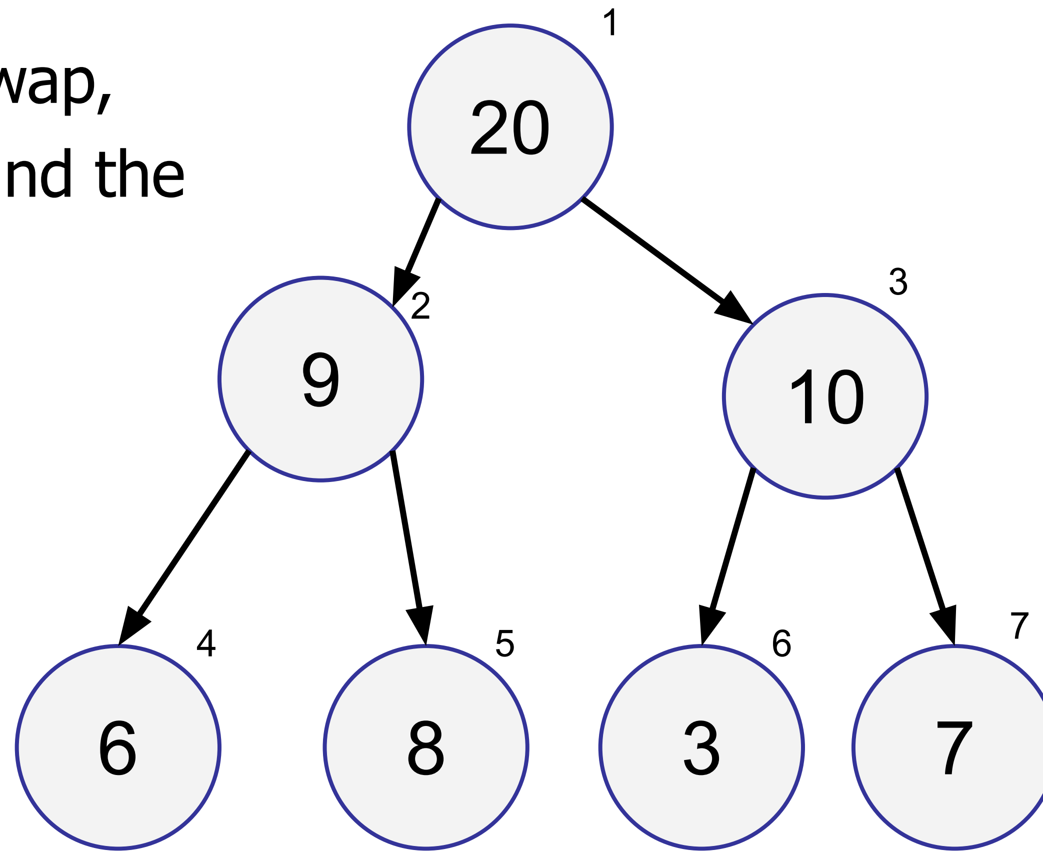


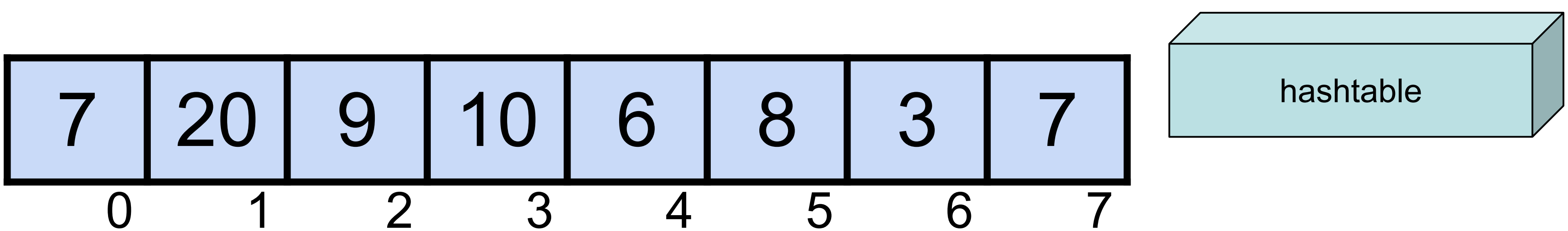


Binary Heaps:

Answer: When we want to swap, we only know the index and the priority of our parent.

But the hashtable only maps IDs to indices.



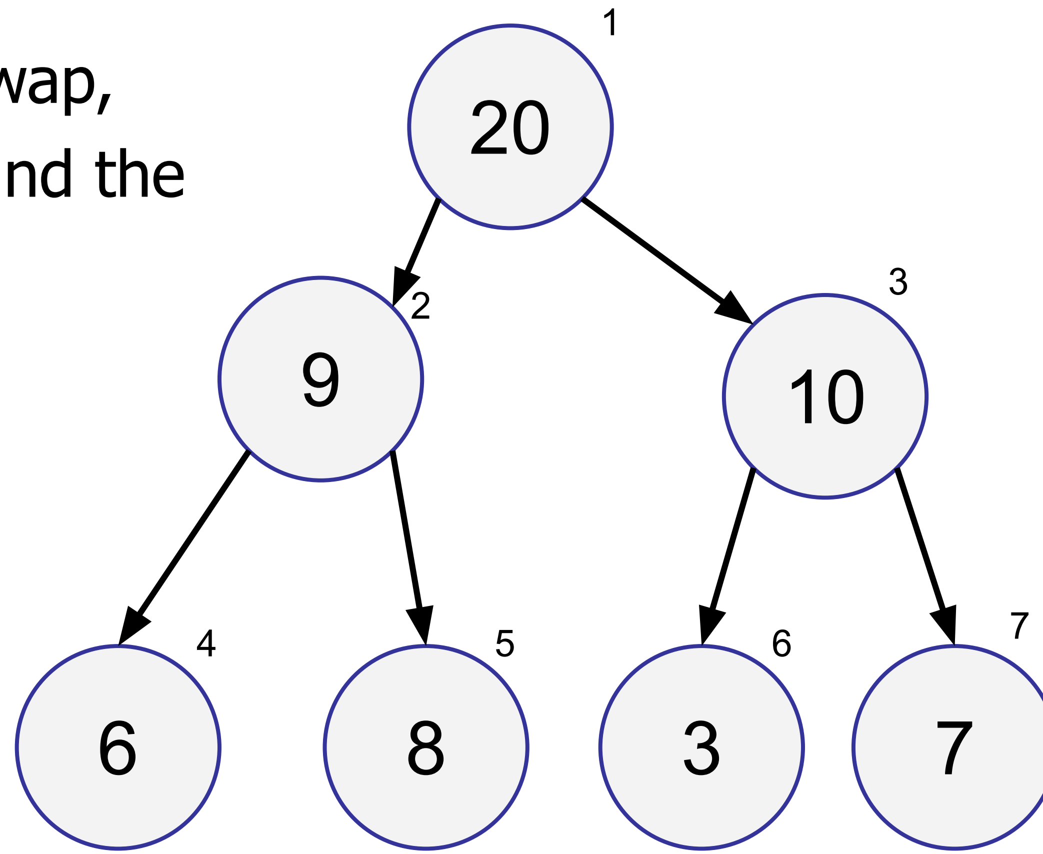


Binary Heaps:

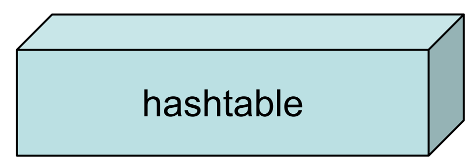
Answer: When we want to swap, we only know the index and the priority of our parent.

But the hashtable only maps IDs to indices.

We need to map indices to IDs



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

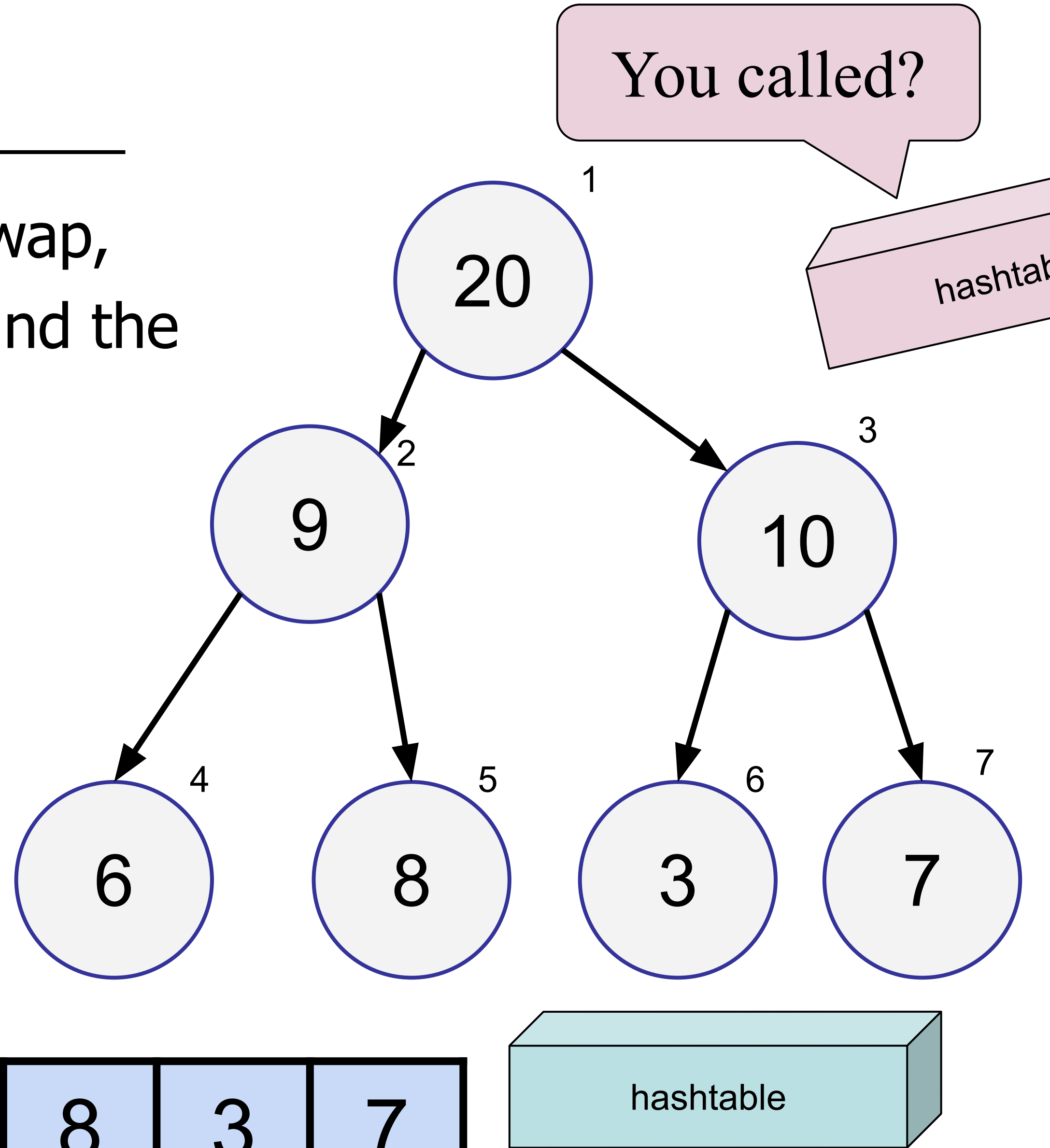


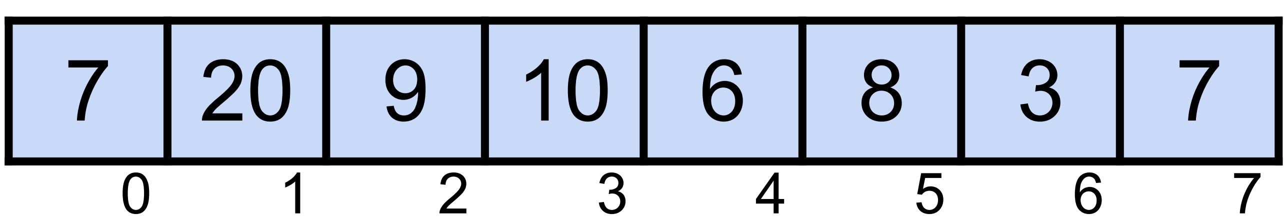
Binary Heaps:

Answer: When we want to swap, we only know the index and the priority of our parent.

But the hashtable only maps IDs to indices.

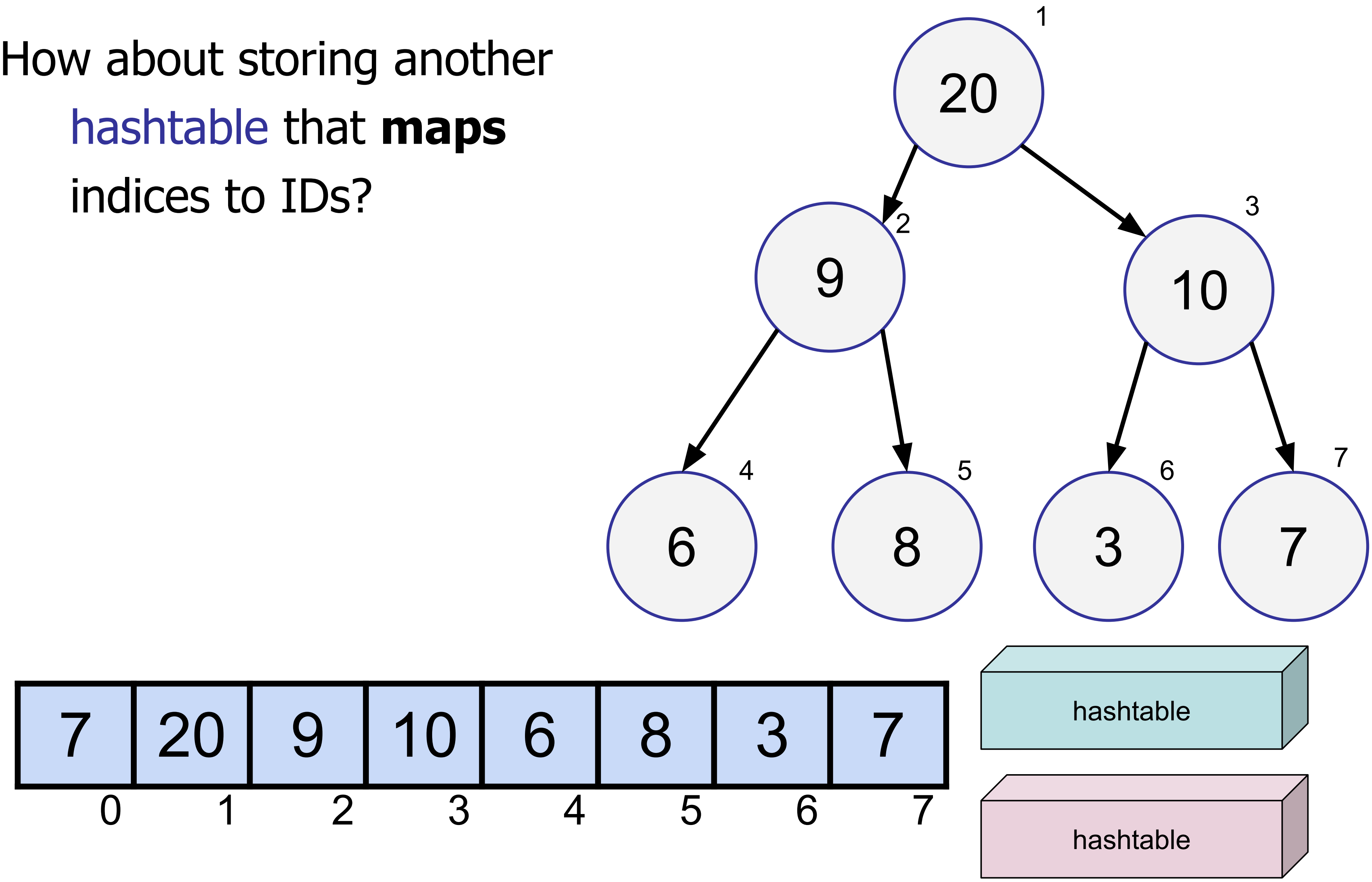
We need to map indices to IDs





Binary Heaps:

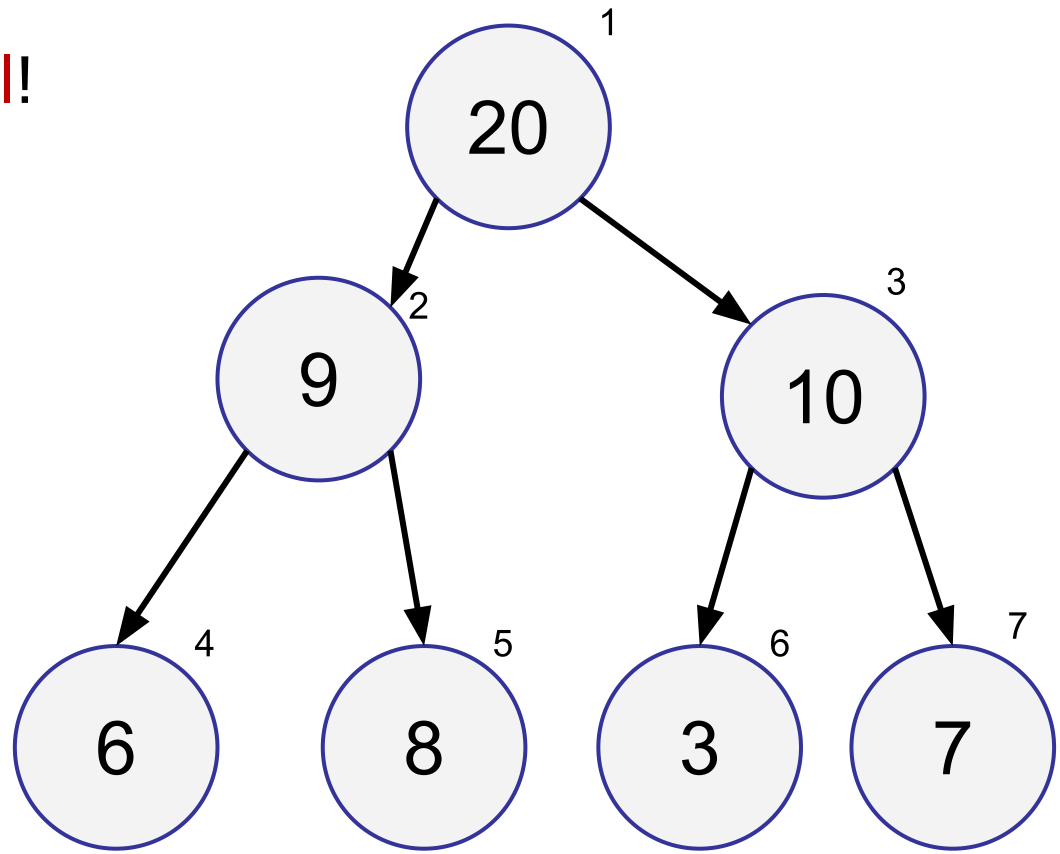
How about storing another
hashtable that **maps**
indices to IDs?



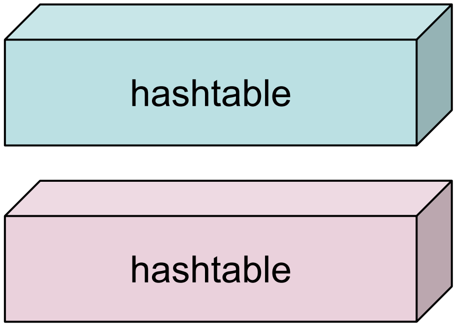
Binary Heaps:

We could. But this is wasteful!

Can we use something else?

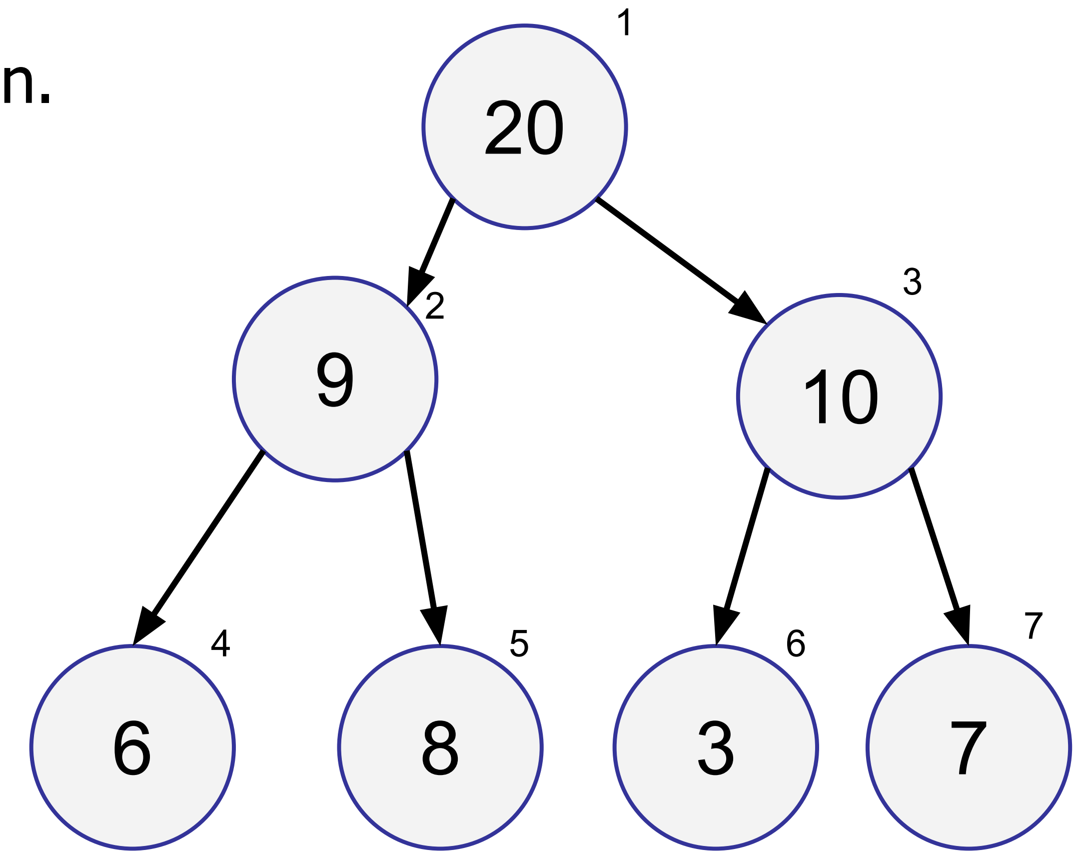


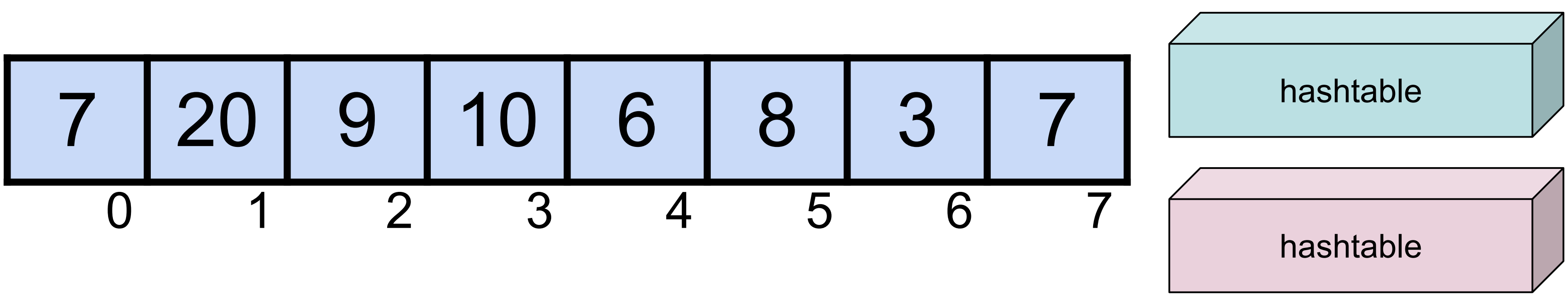
7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7



Binary Heaps:

The indices range from 1 to n.

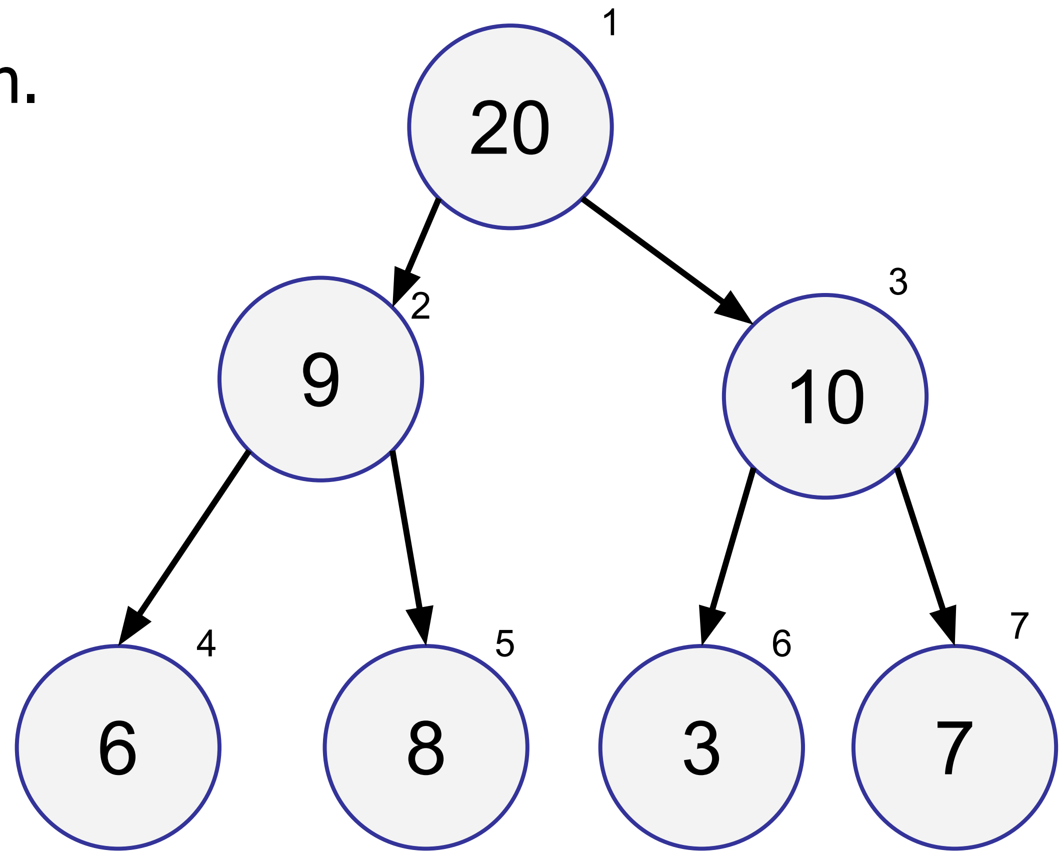


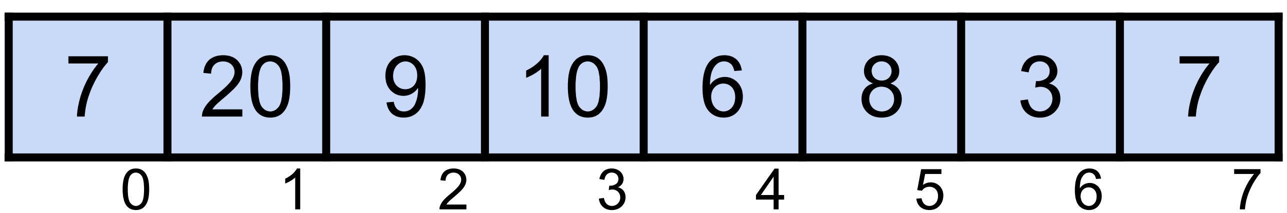


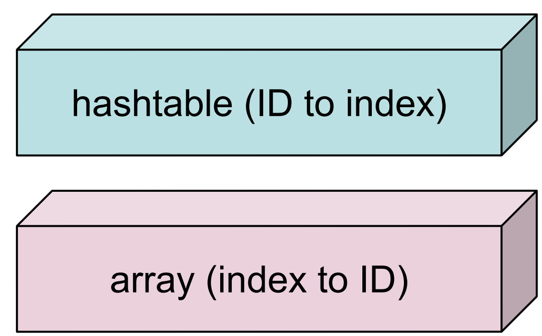
Binary Heaps:

The indices range from 1 to n.

We could just map indices to IDs using an array!





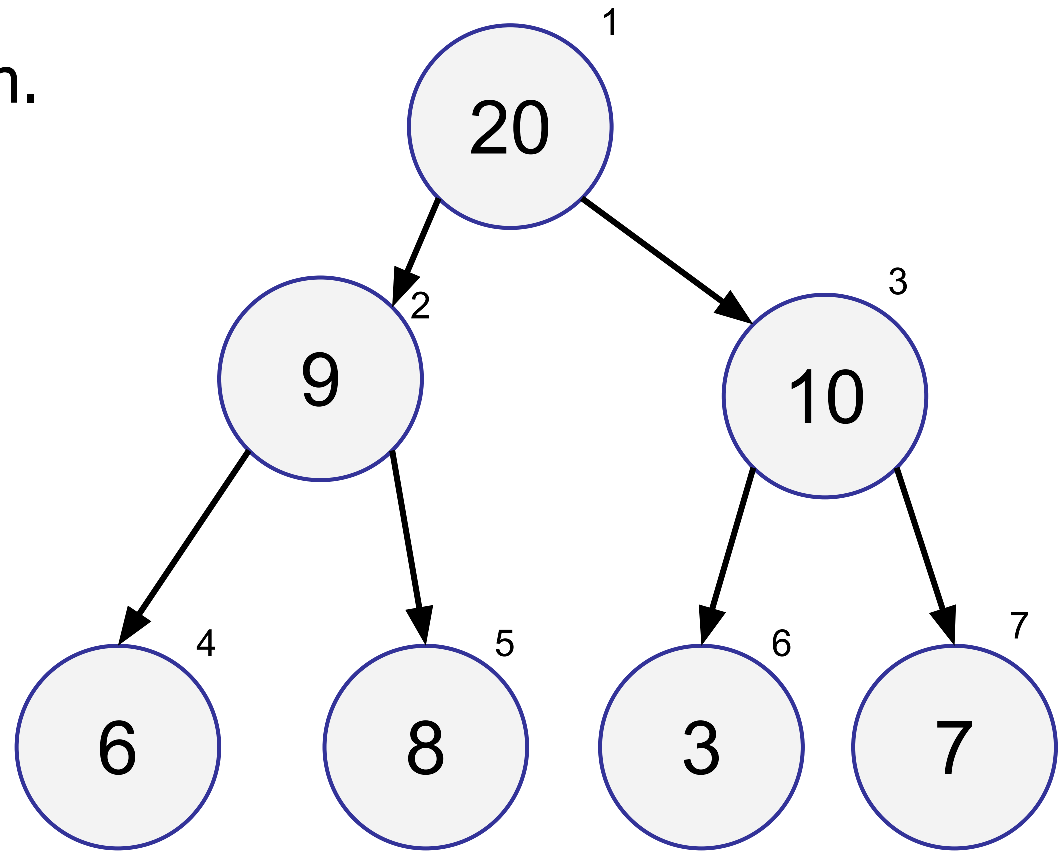


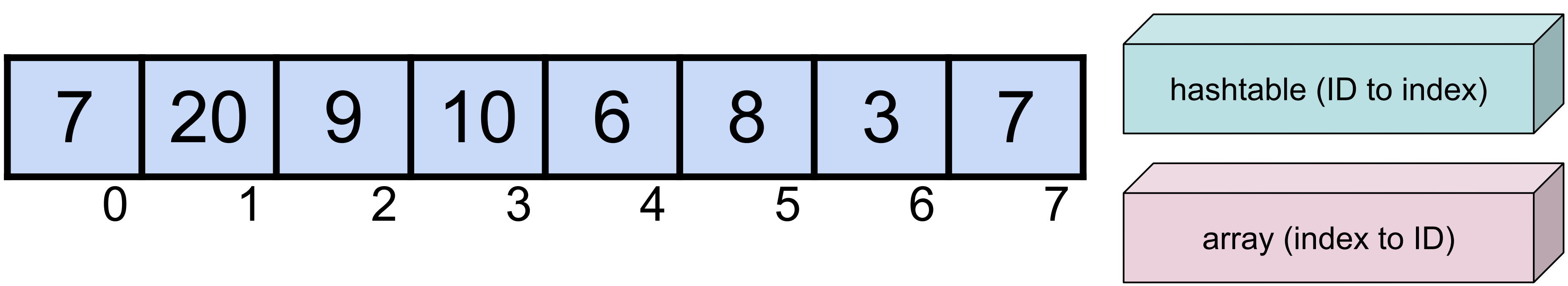
Binary Heaps:

The indices range from 1 to n.

We could just map indices to IDs using an array!

Now at least the index to ID lookup is also O(1) worst case.

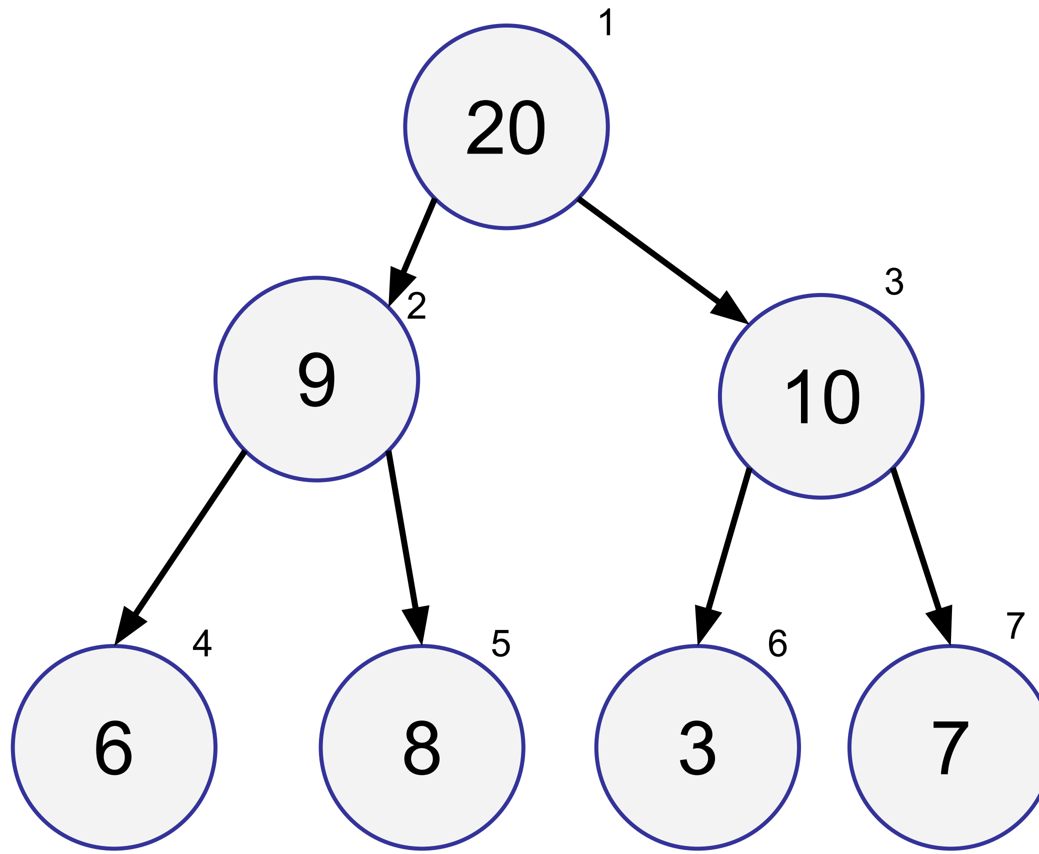




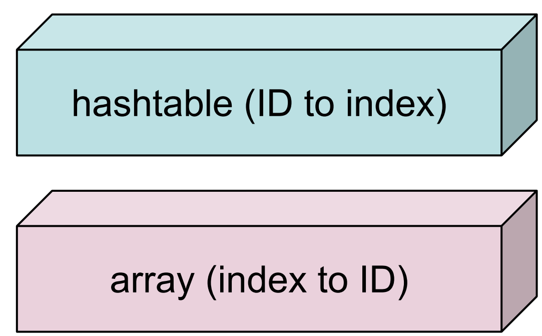
Binary Heaps:

Now when we swap nodes:

1. Update the main array.
2. Update the hashtable.
3. Update the array.



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

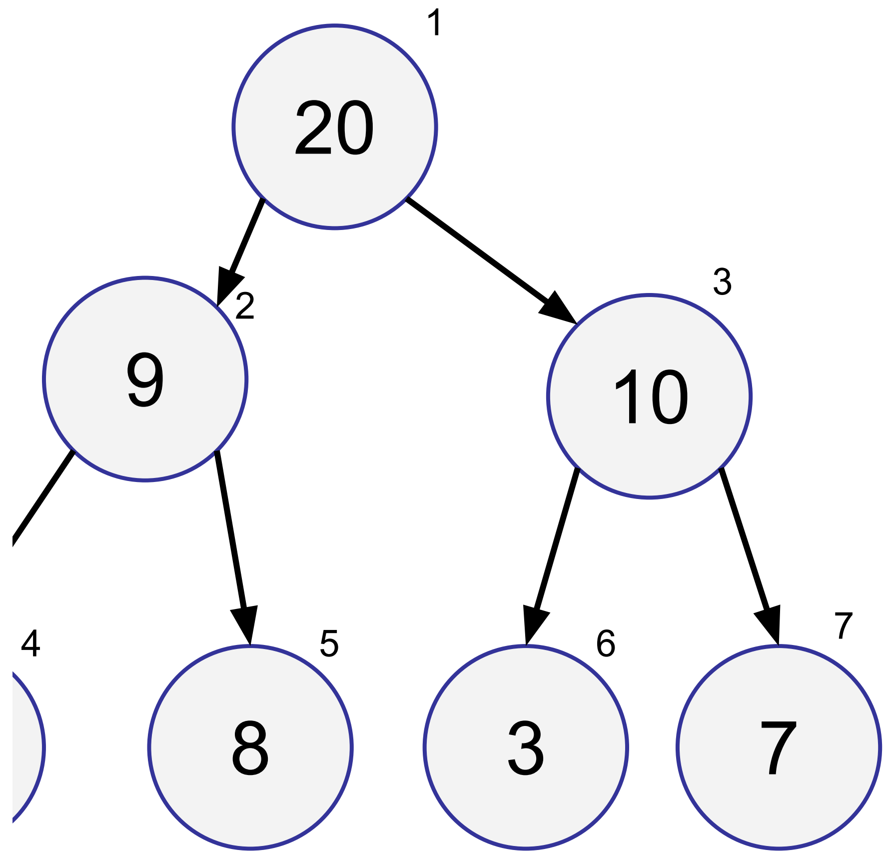


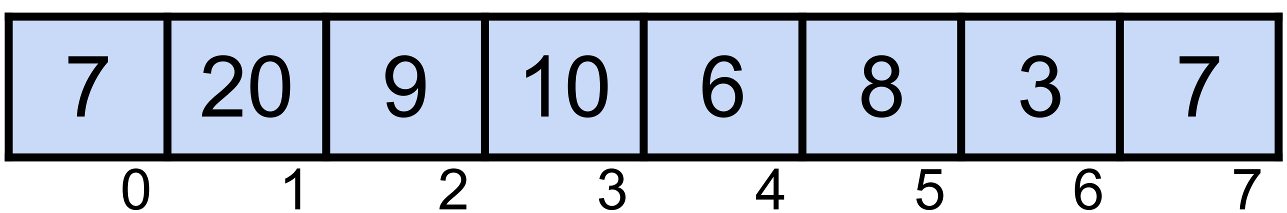
Binary Heaps:

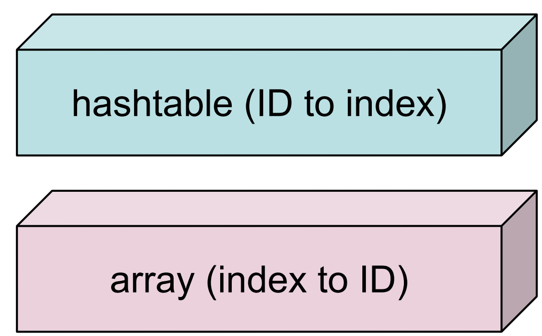
```
void    swap (int i){
  int temp = arr[i/2];
  arr[i/2] = arr[i];
  arr[i] = temp;

  K p_id = idx_to_id[i/2];
  idx_to_id[i/2] = idx_to_id[i];
  idx_to_idx[i] = p_id;

  id_to_idx[p_id] = i;
  id_to_idx[idx_to_id[i/2]] = i/2;
}
```





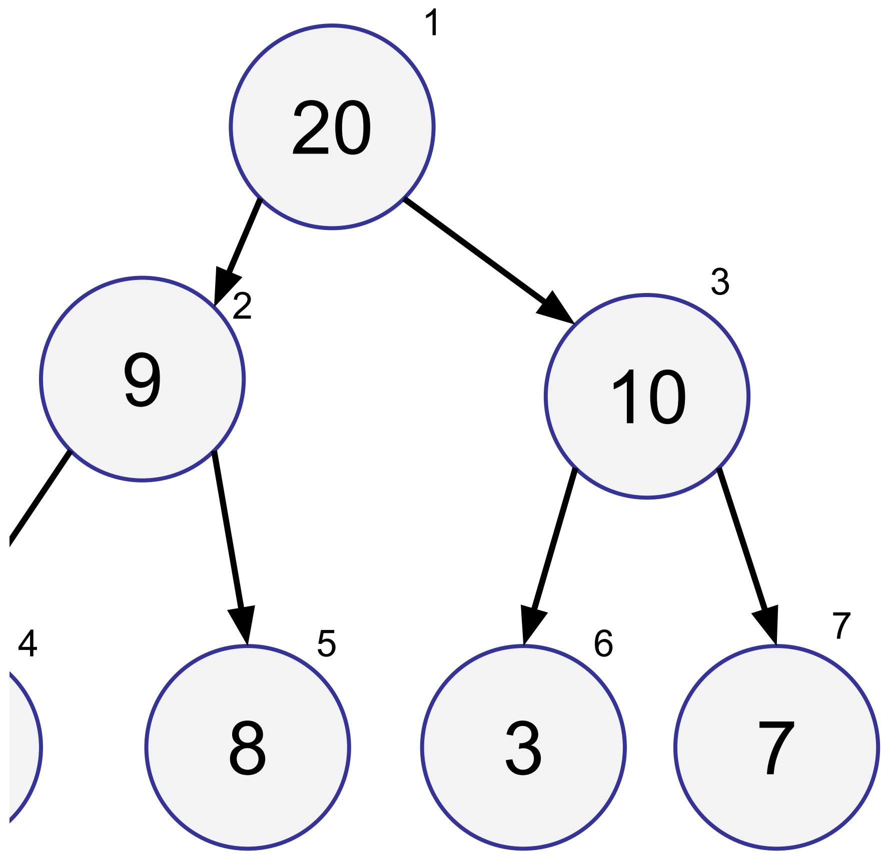


Binary Heaps:

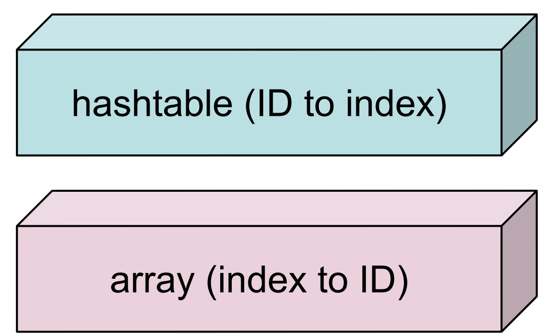
```
void   swap (int i){
  int temp = arr[i/2];
  arr[i/2] = arr[i];
  arr[i] = temp;

  K p_id = idx_to_id[i/2];
  idx_to_id[i/2] = idx_to_id[i];
  idx_to_idx[i] = p_id;

  id_to_idx[p_id] = i;
  id_to_idx[idx_to_id[i/2]] = i/2;
}
```



7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

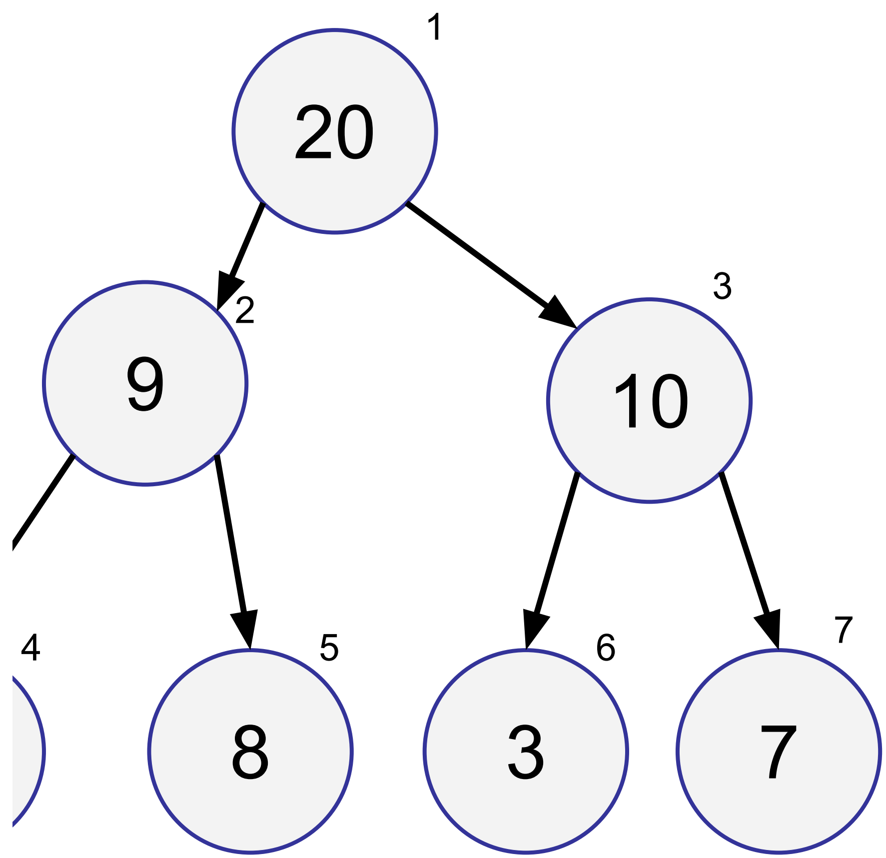


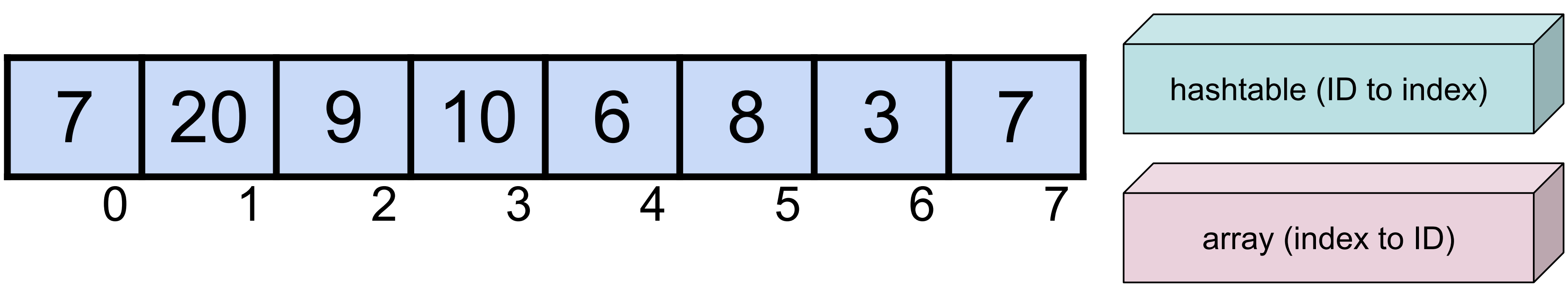
Binary Heaps:

```
void    swap (int i){
  int temp = arr[i/2];
  arr[i/2] = arr[i];
  arr[i] = temp;

  K p_id = idx_to_id[i/2];
  idx_to_id[i/2] = idx_to_id[i];
  idx_to_idx[i] = p_id;

  id_to_idx[p_id] = i;
  id_to_idx[idx_to_id[i/2]] = i/2;
}
```



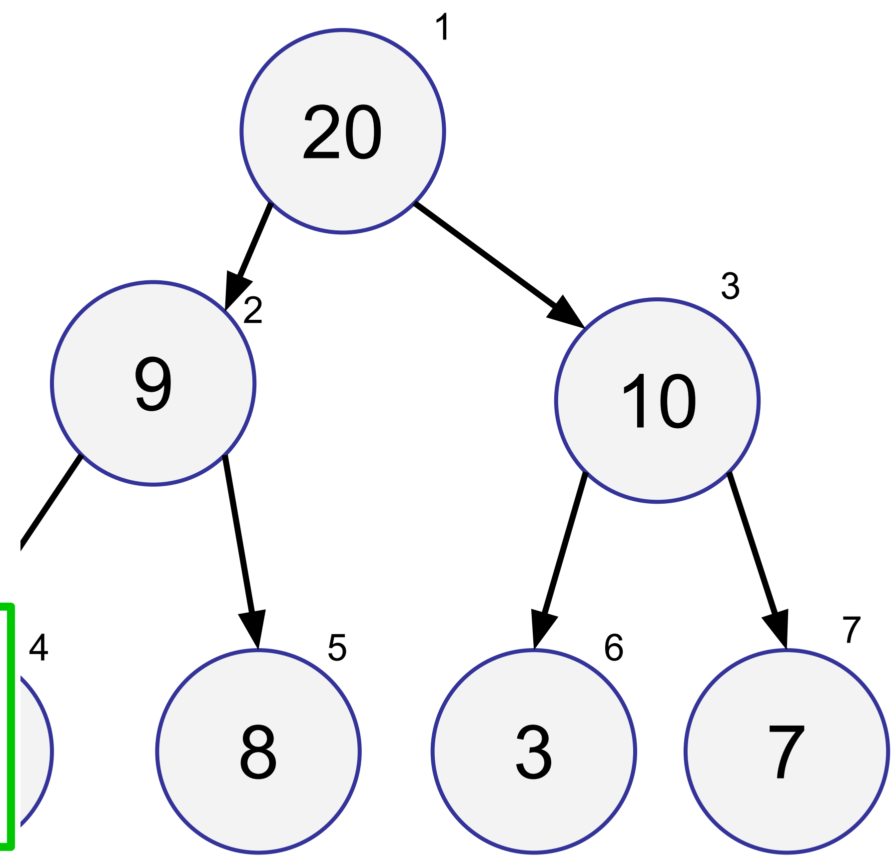


Binary Heaps:

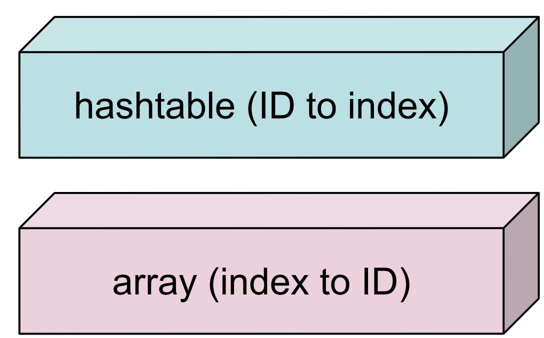
```
void    swap (int i){
  int temp = arr[i/2];
  arr[i/2] = arr[i];
  arr[i] = temp;

  K p_id = idx_to_id[i/2];
  idx_to_id[i/2] = idx_to_id[i];
  idx_to_idx[i] = p_id;

  id_to_idx[p_id] = i;
  id_to_idx[idx_to_id[i/2]] = i/2;
}
```



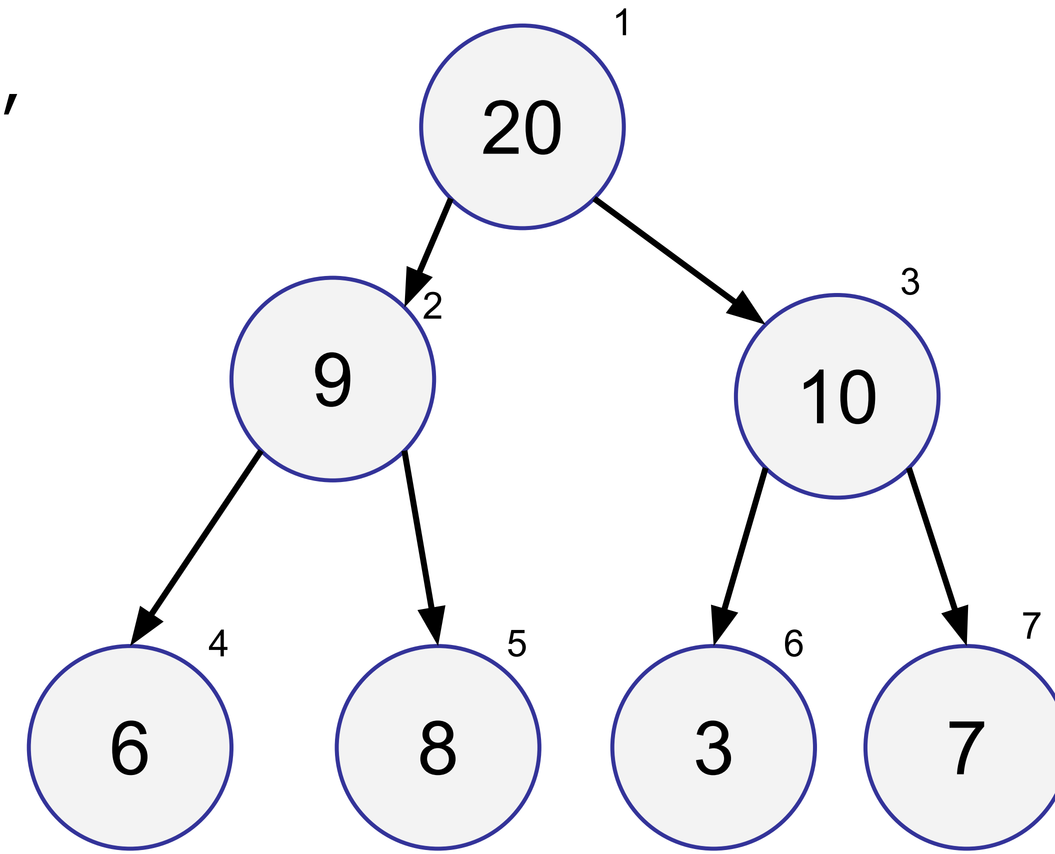
7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7



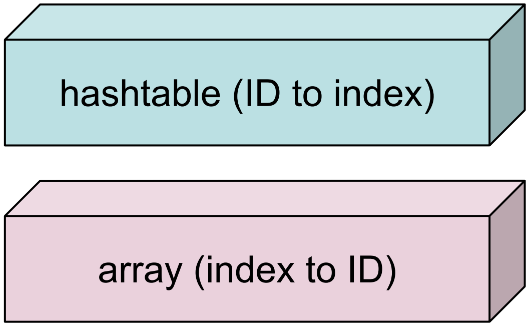
Binary Heaps:

Back to the problem at hand,

increaseKey.

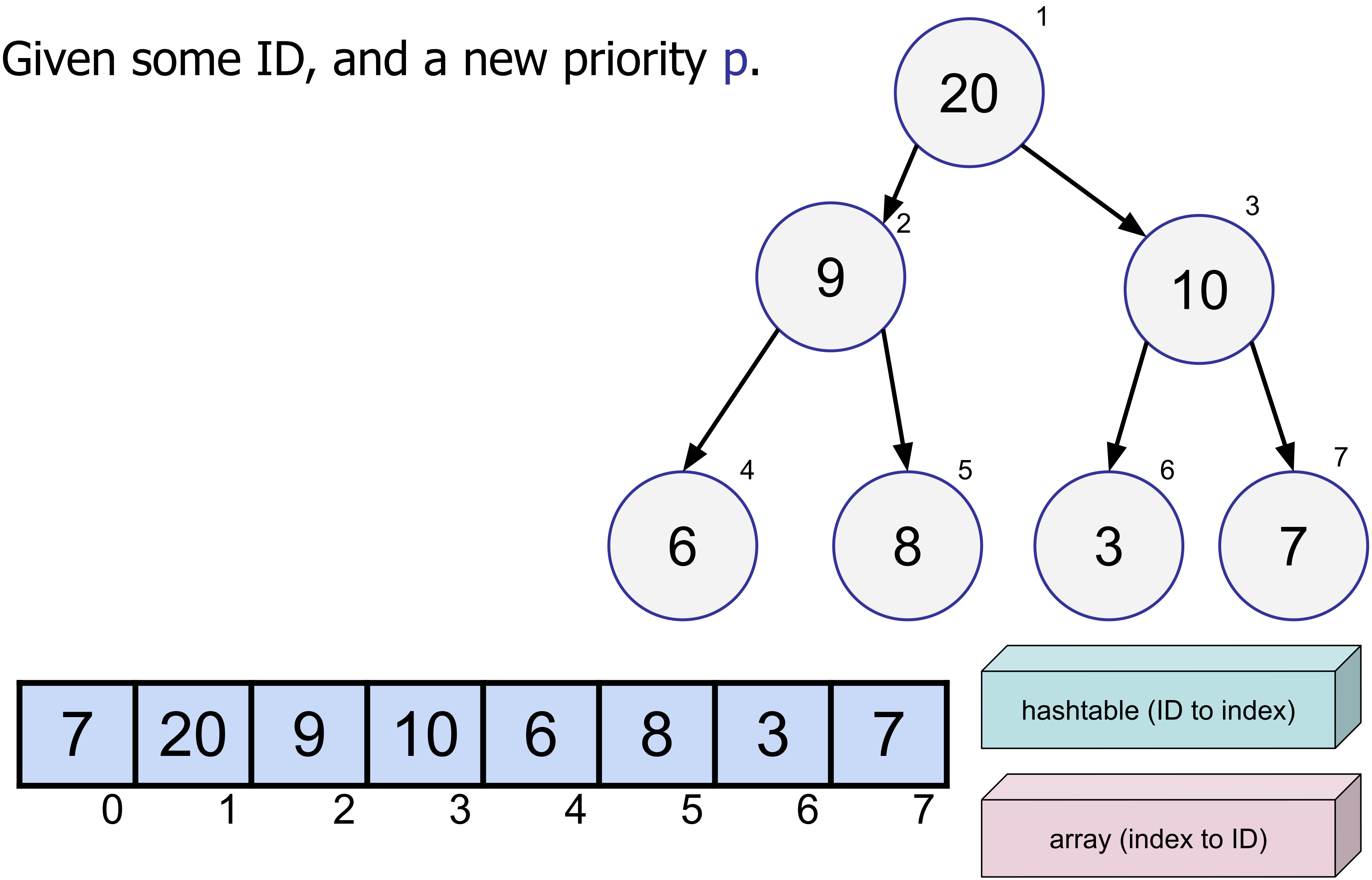


7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7



Binary Heaps:

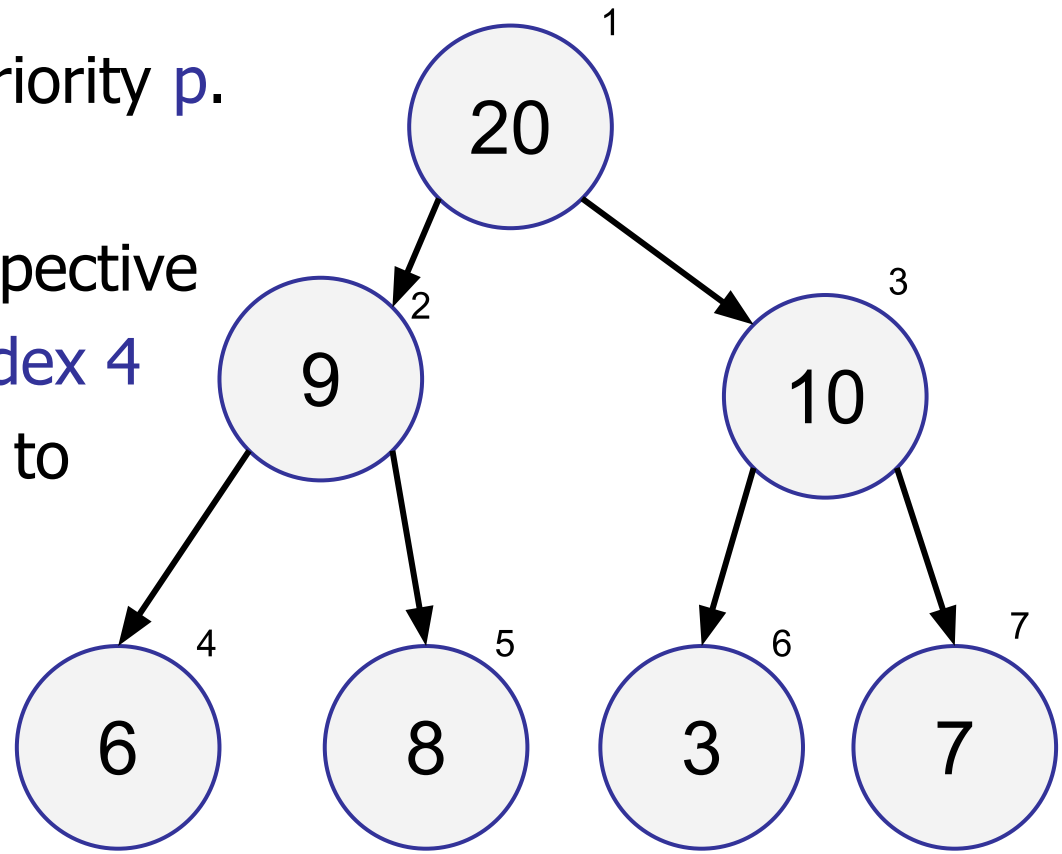
Given some ID, and a new priority p.



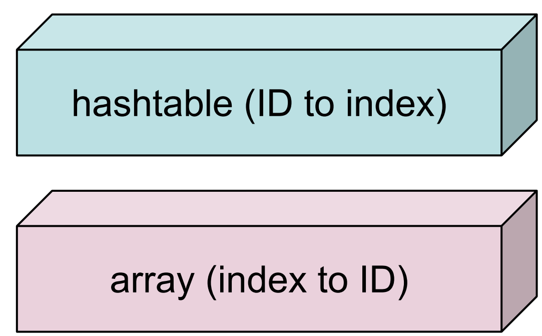
Binary Heaps:

Given some ID, and a new priority p.

Use hashtable to find the respective node. E.g. the node at index 4 needs to increase priority to 11.



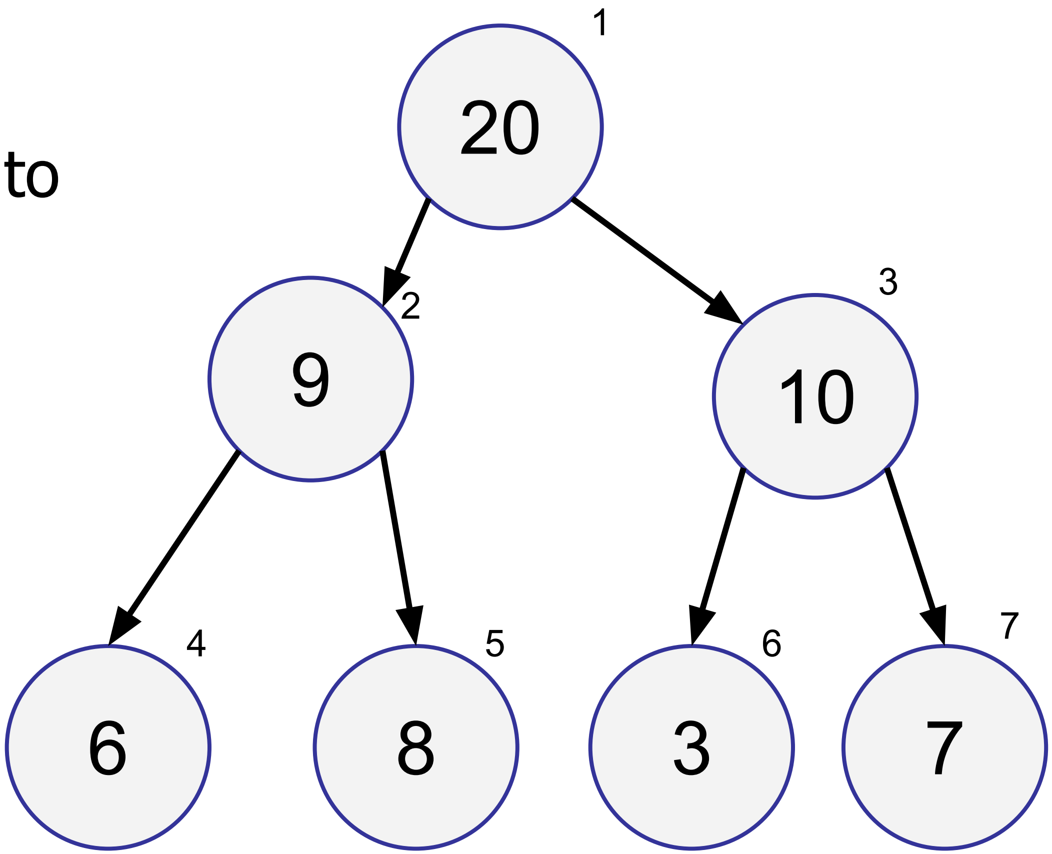
7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7



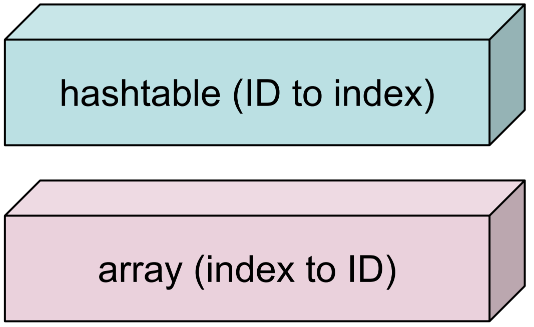
Binary Heaps:

E.g. the node at index 4 needs to increase priority to 11.

Set the new priority, and bubble up



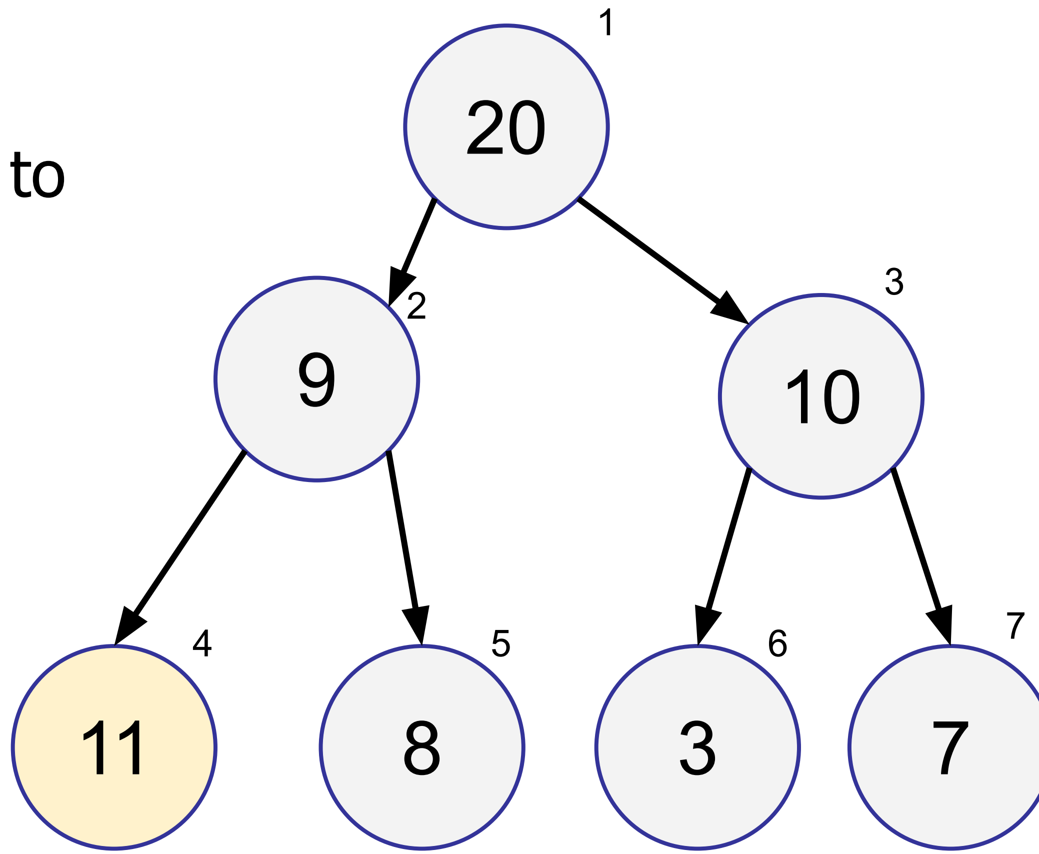
7	20	9	10	6	8	3	7
0	1	2	3	4	5	6	7

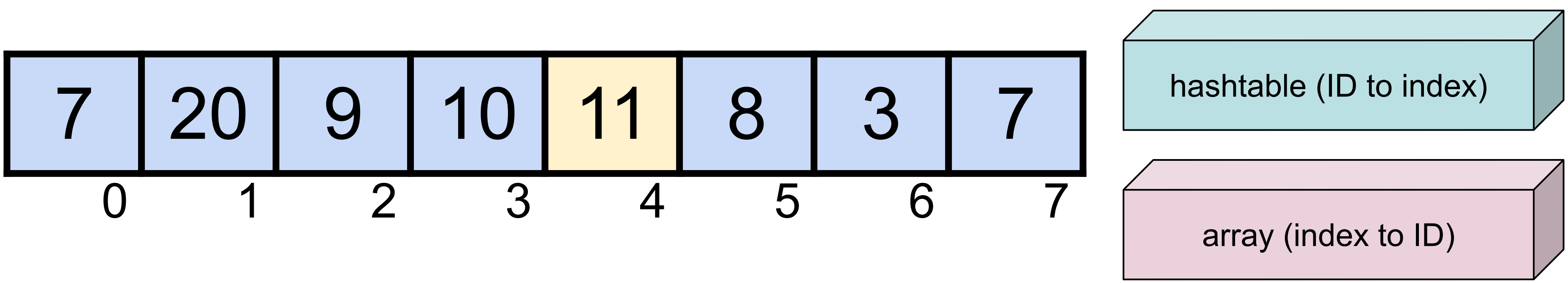


Binary Heaps:

E.g. the node at index 4 needs to increase priority to 11.

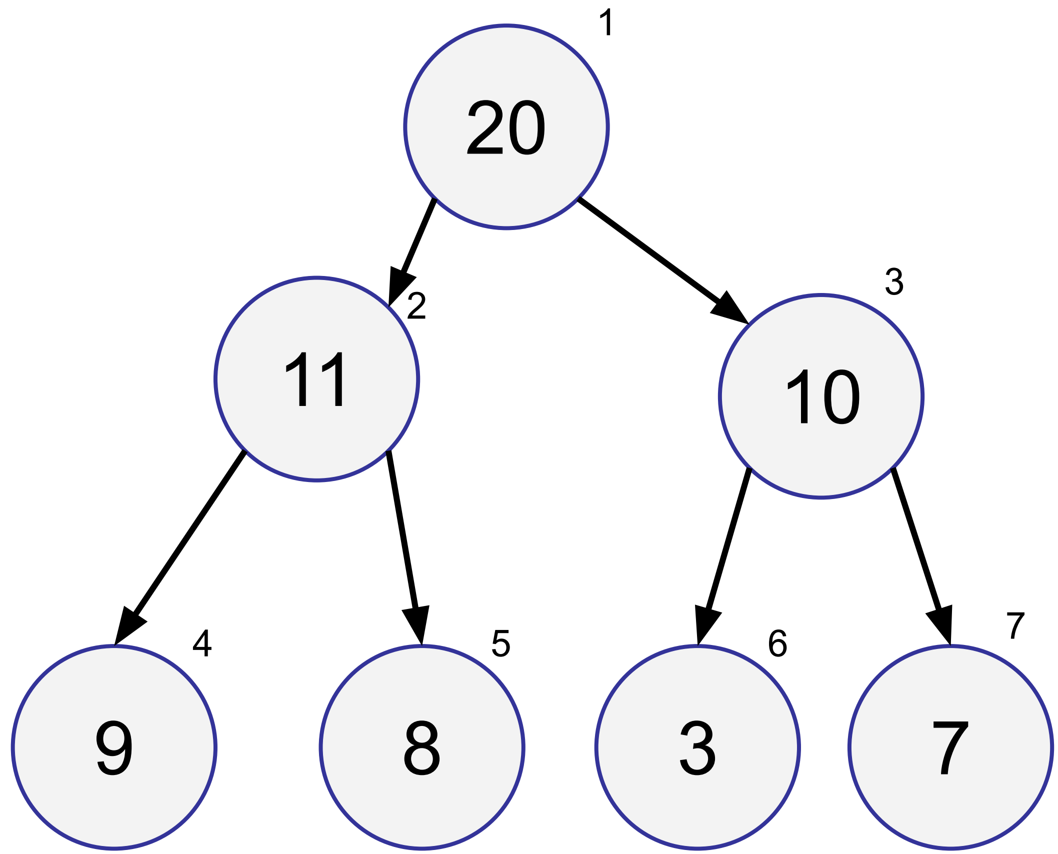
Set the new priority, and bubble up

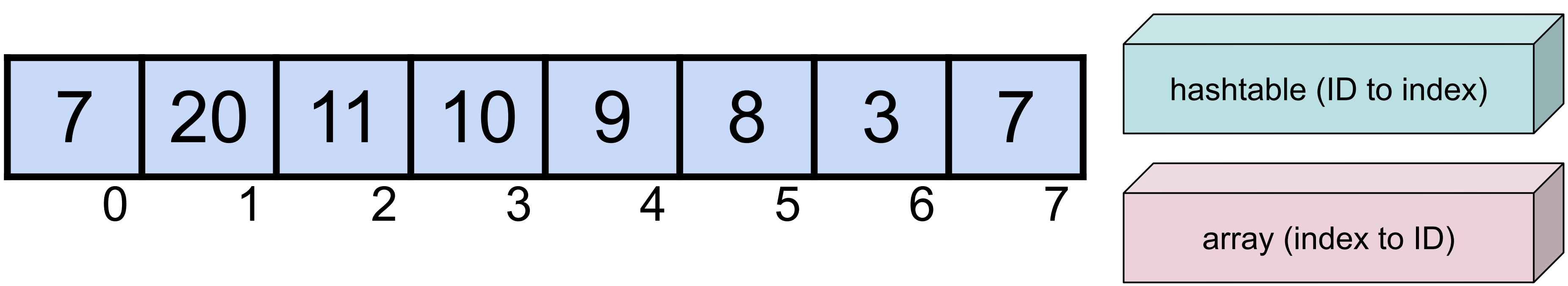




Binary Heaps:

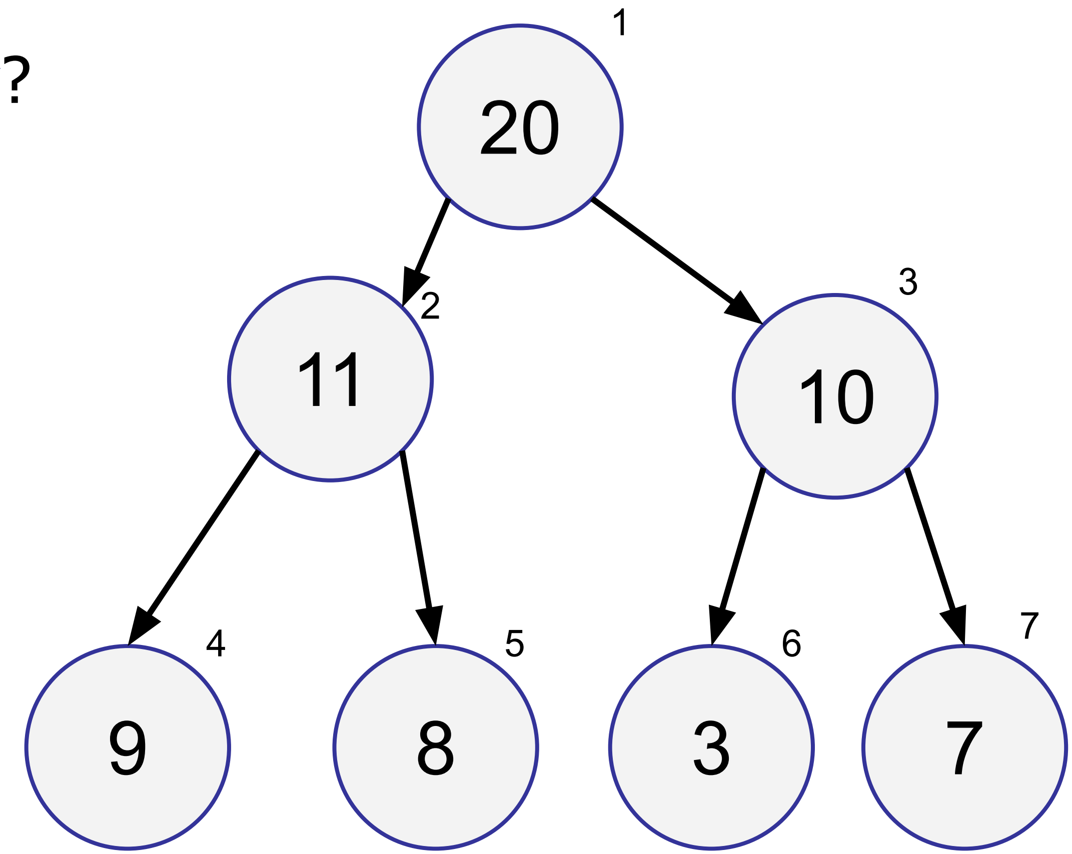
For the same reason, this is
O(log n)



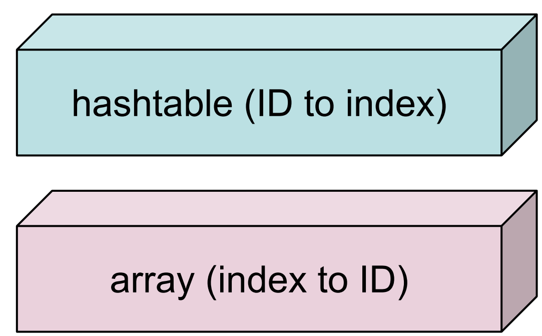


Binary Heaps:

But what about decreaseKey?



7	20	11	10	9	8	3	7
0	1	2	3	4	5	6	7

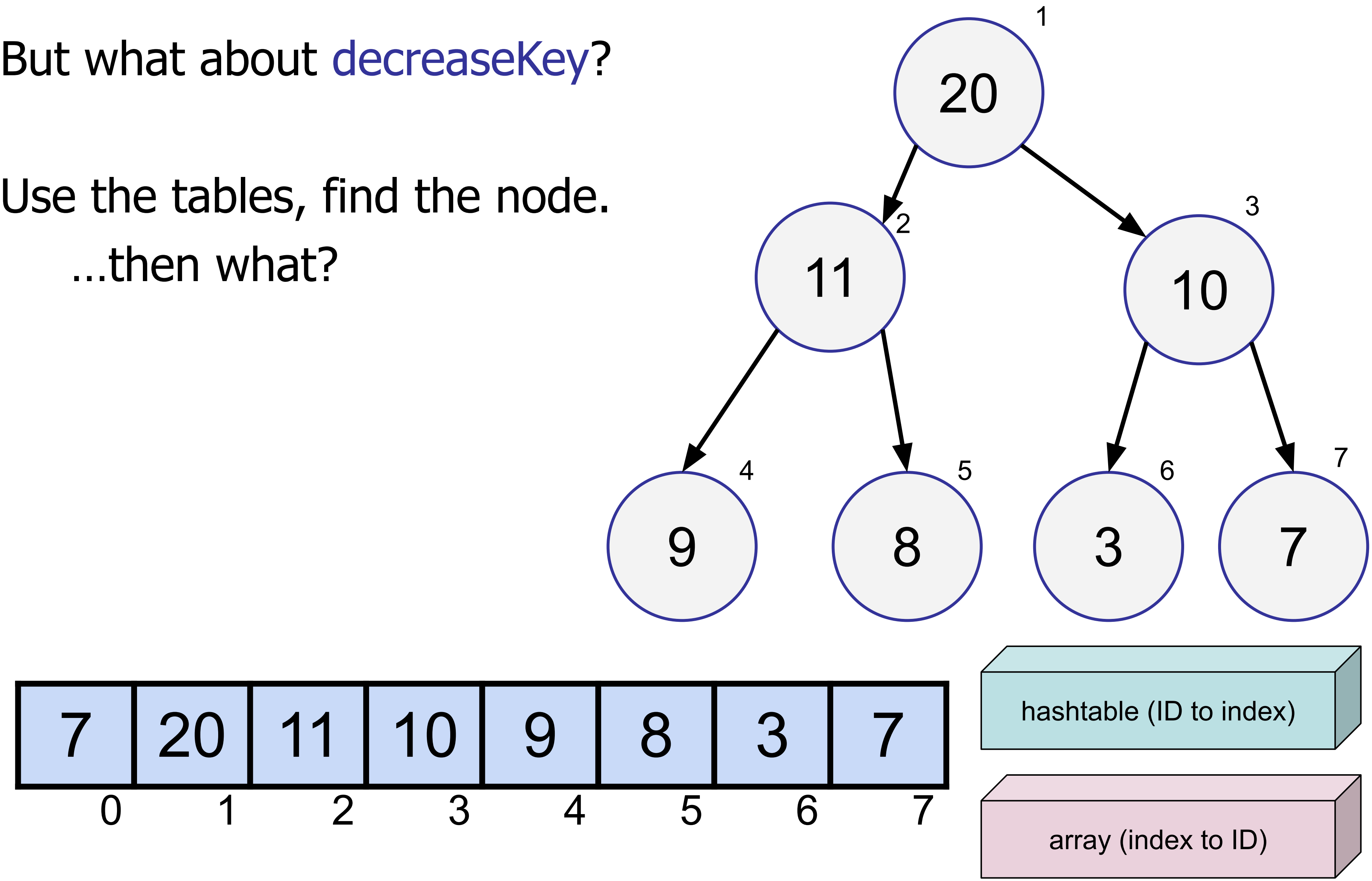


Binary Heaps:

But what about decreaseKey?

Use the tables, find the node.

...then what?



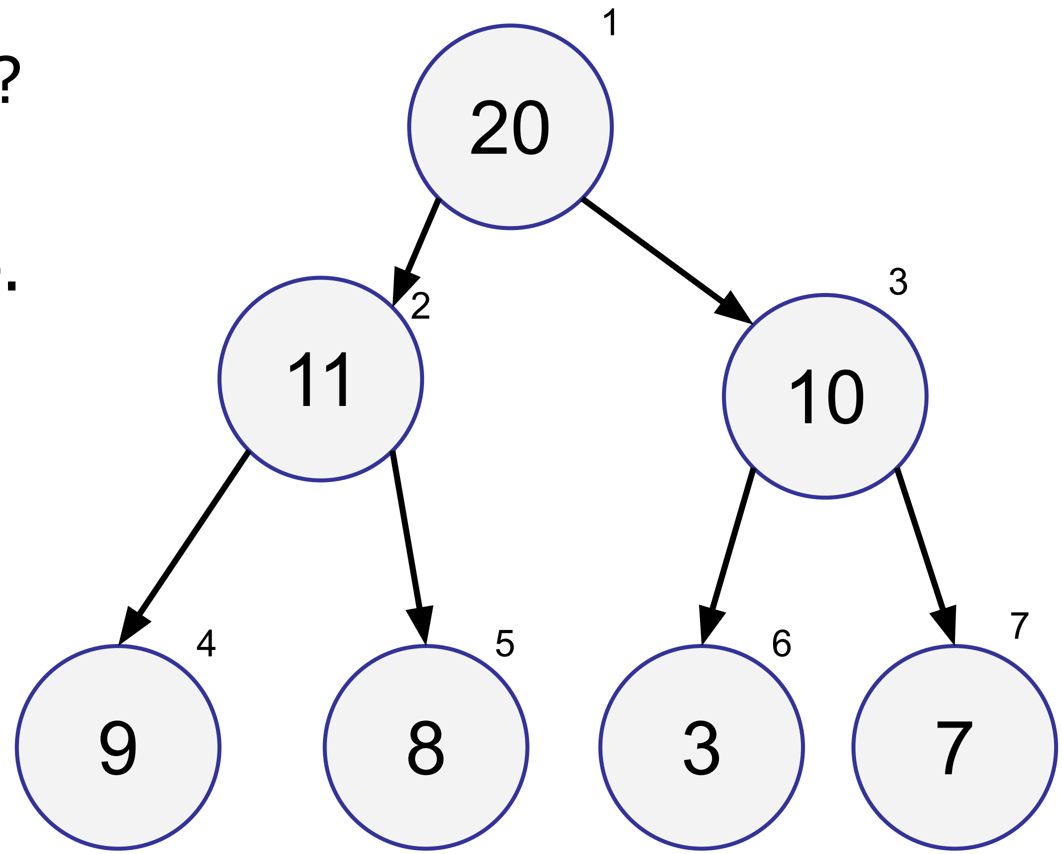
Binary Heaps:

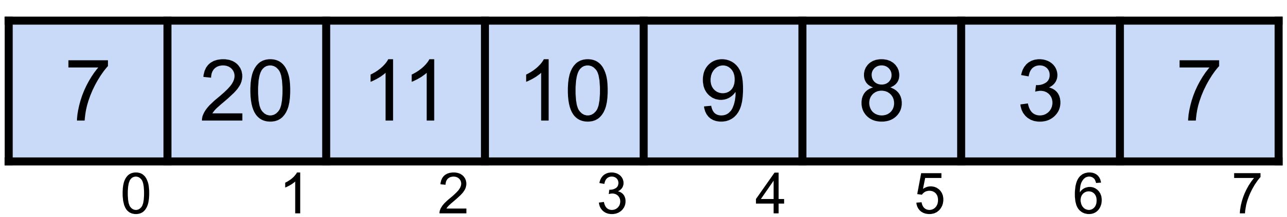
But what about decreaseKey?

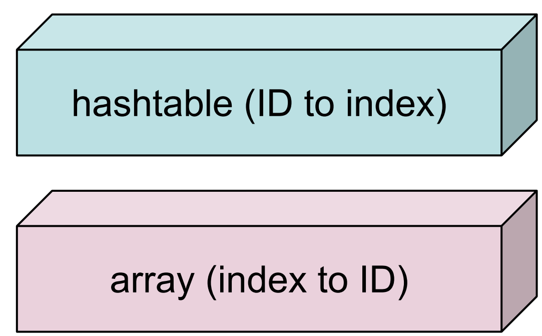
Use the tables, find the node.

...then what?

bubble down instead!



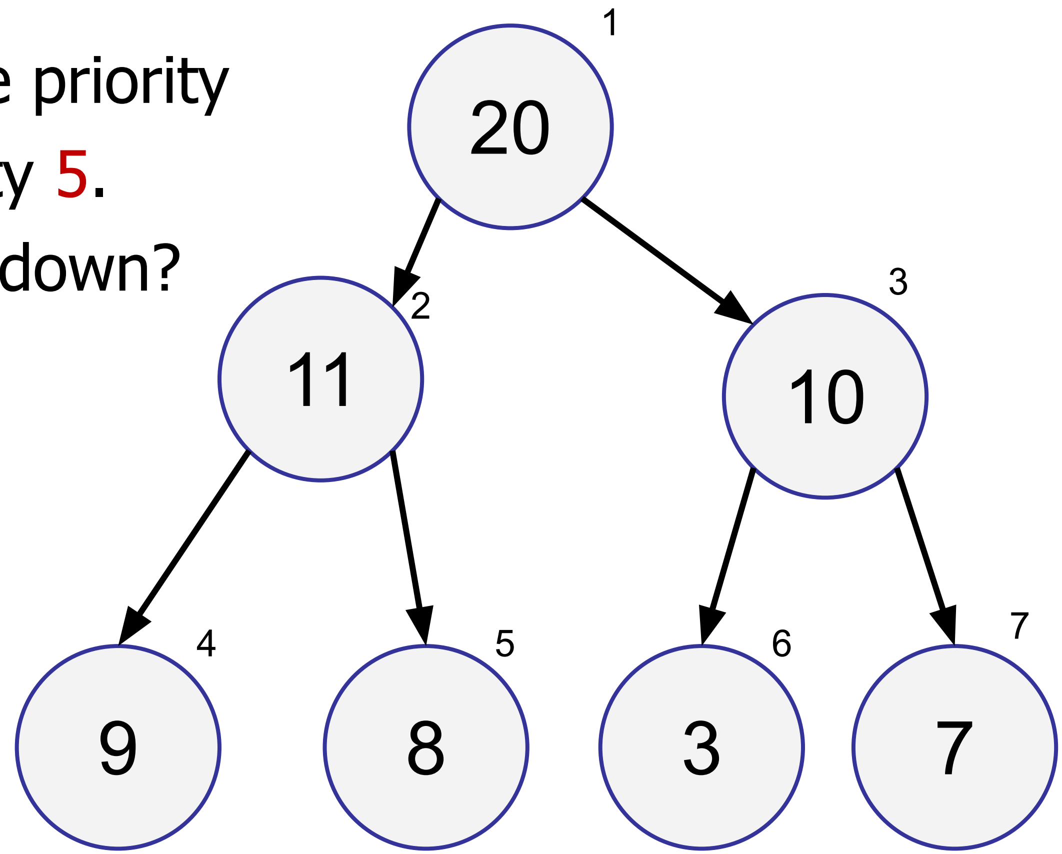


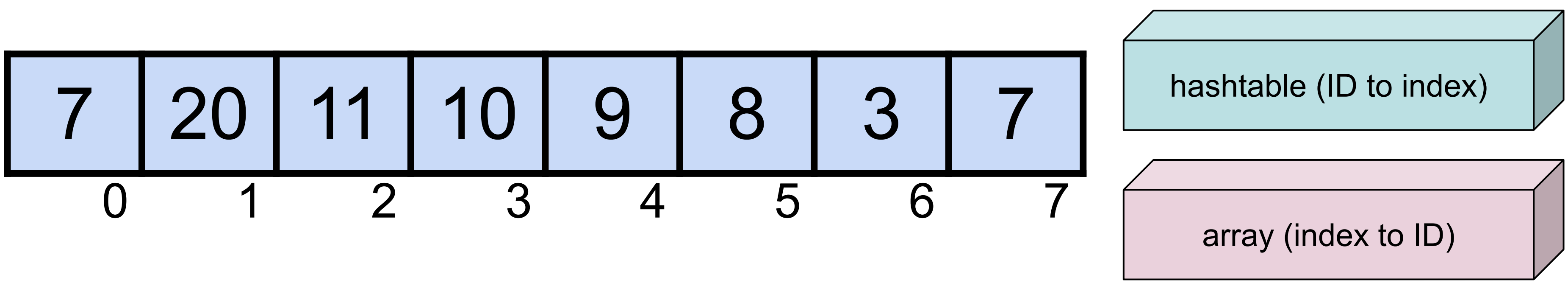


Binary Heaps:

E.g. we need to decrease the priority of the root node to priority 5.

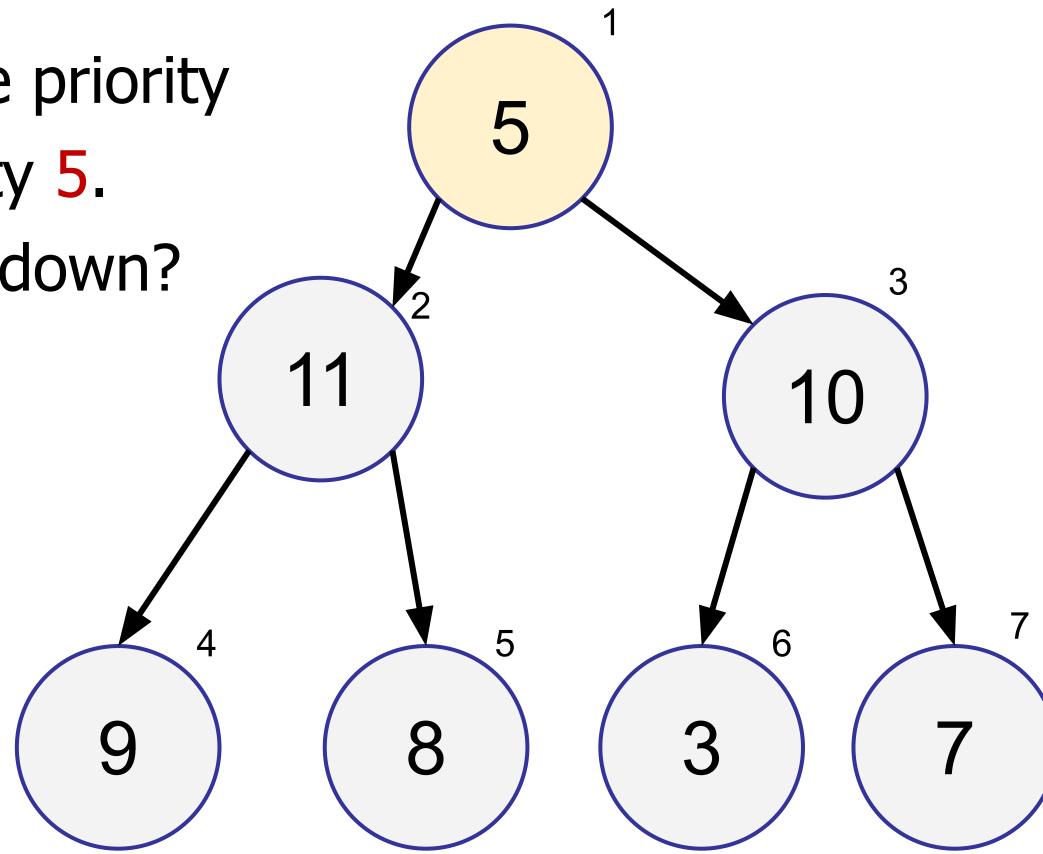
How should we bubble it down?

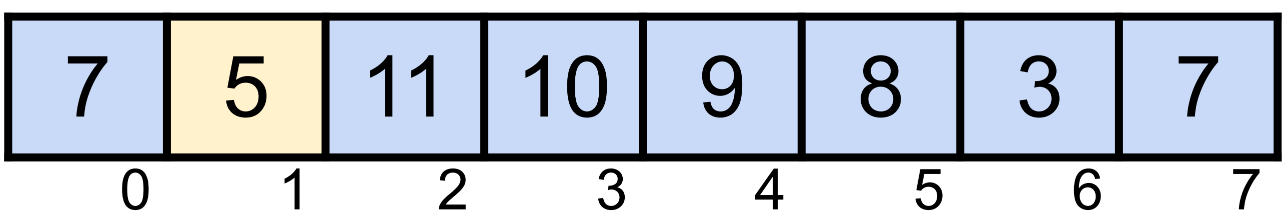


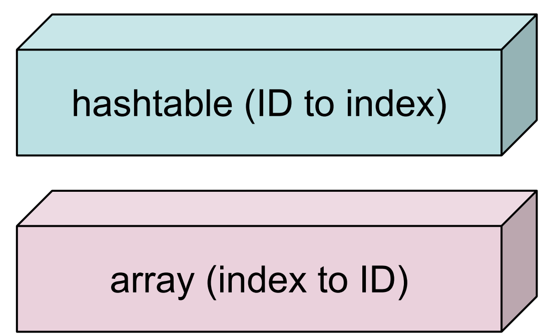


Binary Heaps:

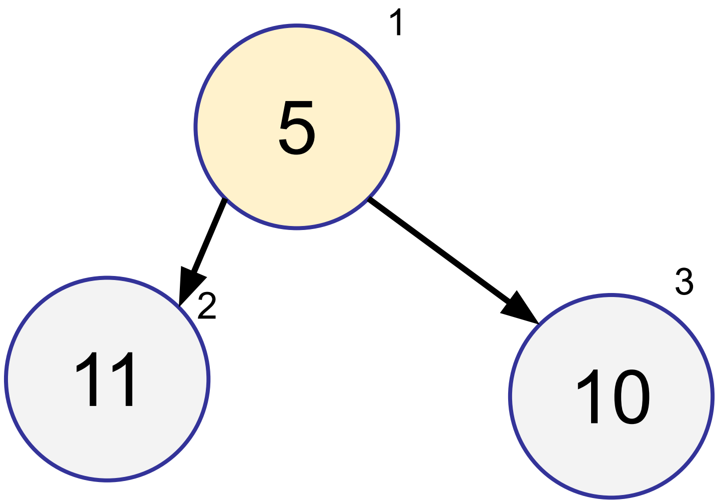
E.g. we need to decrease the priority of the root node to priority 5. How should we bubble it down?







To maintain the heap priority, we should swap it with:



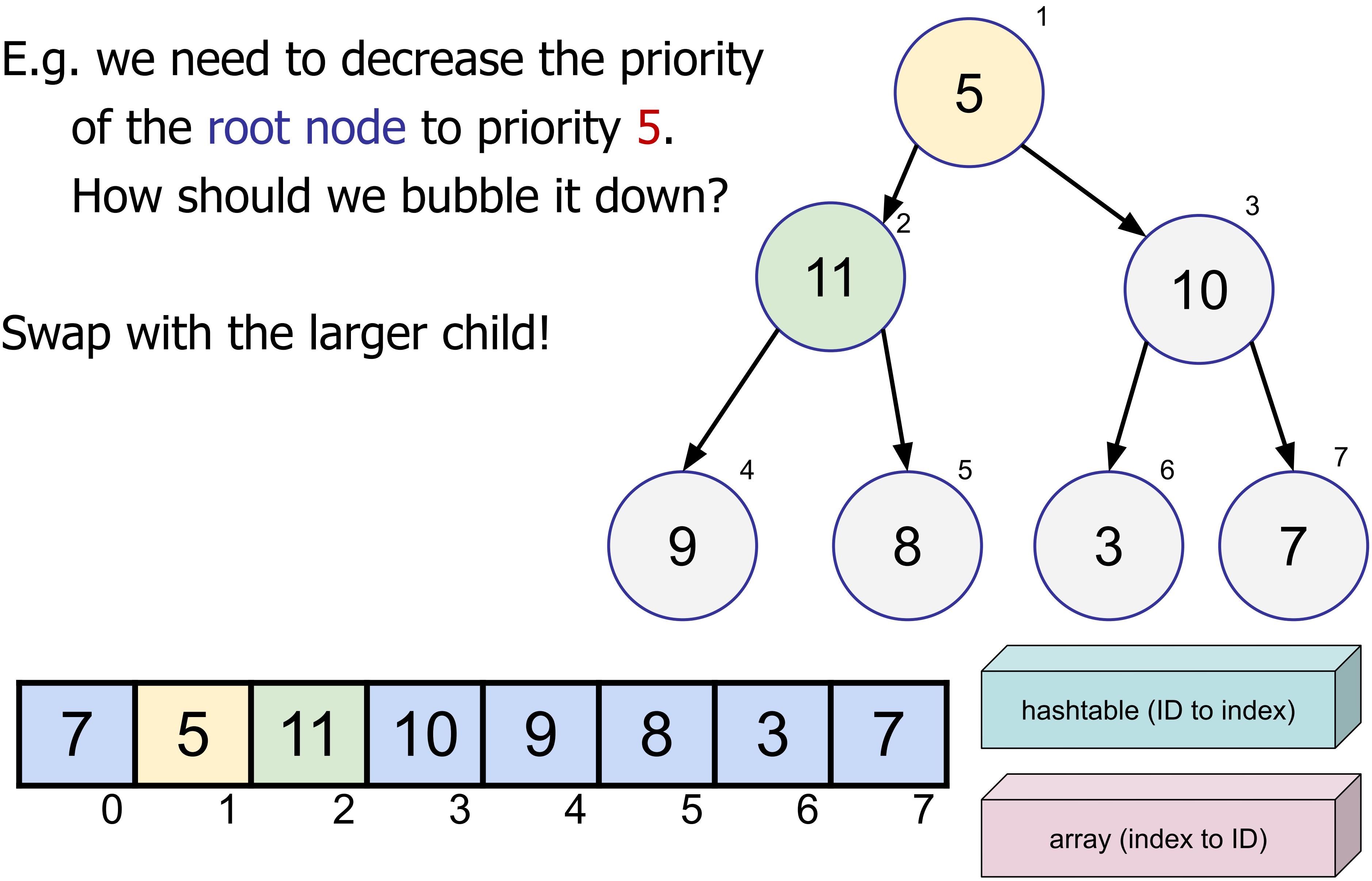
1. The left child
2. The right child
3. Trick question, no swapping.

Binary Heaps:

E.g. we need to decrease the priority of the root node to priority 5.

How should we bubble it down?

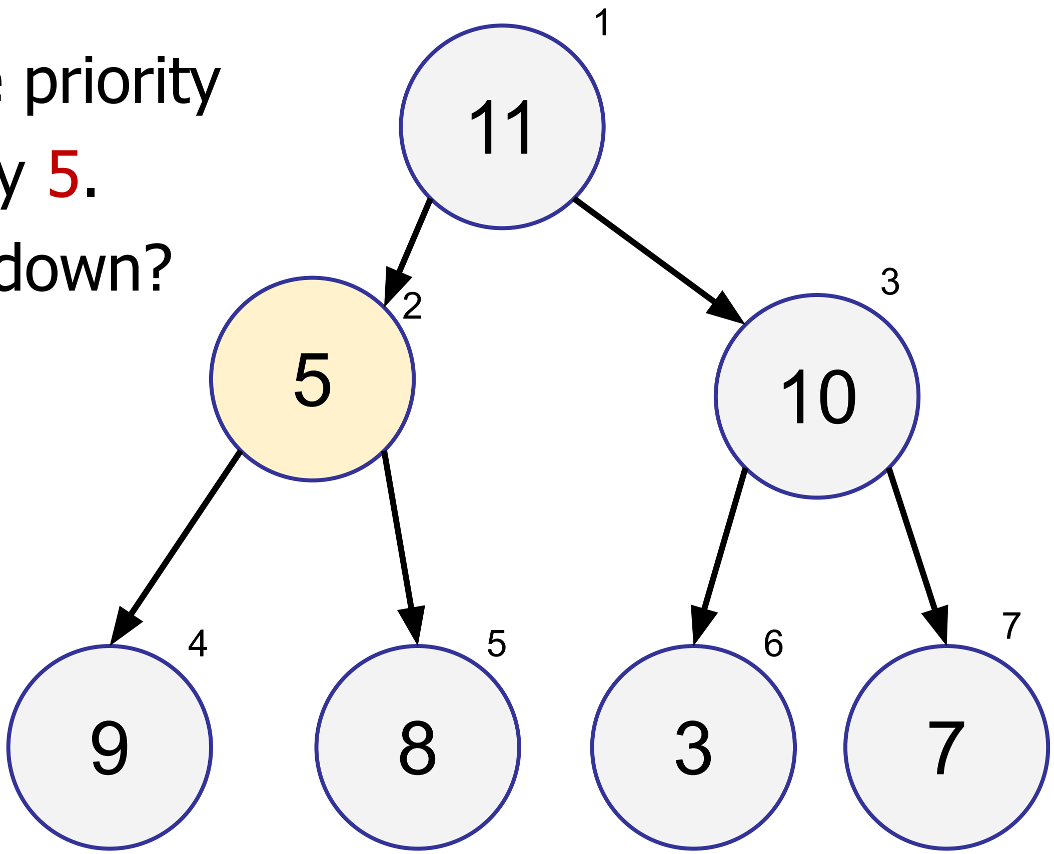
Swap with the larger child!

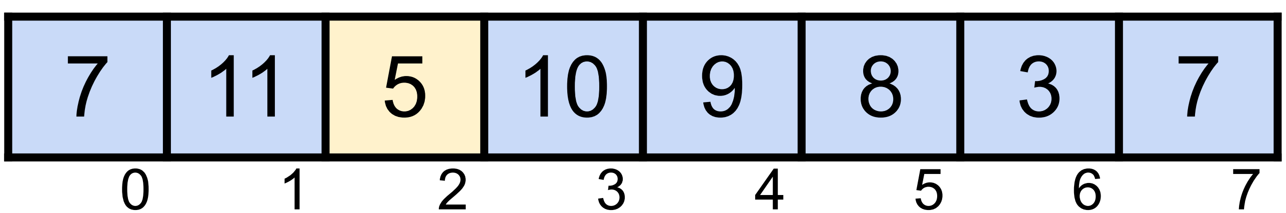


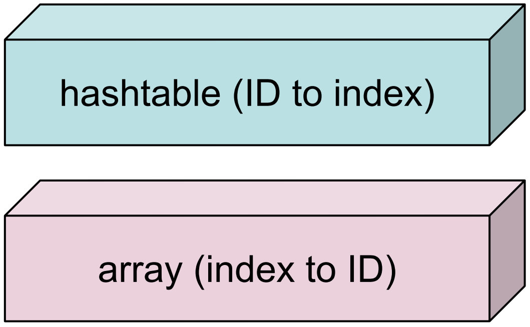
Binary Heaps:

E.g. we need to decrease the priority of the root node to priority 5.
How should we bubble it down?

Swap with the larger child!



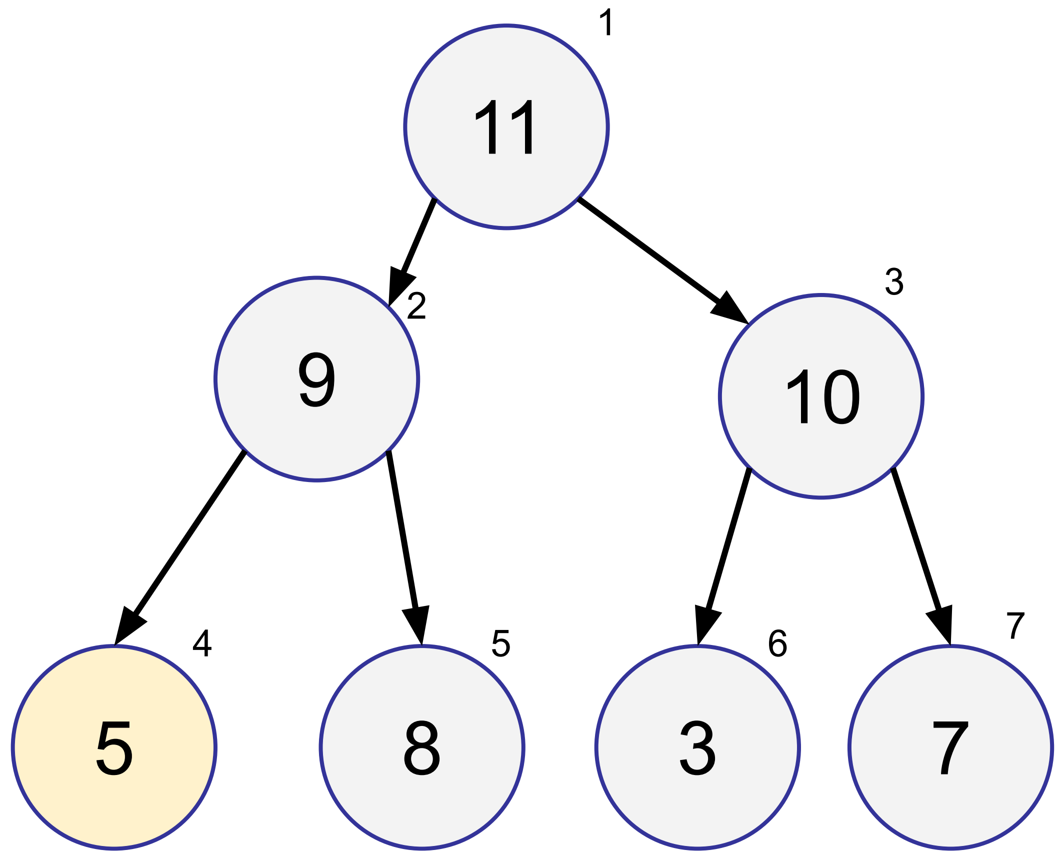


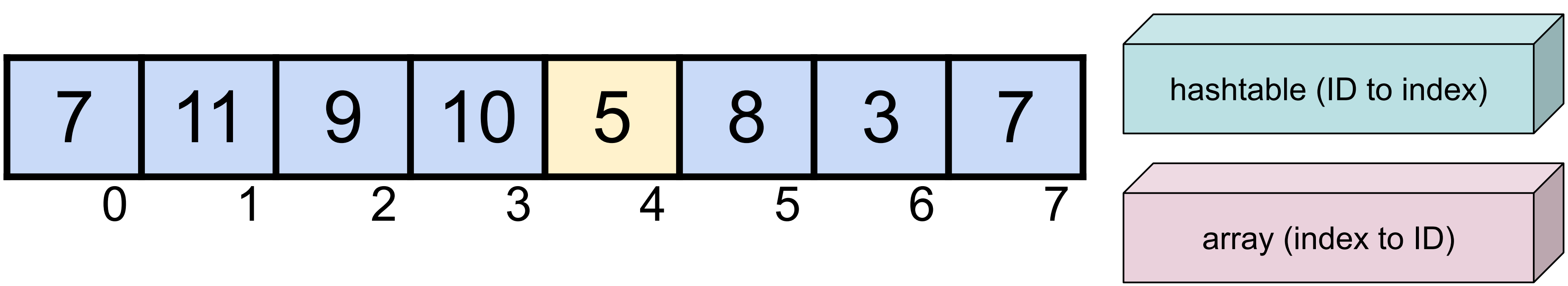


Binary Heaps:

Repeat until:

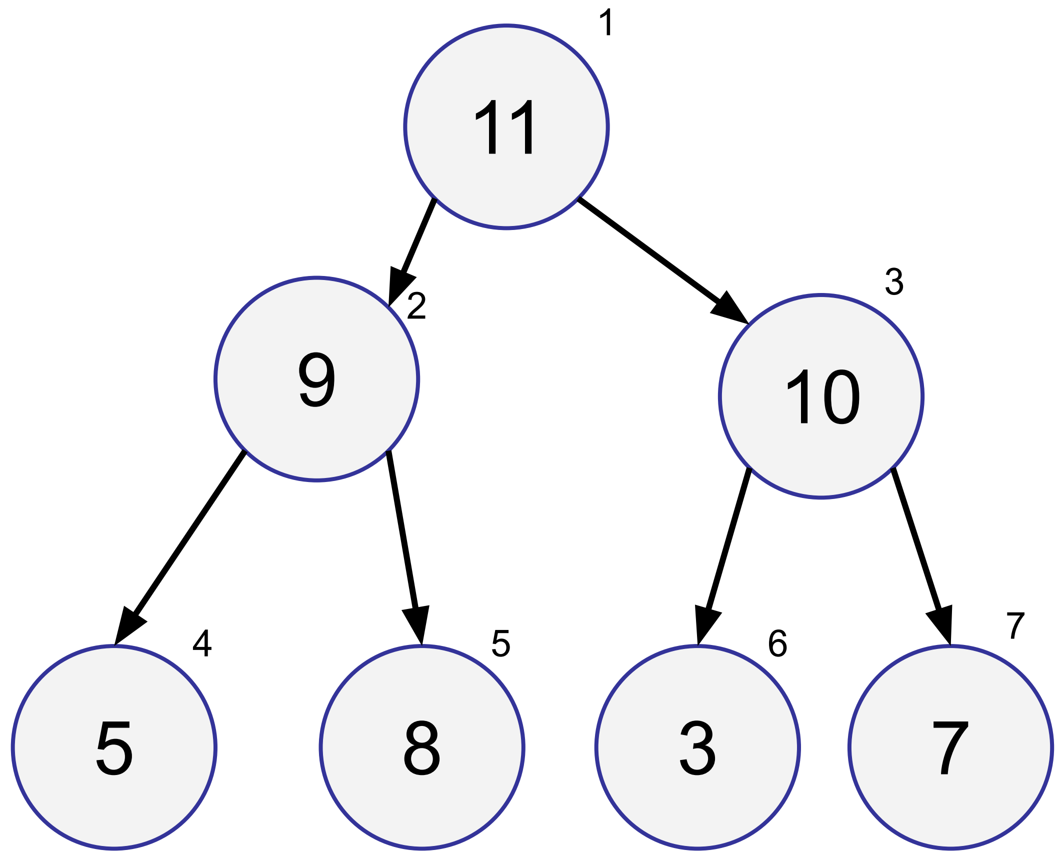
1. At leaf; or
2. Larger than both children



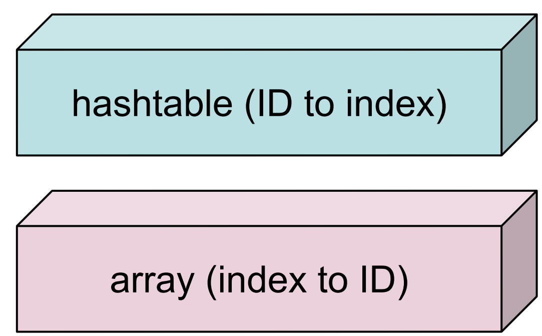


Binary Heaps:

If we did this, we are always maintaining both invariants!

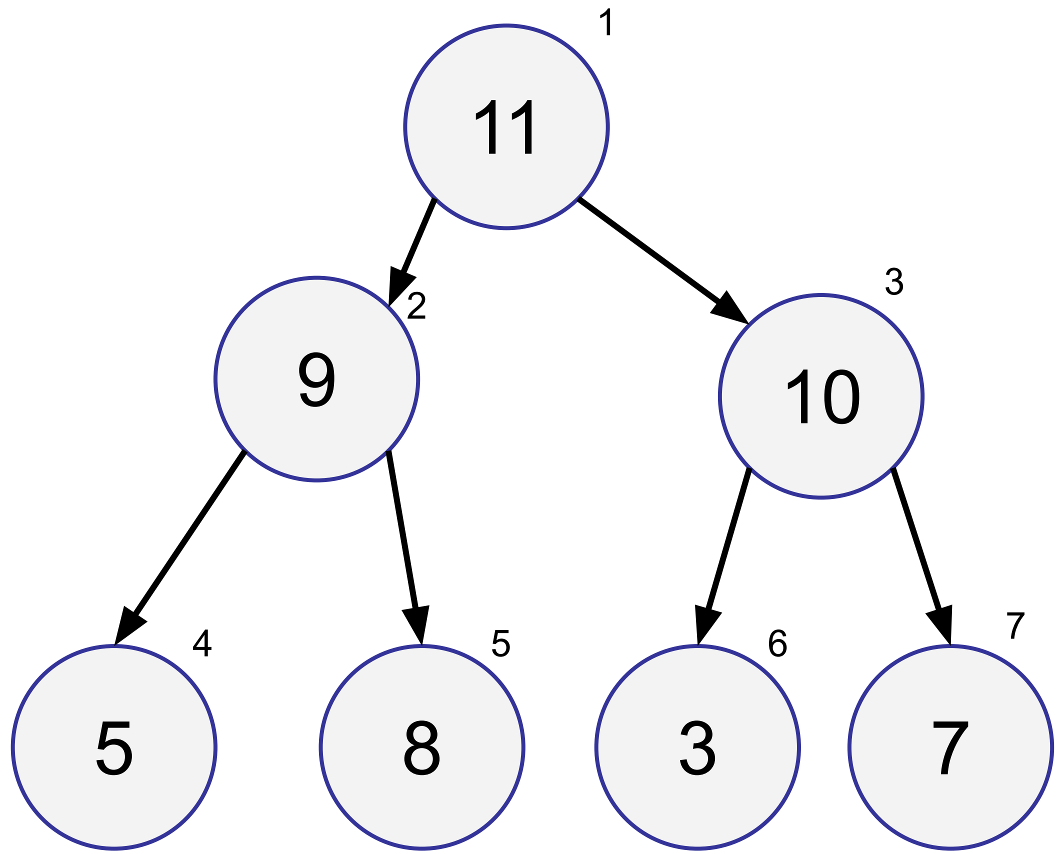


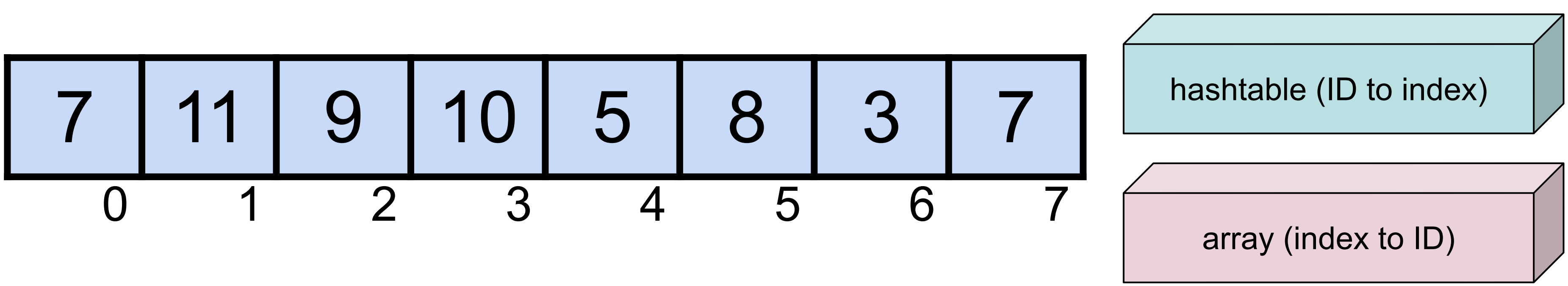
7	11	9	10	5	8	3	7
0	1	2	3	4	5	6	7



Binary Heaps:

What about extractMax?

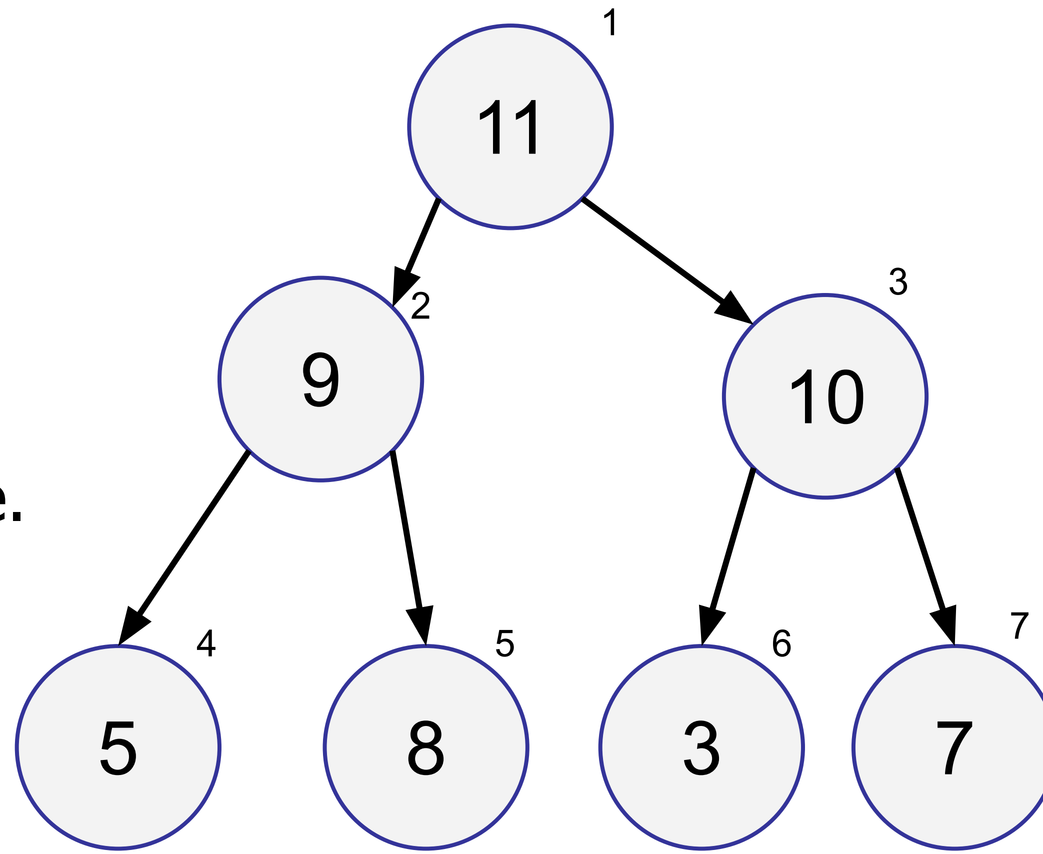




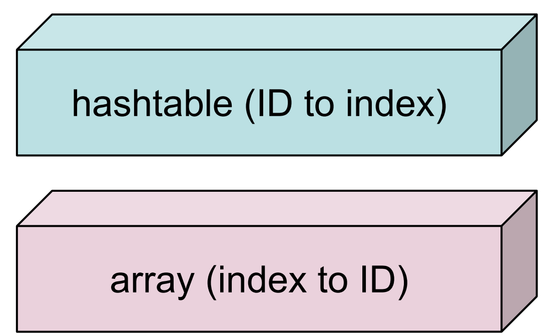
Binary Heaps:

What about extractMax?

1. remove root node.
2. pull up max child to replace missing node.
3. Recurse on either subtree.



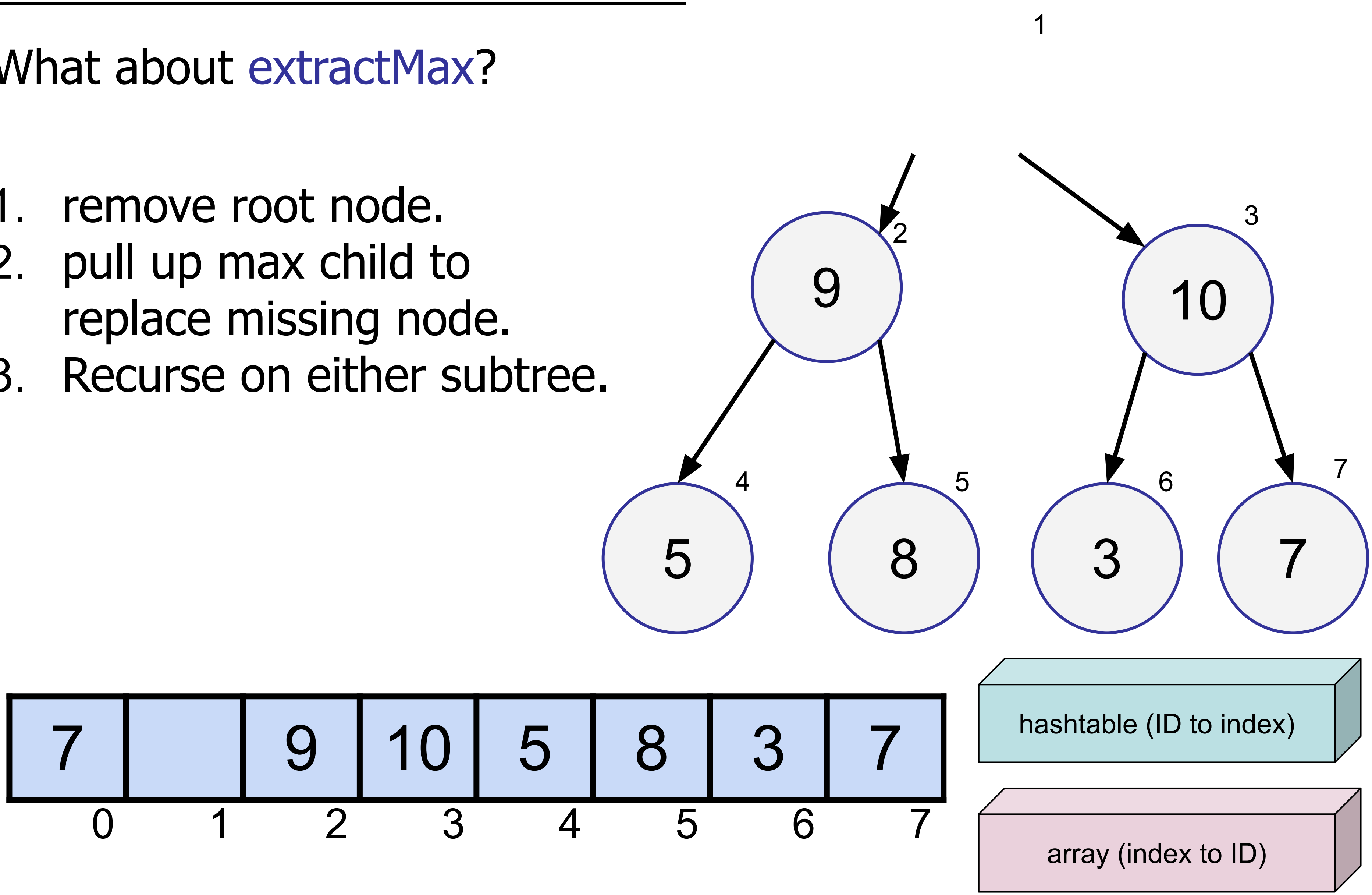
7	11	9	10	5	8	3	7
0	1	2	3	4	5	6	7



Binary Heaps:

What about extractMax?

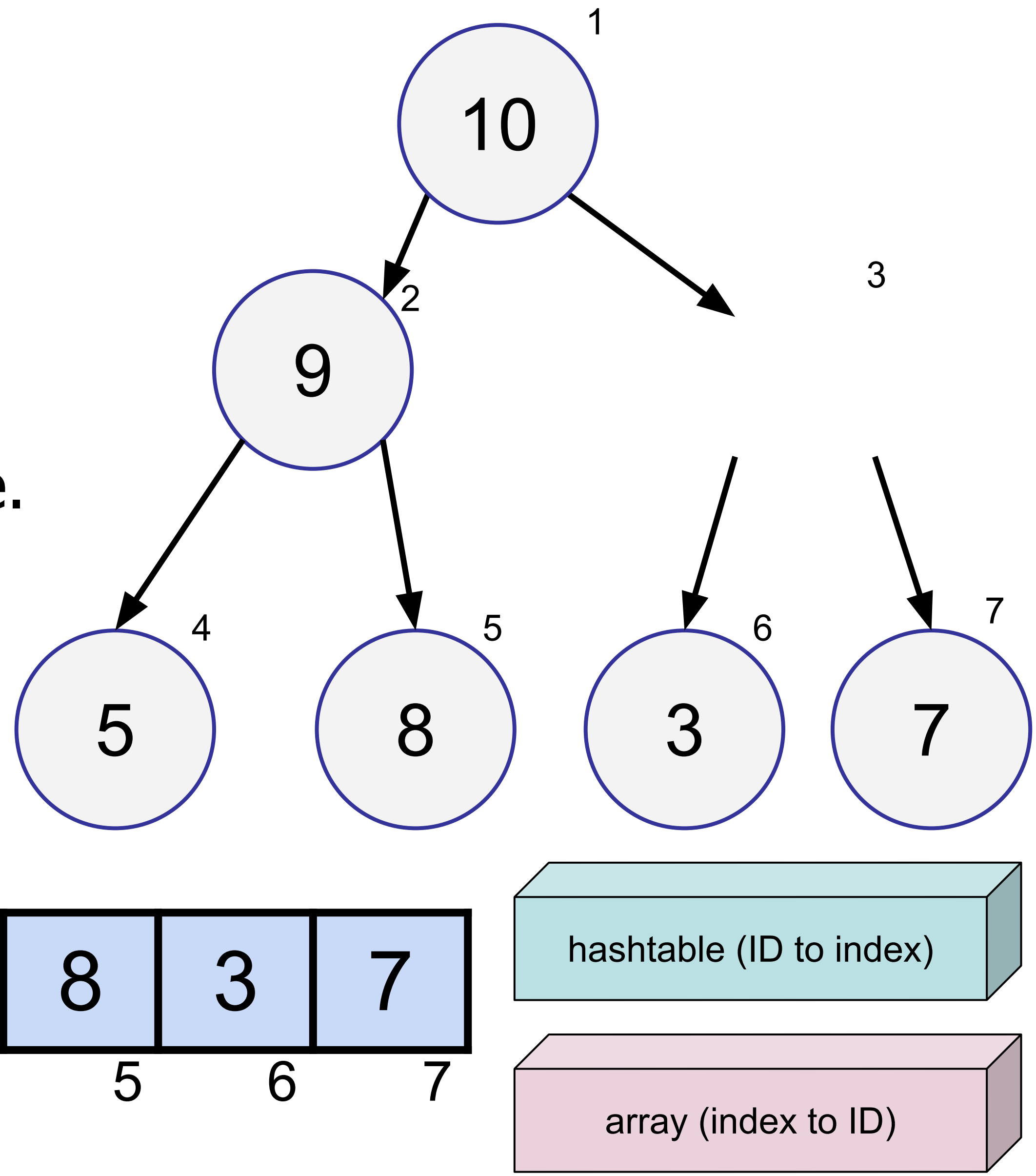
1. remove root node.
2. pull up max child to replace missing node.
3. Recurse on either subtree.



Binary Heaps:

What about extractMax?

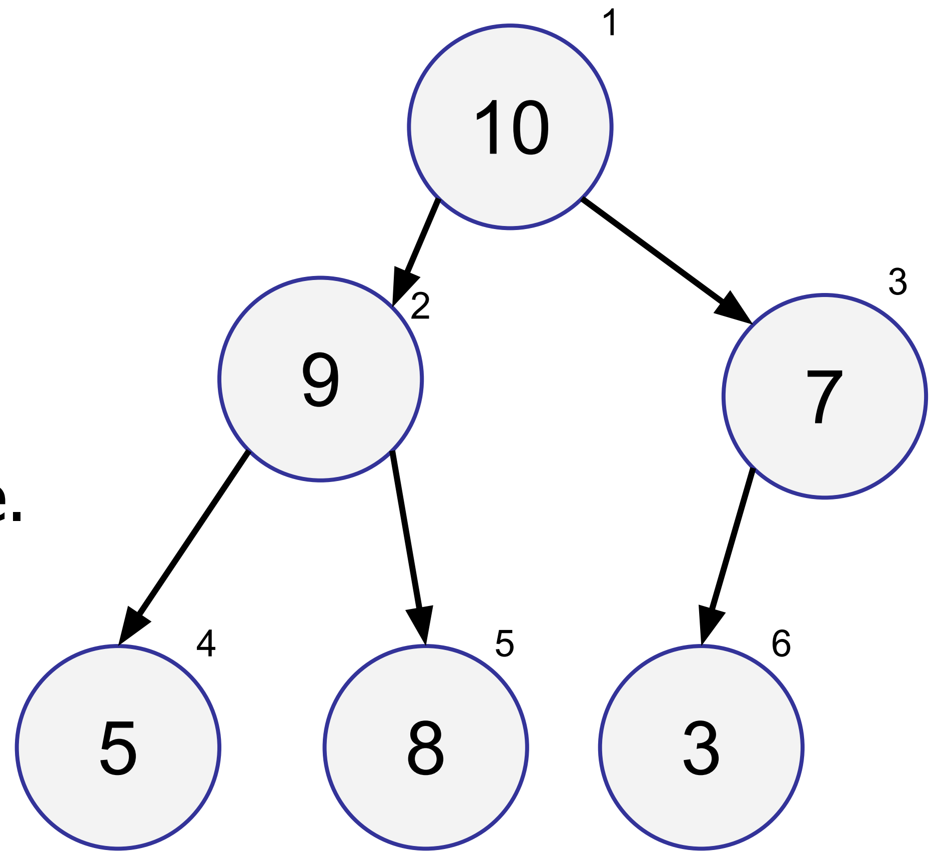
1. remove root node.
2. pull up max child to replace missing node.
3. Recurse on either subtree.

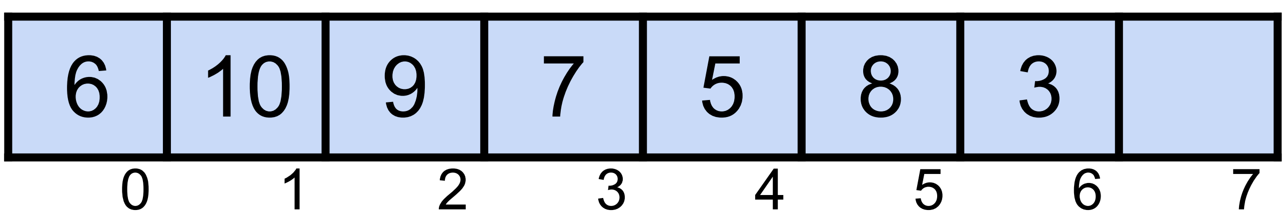


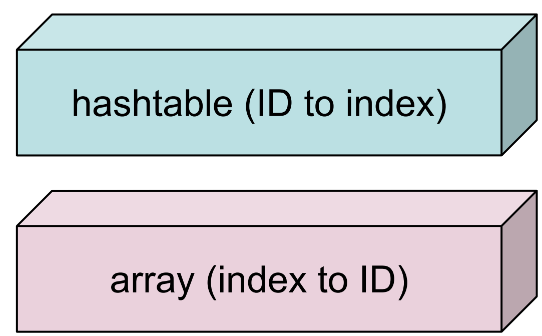
Binary Heaps:

What about extractMax?

1. remove root node.
2. pull up max child to replace missing node.
3. Recurse on either subtree.



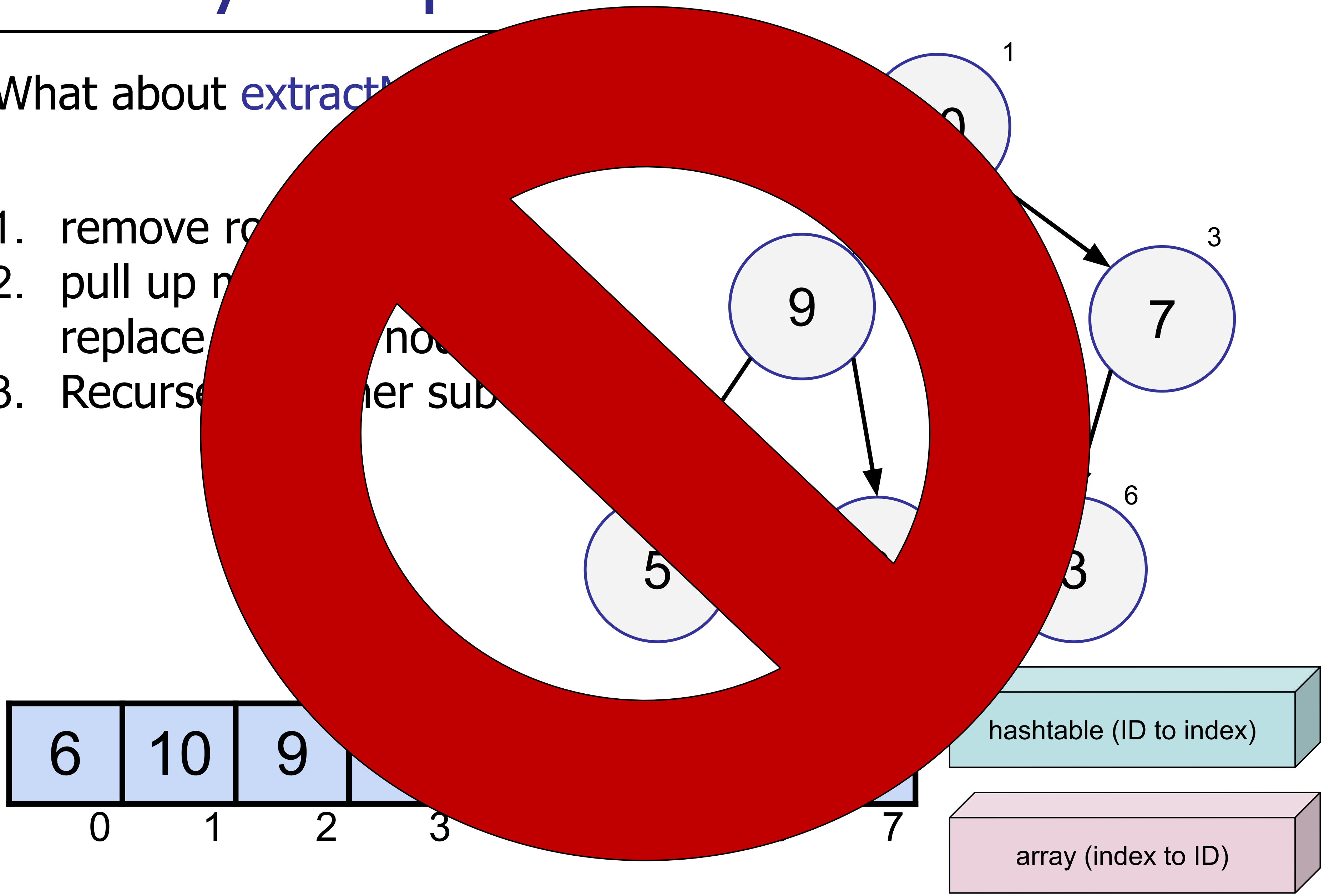




Binary Heaps:

What about extractM

1. remove ro
2. pull up n
   replace no
3. Recurse her sub

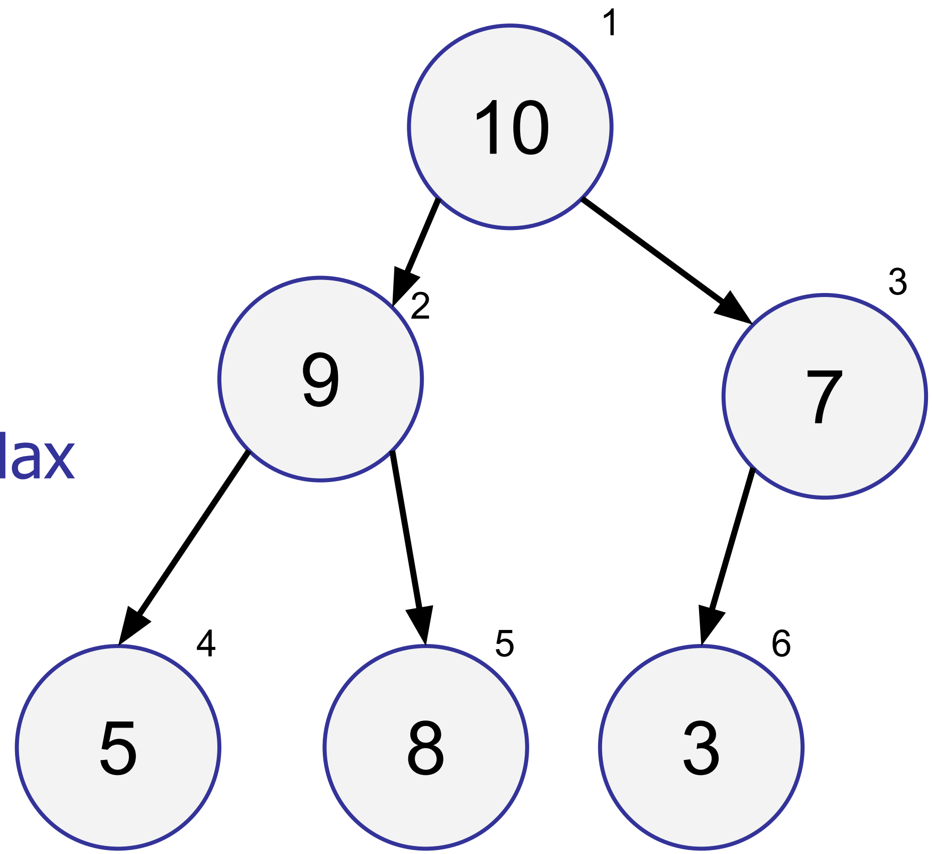


Binary Heaps:

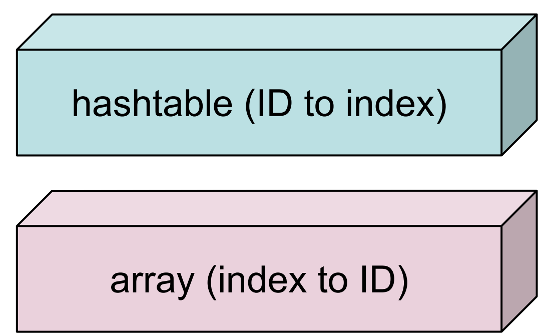
What about extractMax?

Except this is wrong!

What happens if we extractMax one more time?



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

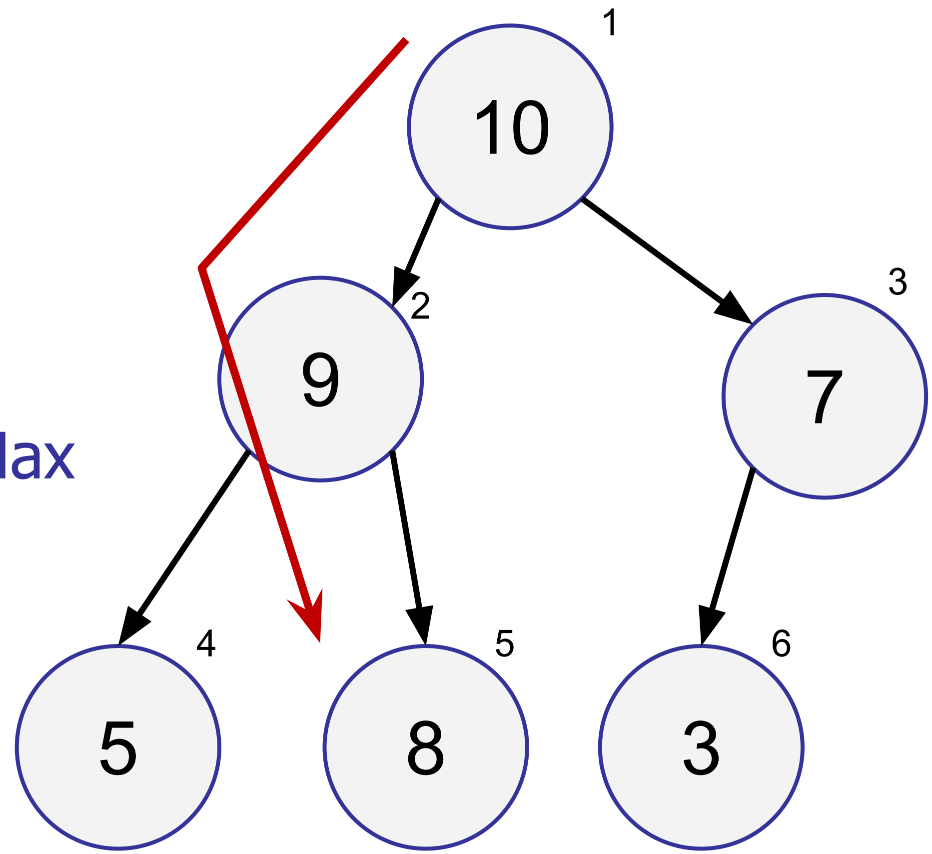


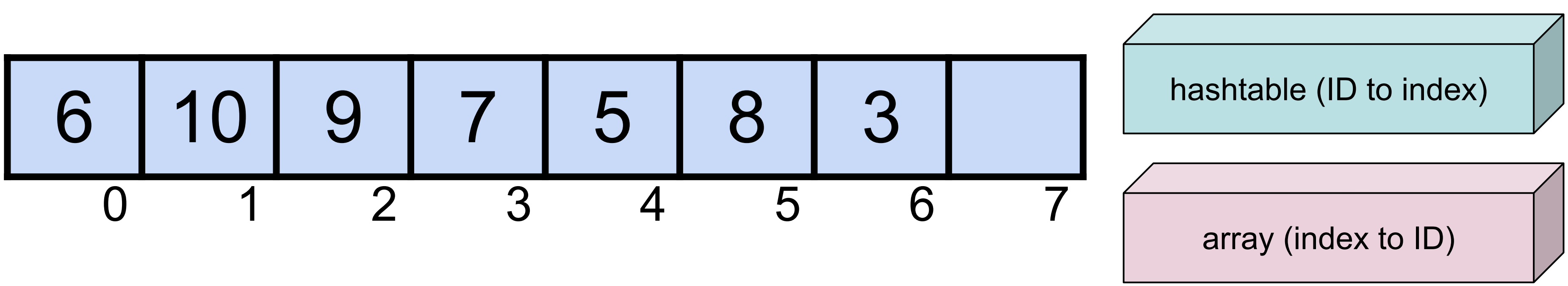
Binary Heaps:

What about extractMax?

Except this is wrong!

What happens if we extractMax one more time?





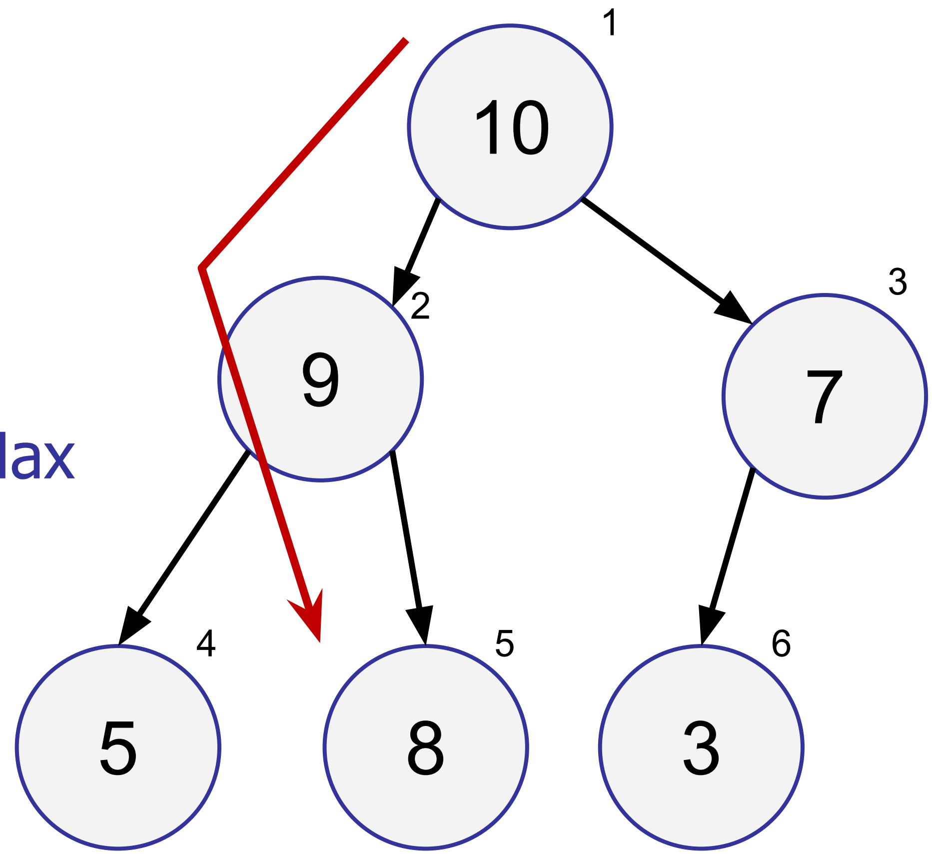
Binary Heaps:

What about extractMax?

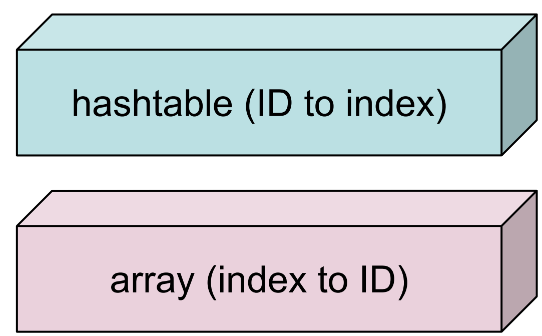
Except this is wrong!

What happens if we extractMax one more time?

The shape invariant is violated!



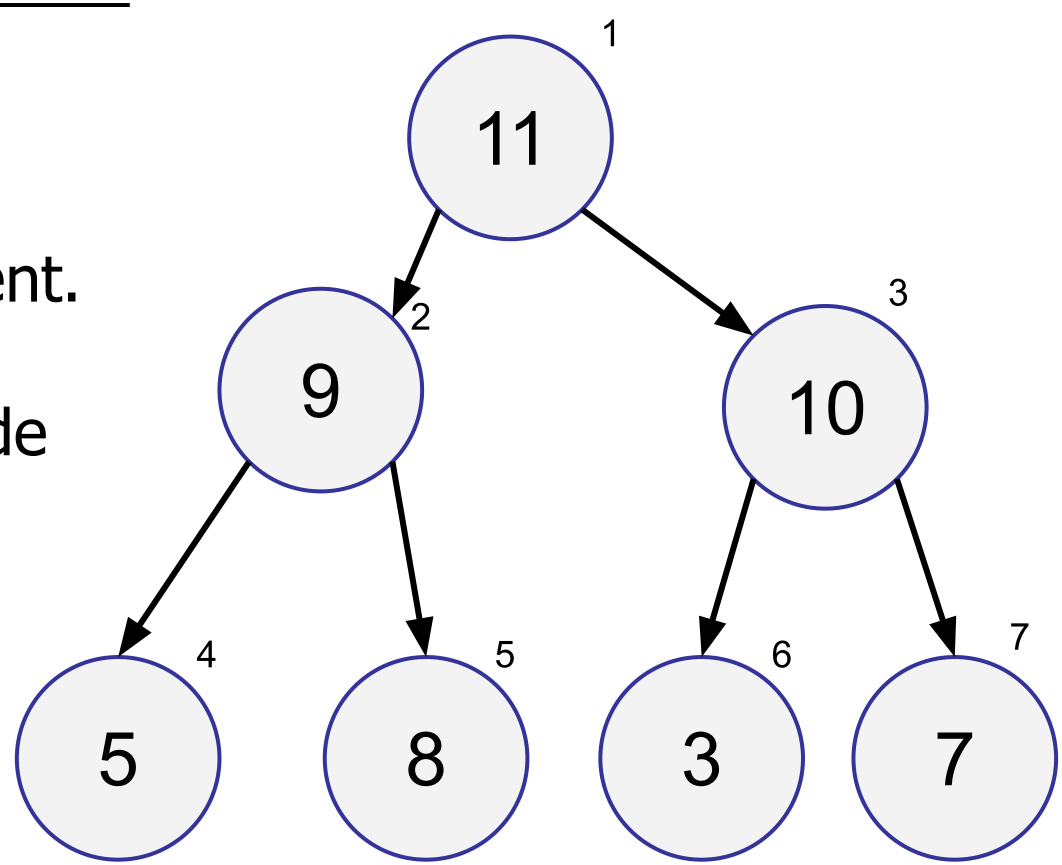
6	10	9	7	5	8	3	
0	1	2	3	4	5	6	7



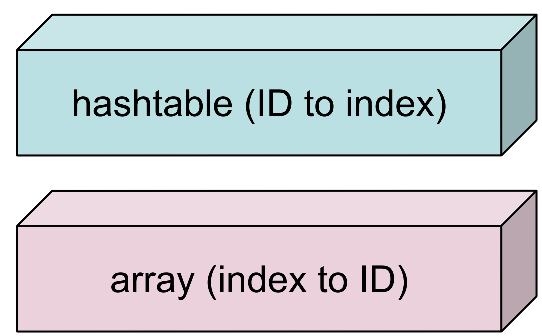
Binary Heaps:

Fix:

1. swap root with last element.
2. remove new last element
3. bubble down the root node



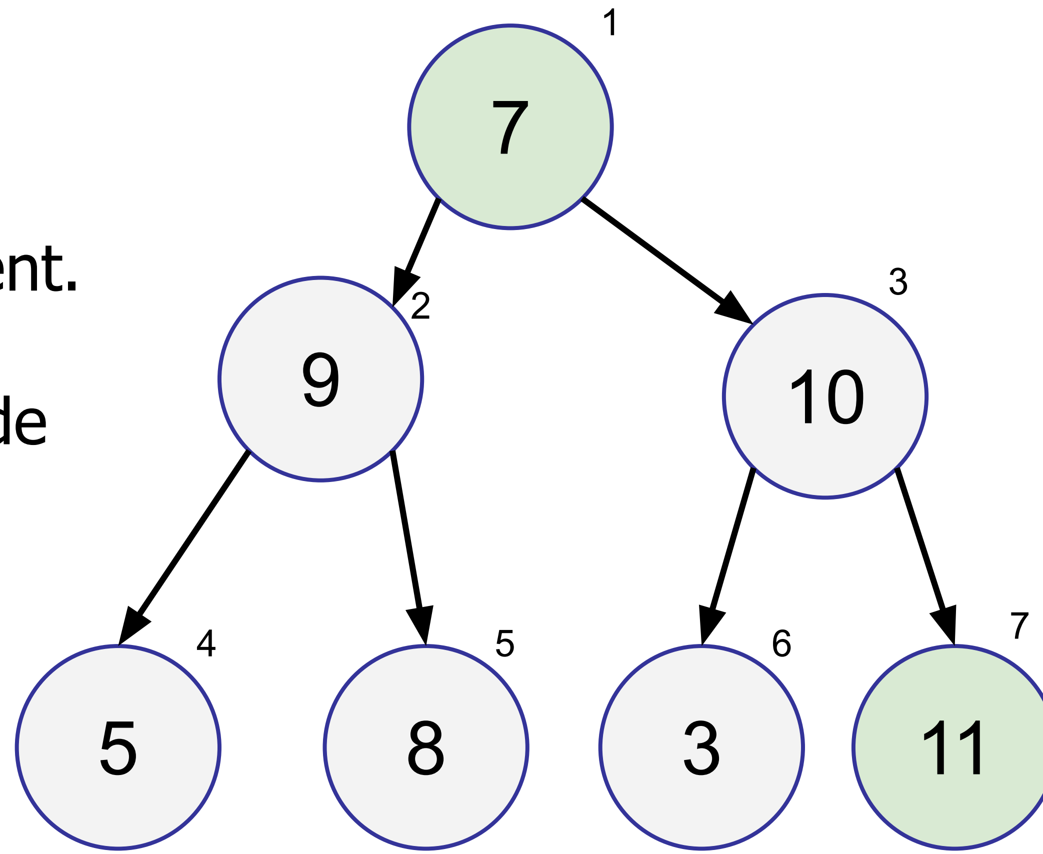
7	11	9	10	5	8	3	7
0	1	2	3	4	5	6	7



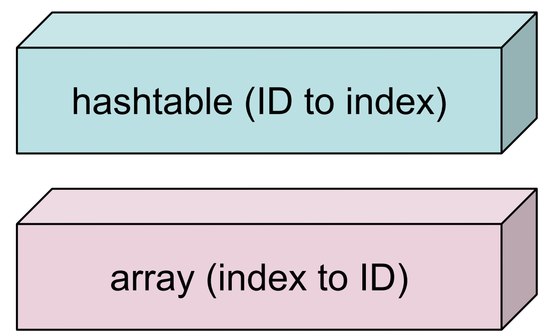
Binary Heaps:

Fix:

1. swap root with last element.
2. remove new last element
3. bubble down the root node



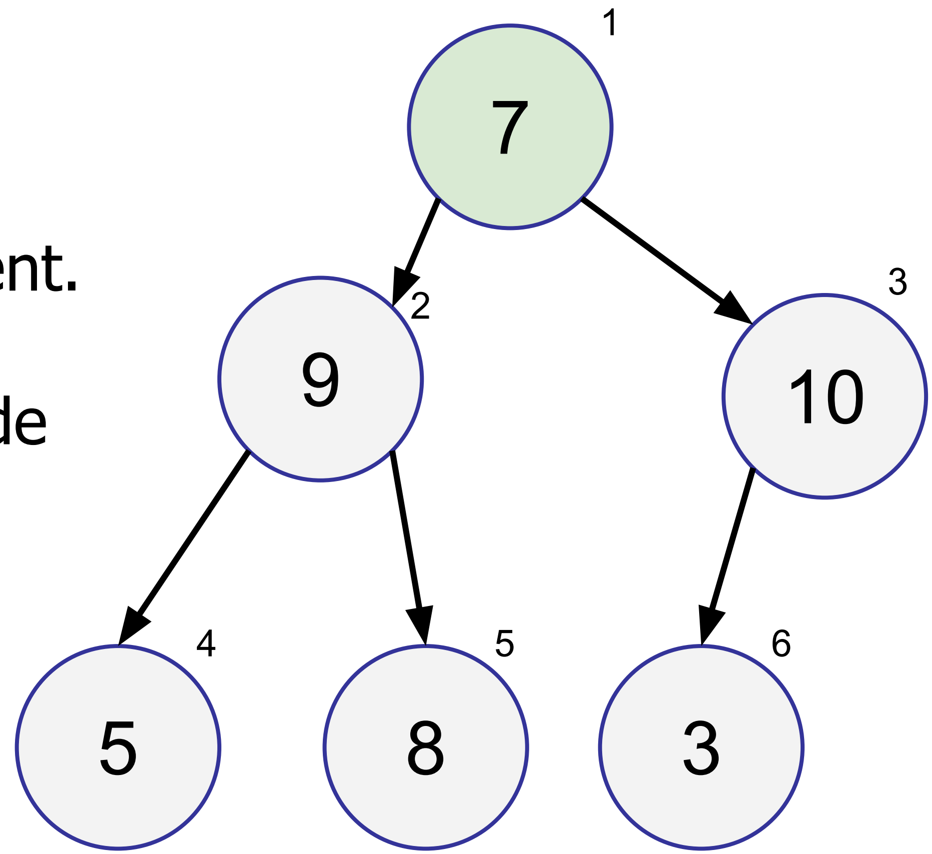
7	7	9	10	5	8	3	11
0	1	2	3	4	5	6	7

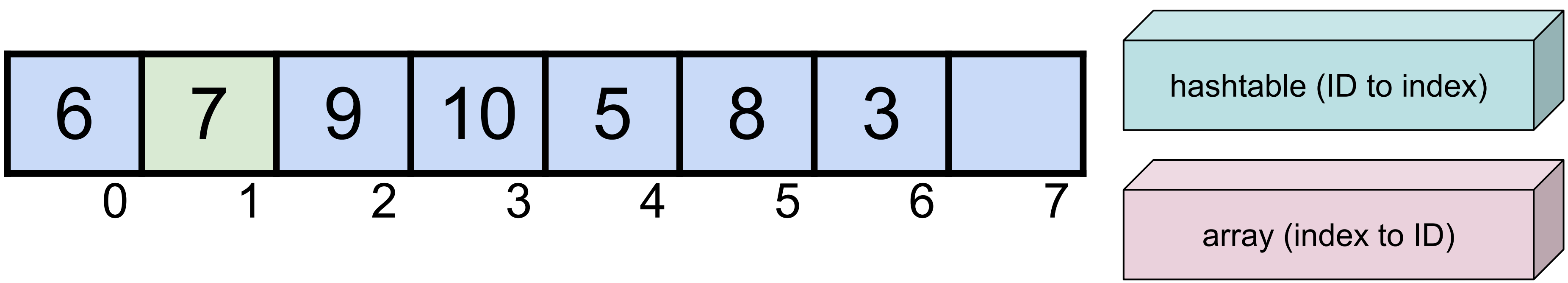


Binary Heaps:

Fix:

1. swap root with last element.
2. remove new last element
3. bubble down the root node

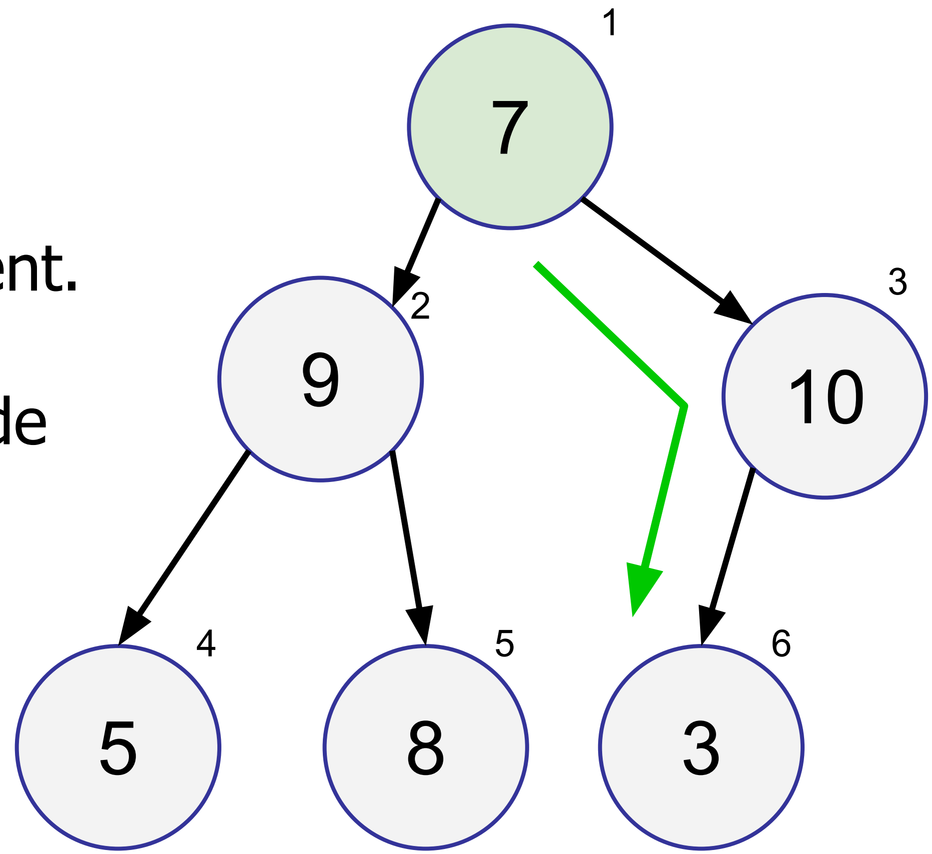


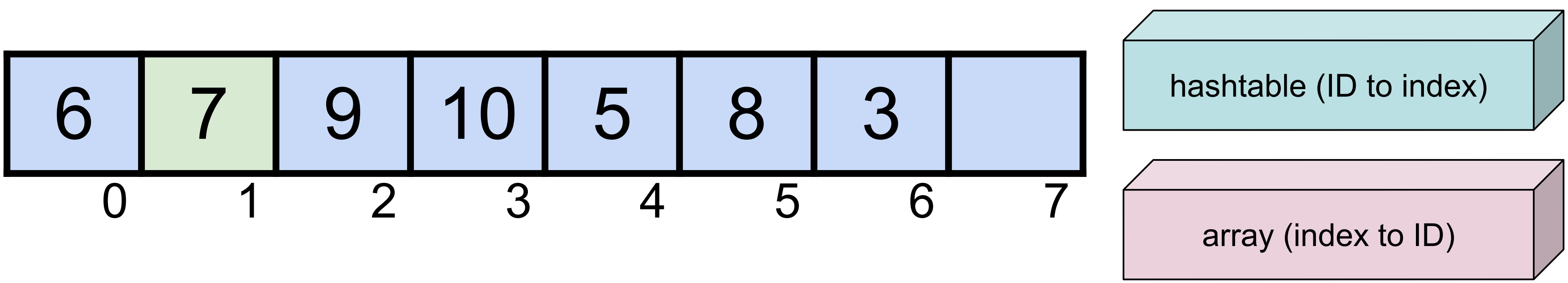


Binary Heaps:

Fix:

1. swap root with last element.
2. remove new last element
3. bubble down the root node



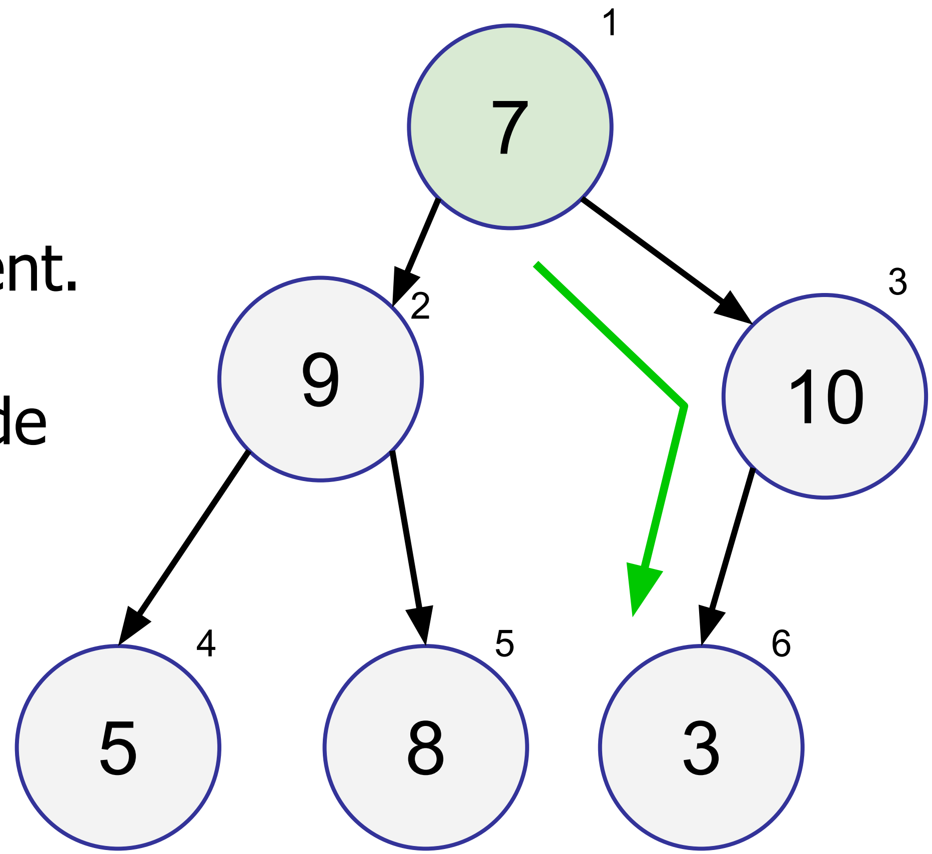


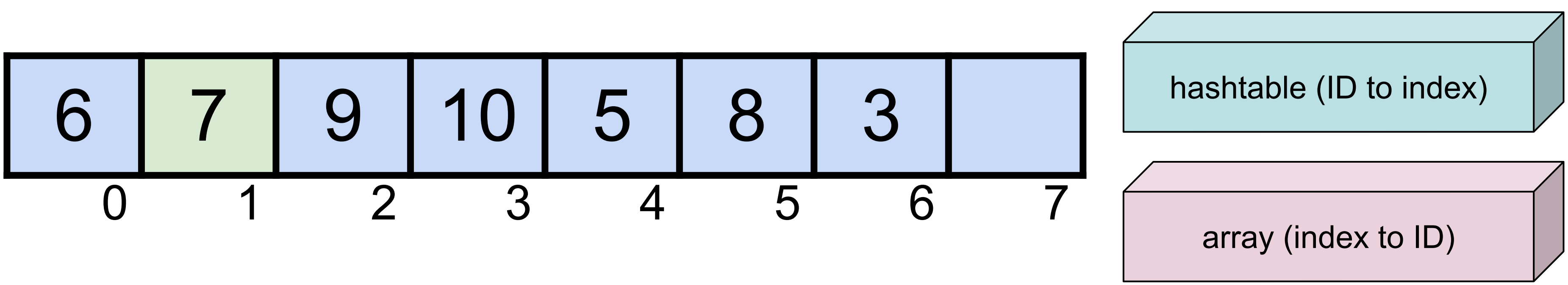
Binary Heaps:

Fix:

1. swap root with last element.
2. remove new last element
3. bubble down the root node

This maintains **both** invariants.





Binary Heaps:

One more advantage about binary heaps:

Given an array of n elements, we can turn it into a binary heap in O(n) time!

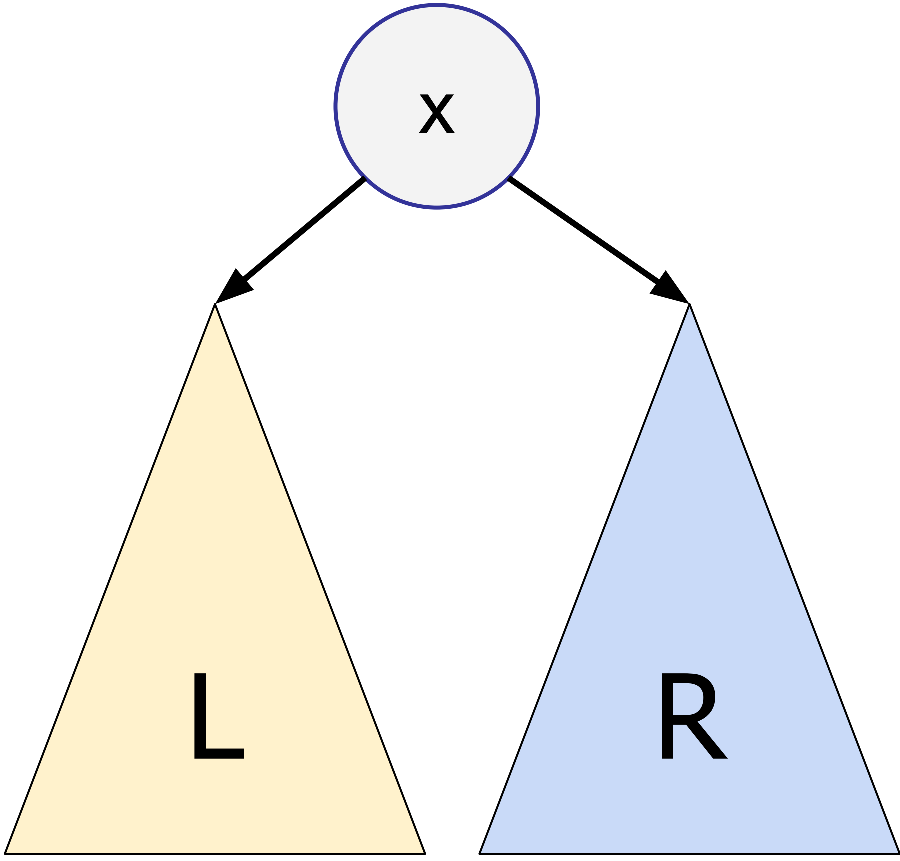
We cannot build a balanced binary search tree from an array of n elements in O(n) time! (Wait till CS3230 for a proof of this).

Binary Heaps:

Setup:

Let x be the parent of two subheaps L and R.

I.e. L and R satisfy **both** the shape and heap invariant.



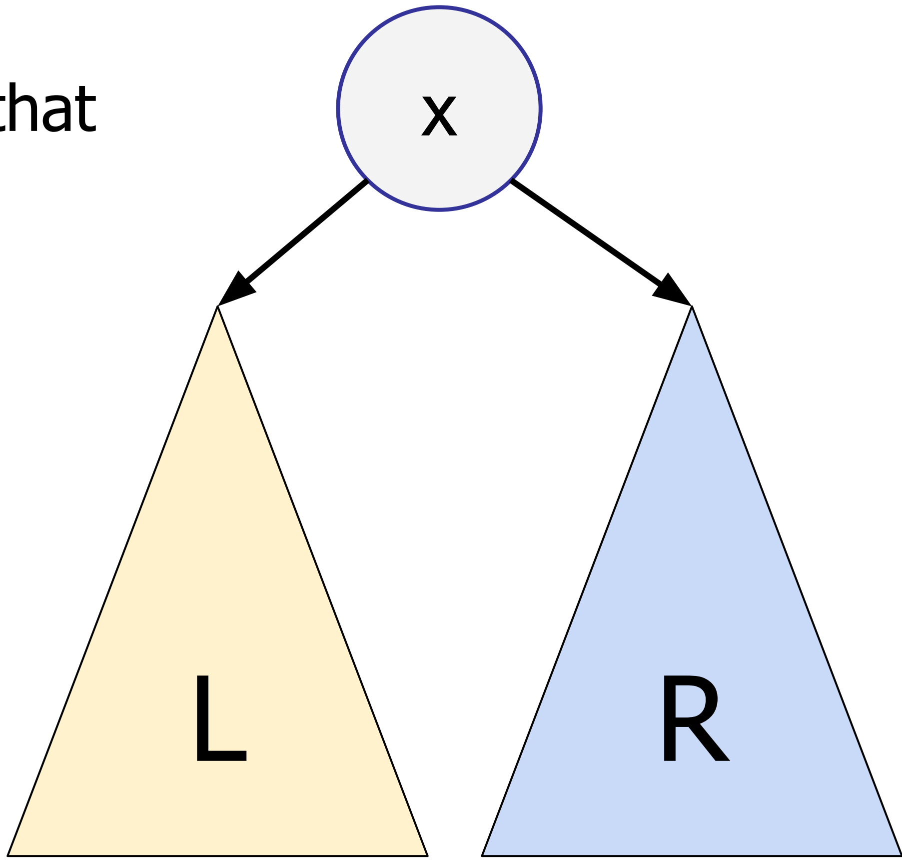
Binary Heaps:

Setup:

Let x be the parent of two subheaps L and R.

I.e. L and R satisfy **both** the shape and heap invariant.

Want to show via strong induction that
if we bubbleDown **x**, then the
resulting heap satisfies both
shape and heap invariants.



Binary Heaps:

Setup:

Let x be the parent of two subheaps L and R.

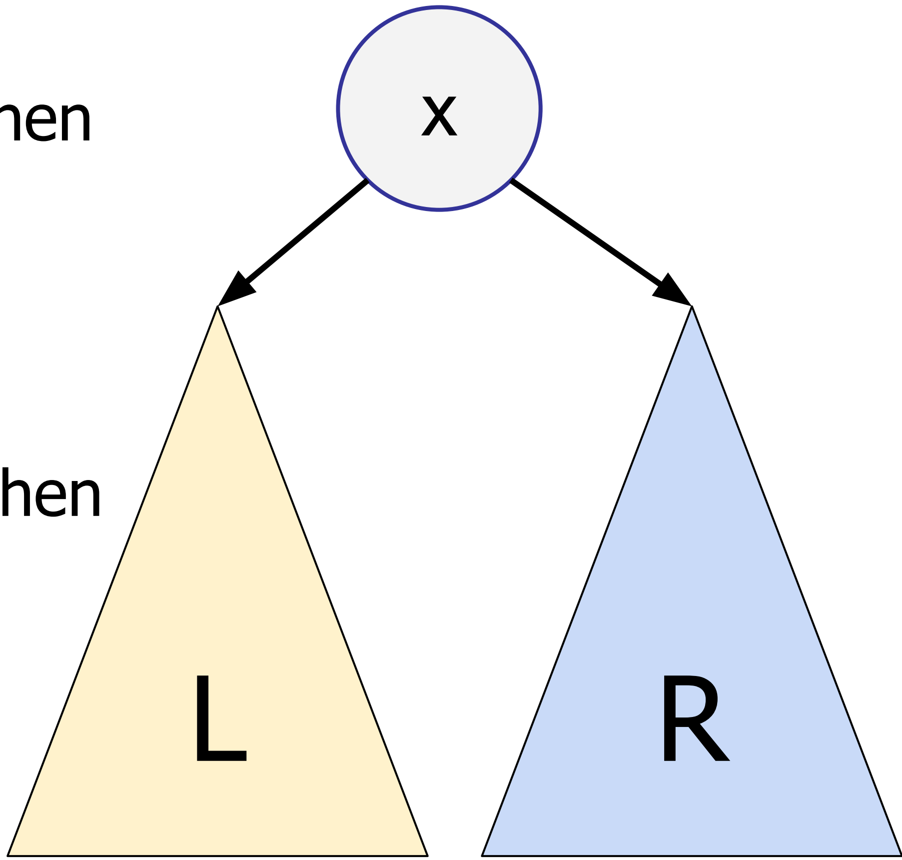
I.e. L and R satisfy **both** the shape and heap invariant.

Assume for strong induction that:

if x were the root of subtree **L** then bubbling down x would create a heap out of subtree **L**

and

if x were the root of subtree **R** then bubbling down x would create a heap out of subtree **R**



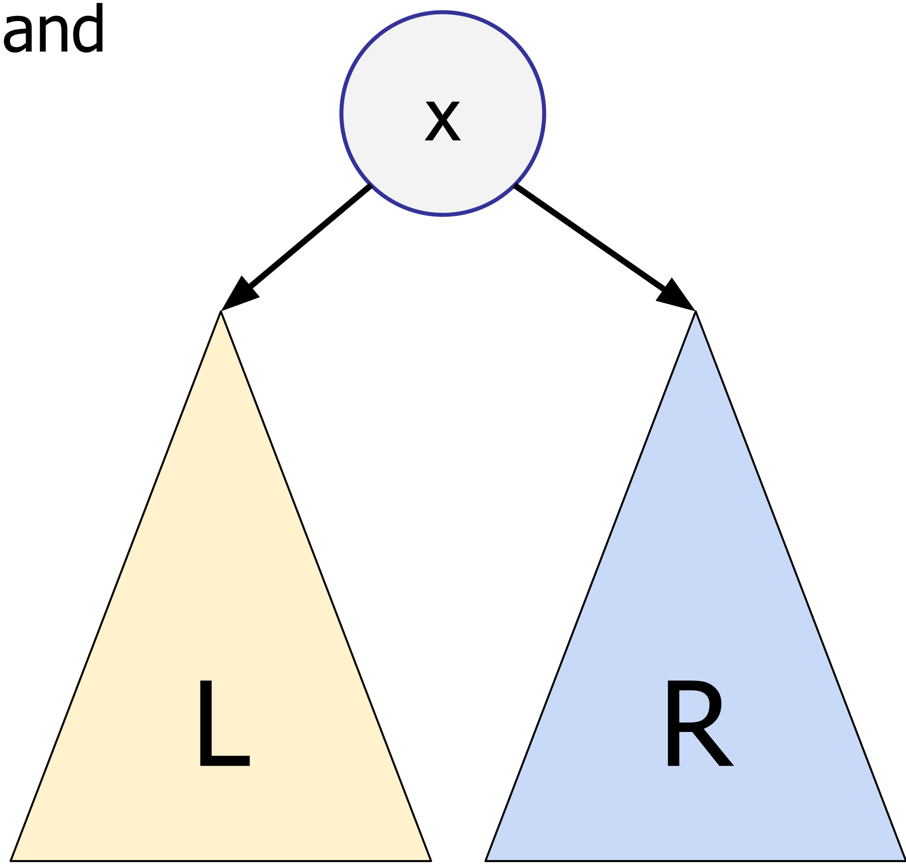
Binary Heaps:

Setup:

Let x be the parent of two subheaps L and R.

I.e. L and R satisfy **both** the shape and heap invariant.

Then if x is larger than both **L**.root and **R**.root, then the tree rooted at x is also a heap.



Binary Heaps:

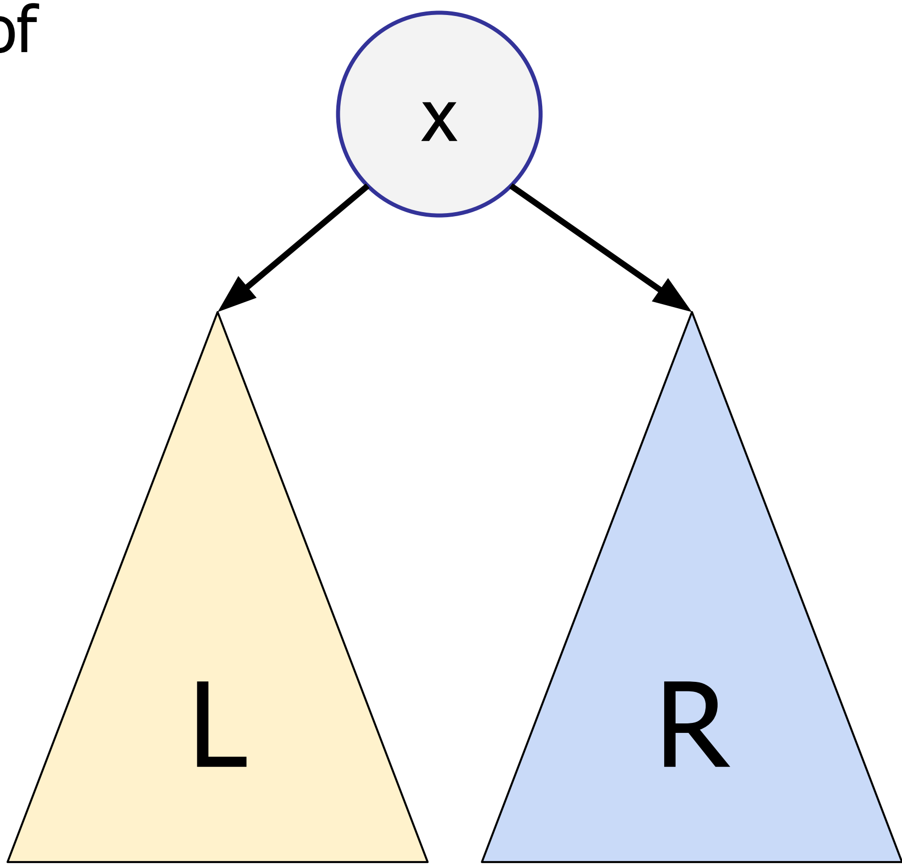
Setup:

Let x be the parent of two subheaps L and R.

I.e. L and R satisfy **both** the shape and heap invariant.

Then if x is smaller than at least 1 of its 2 children, then by swapping **x** with the larger child,

due to our assumption, recursively calling bubbleDown on the respective subtree will ensure both **L** and **R** are heaps.



Binary Heaps:

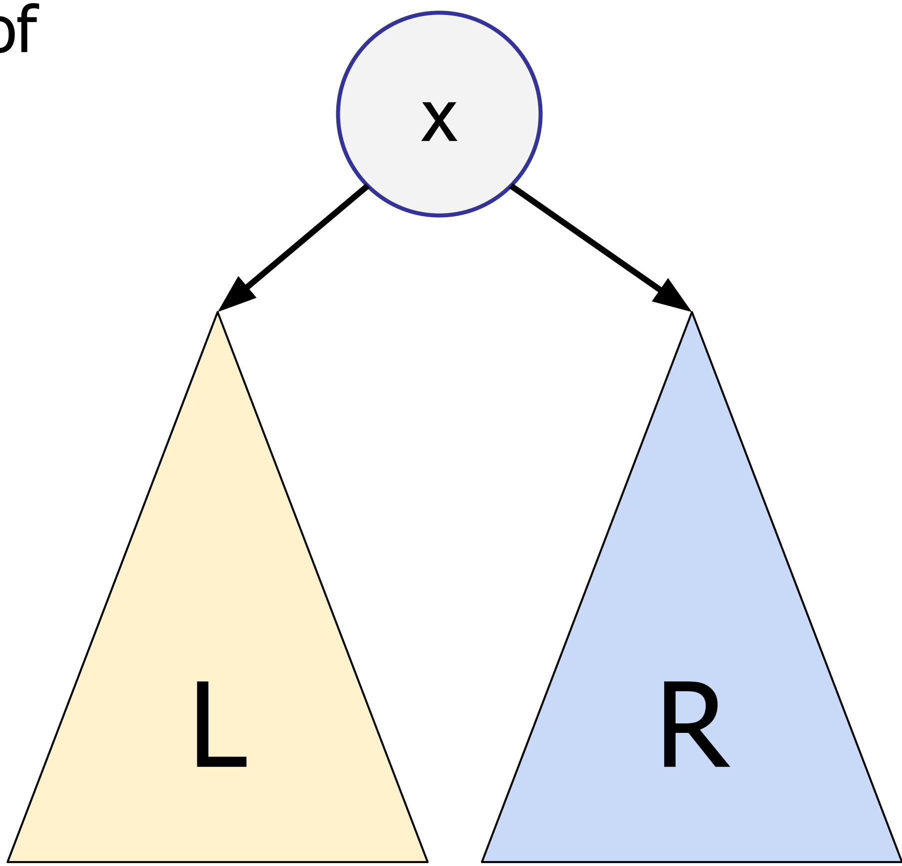
Setup:

Let x be the parent of two subheaps L and R.

I.e. L and R satisfy **both** the shape and heap invariant.

Then if x is smaller than at least 1 of its 2 children, then by swapping **x** with the larger child,

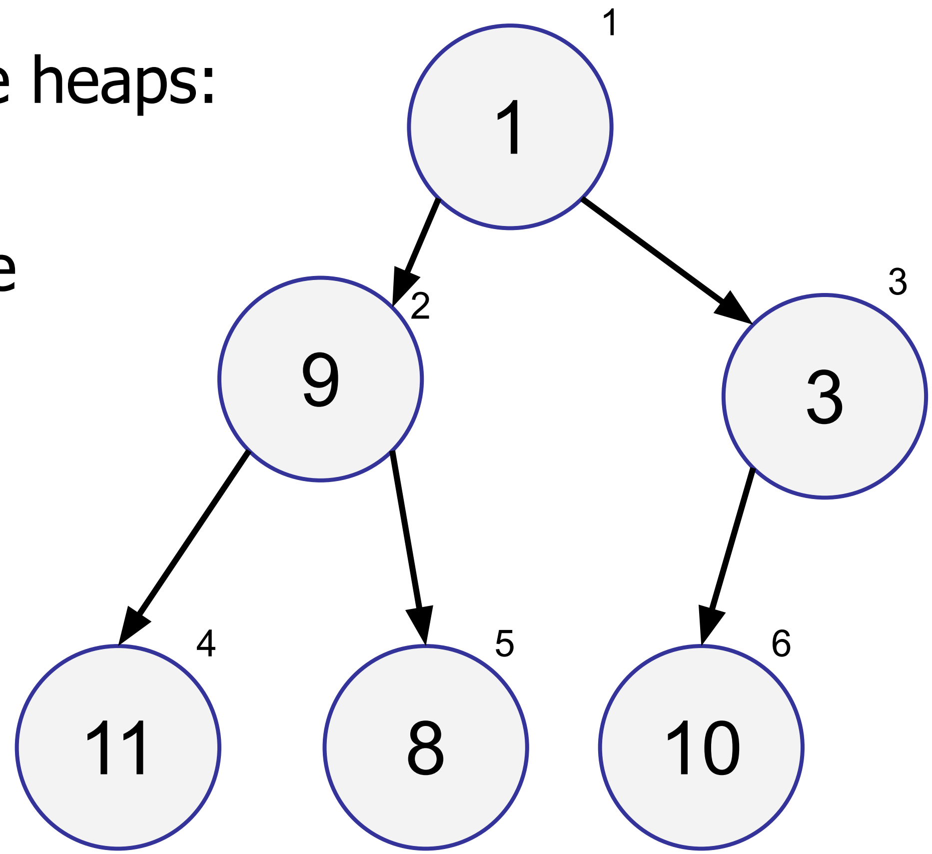
Furthermore, the new value at node x is larger than both of its children. Thus, the tree rooted at x is a heap.



Binary Heaps:

To use bubbleDown to create heaps:

Start bubbling down from the first non-leaf node.

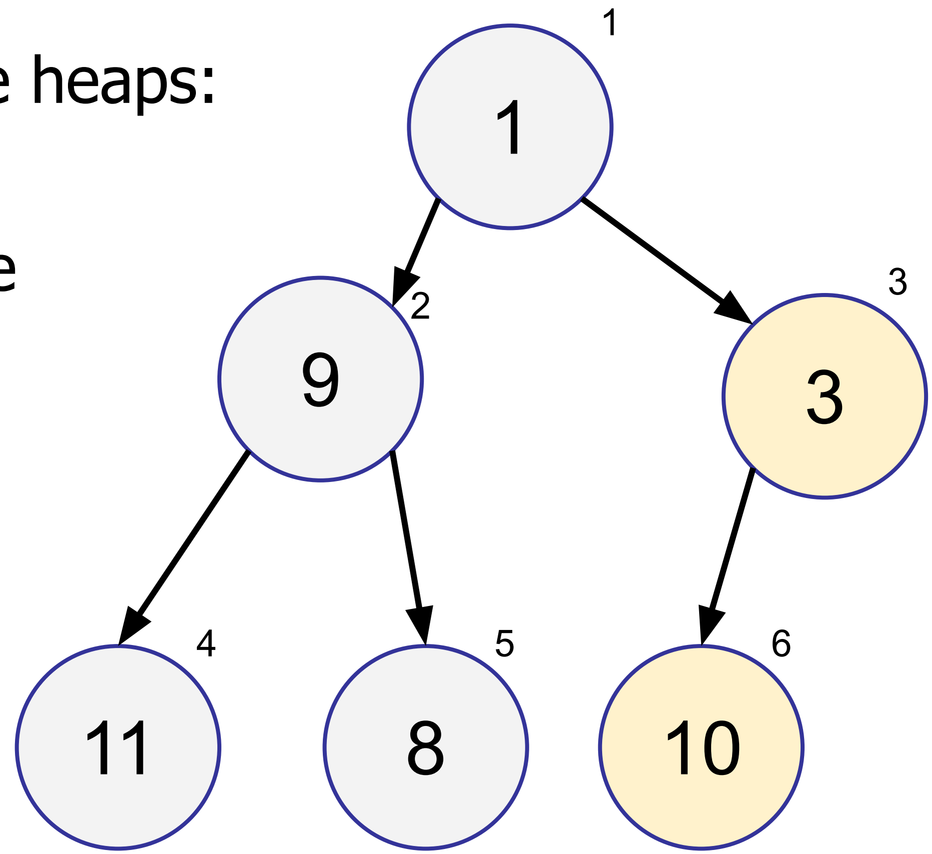


6	1	9	3	11	8	10	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Start bubbling down from the first non-leaf node.

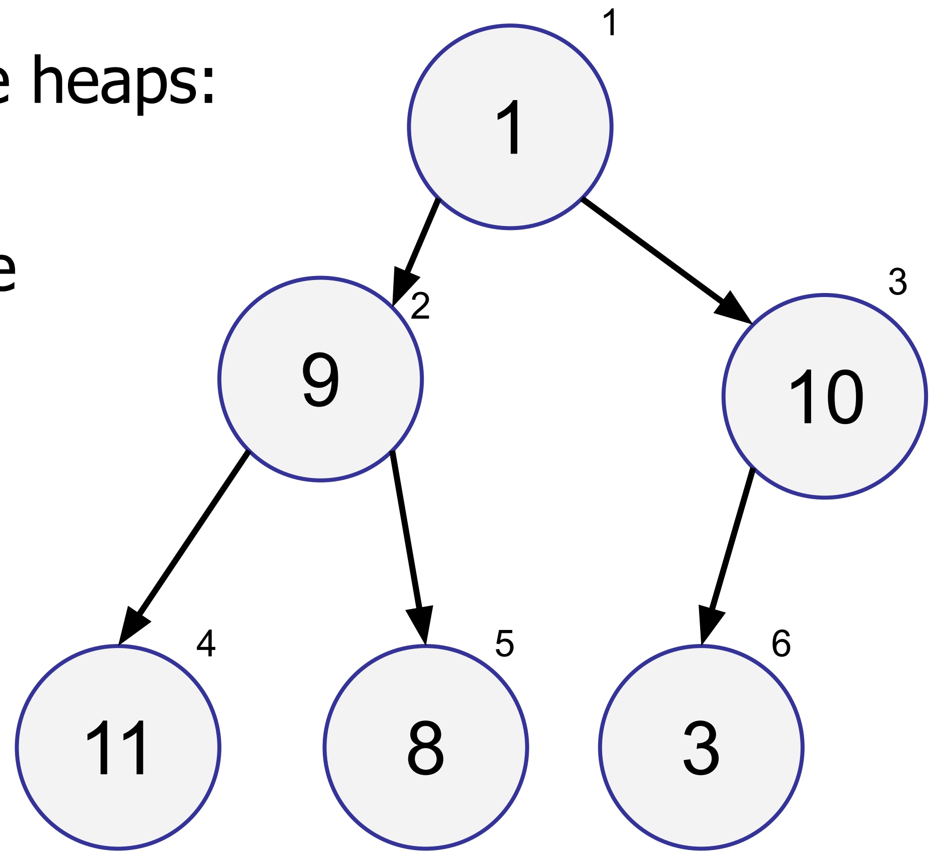


6	1	9	3	11	8	10	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Start bubbling down from the first non-leaf node.

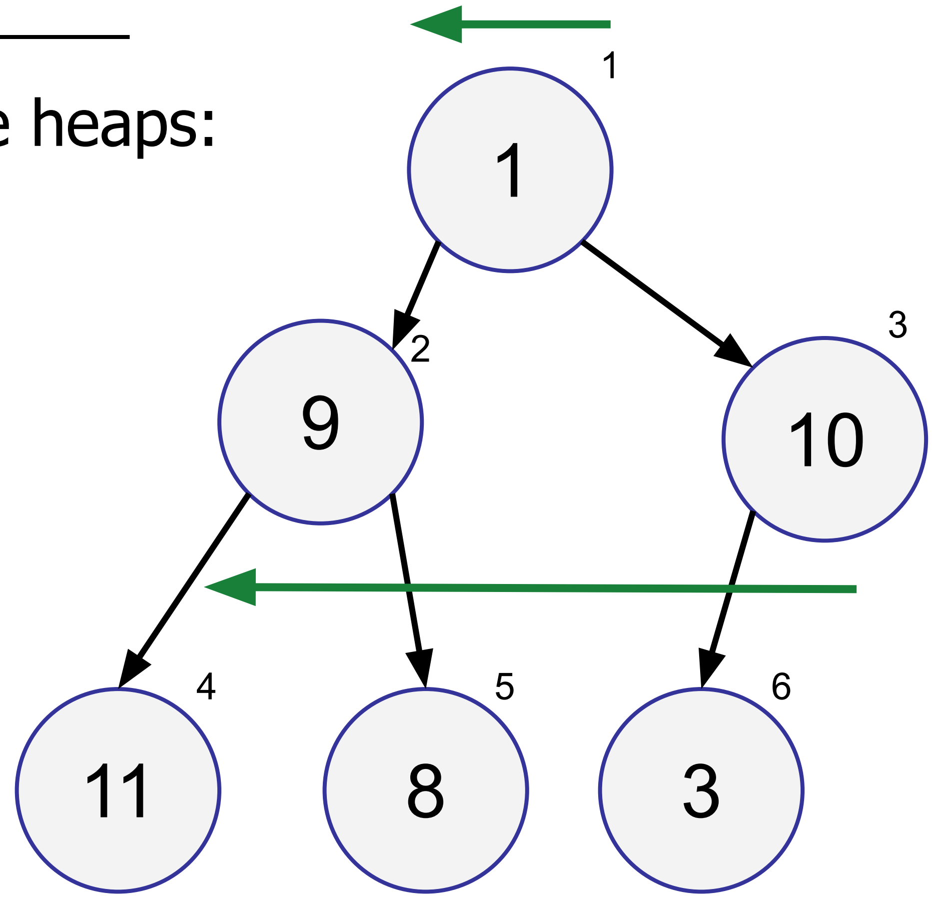


6	1	9	10	11	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Then repeat left-wards, then up the tree

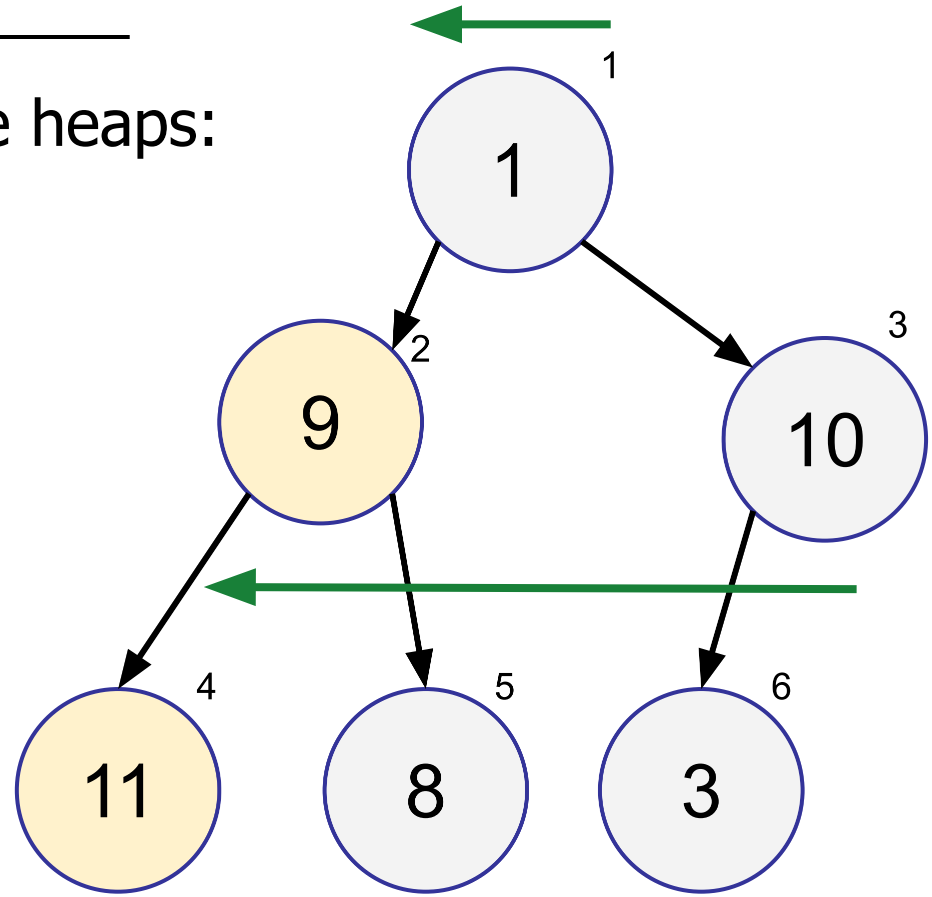


6	1	9	10	11	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Then repeat left-wards, then up the tree

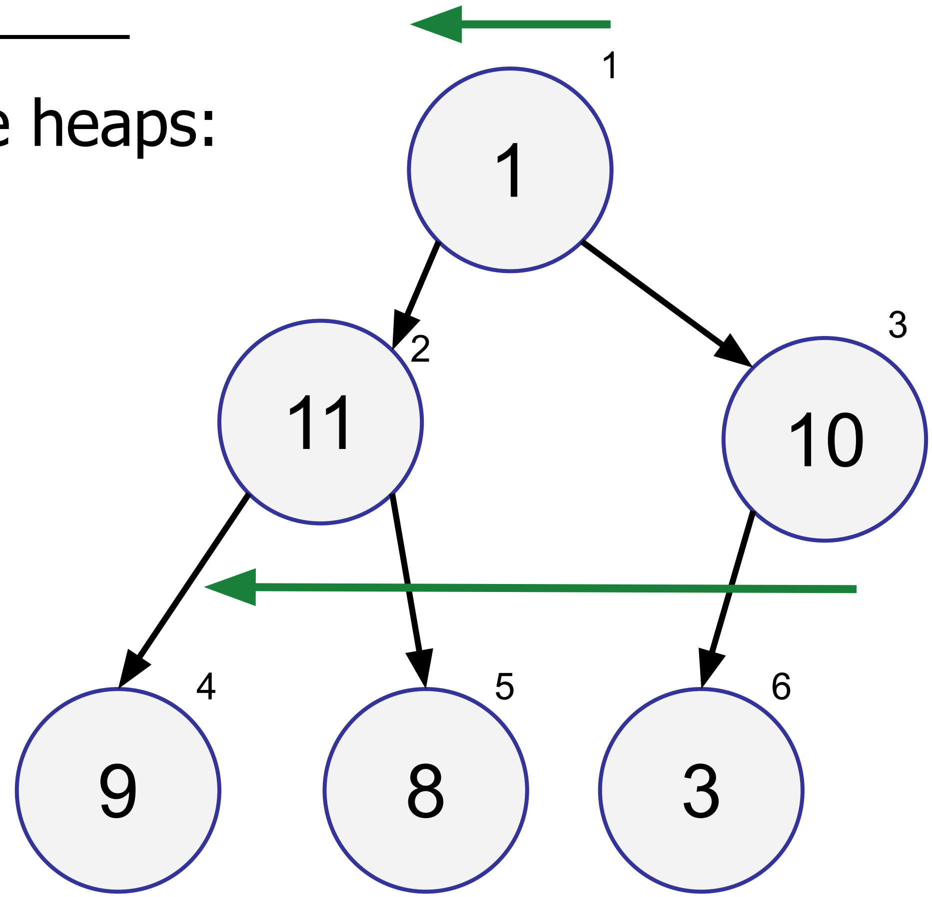


6	1	9	10	11	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Then repeat left-wards, then up the tree

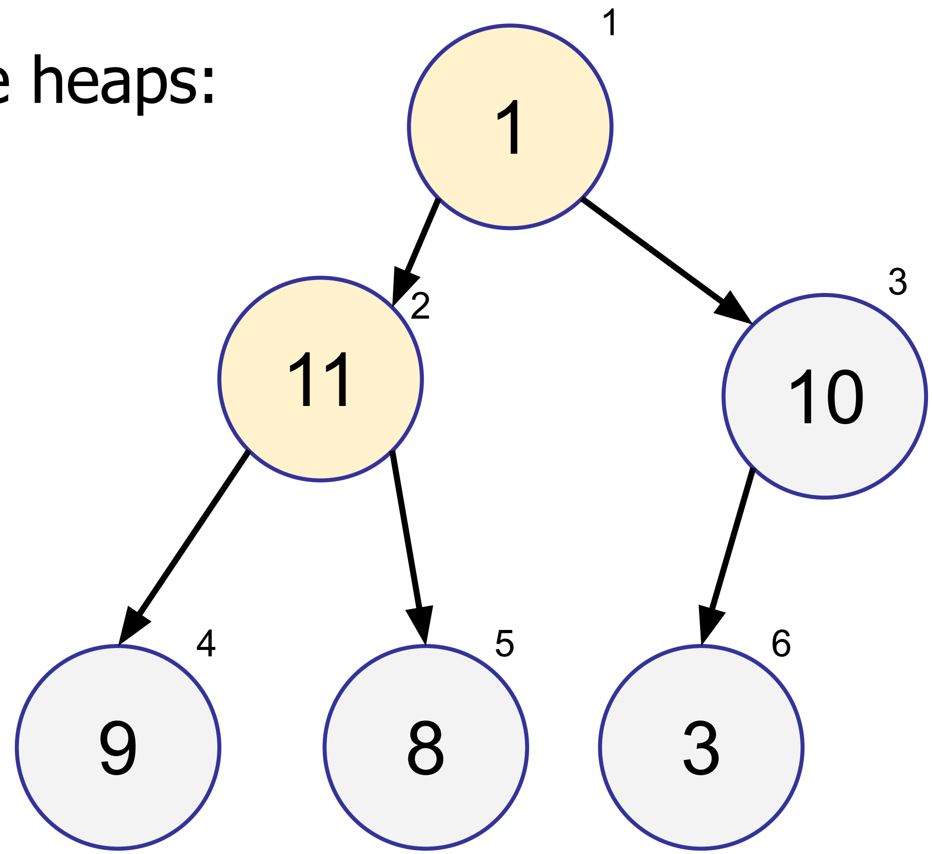


6	1	11	10	9	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Then repeat left-wards, then up the tree

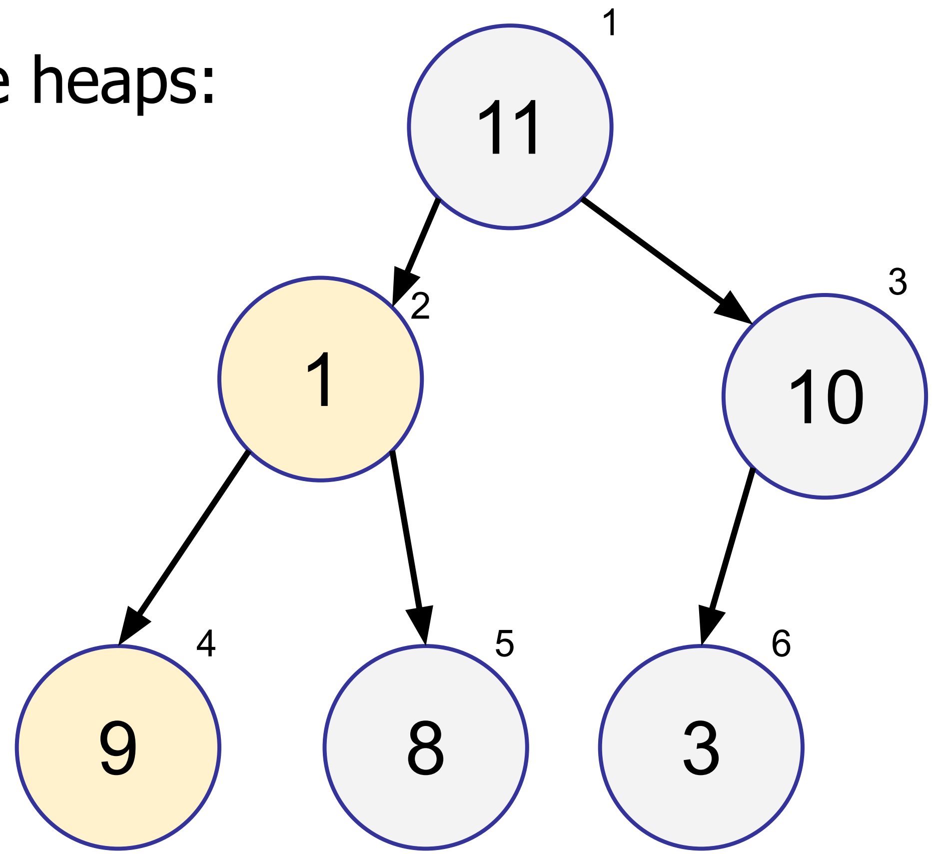


6	1	11	10	9	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

To use bubbleDown to create heaps:

Then repeat left-wards, then up the tree

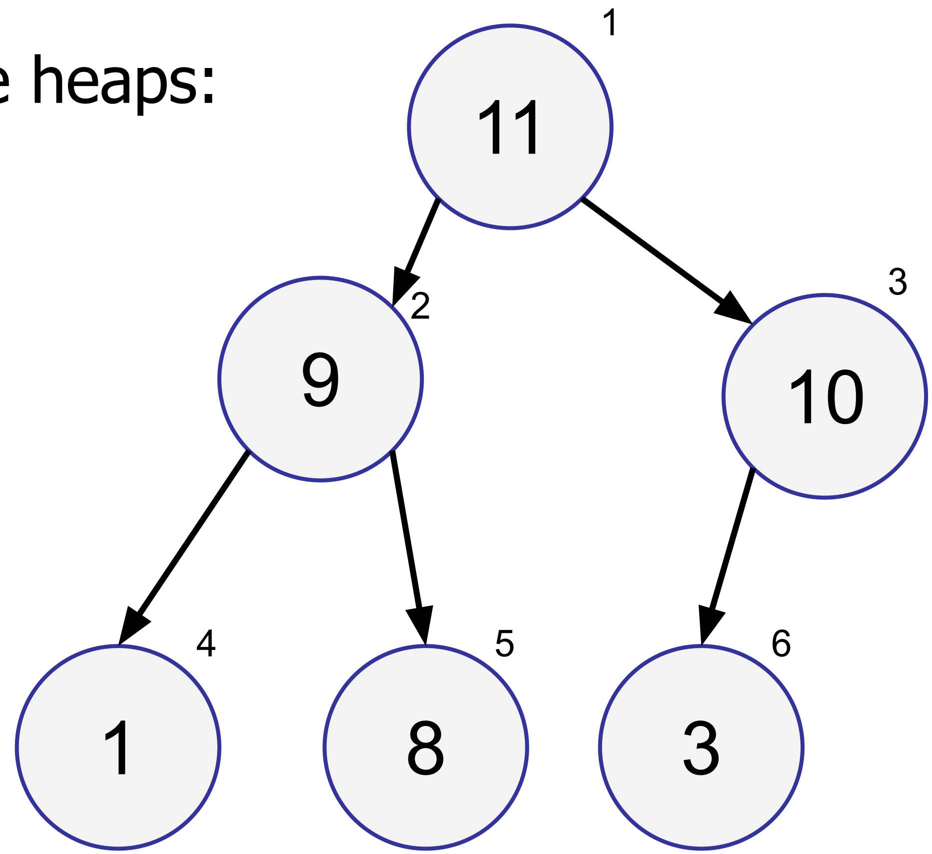


0	1	2	3	4	5	6	7
6	11	1	10	9	8	3	

Binary Heaps:

To use bubbleDown to create heaps:

Then repeat left-wards, then up the tree



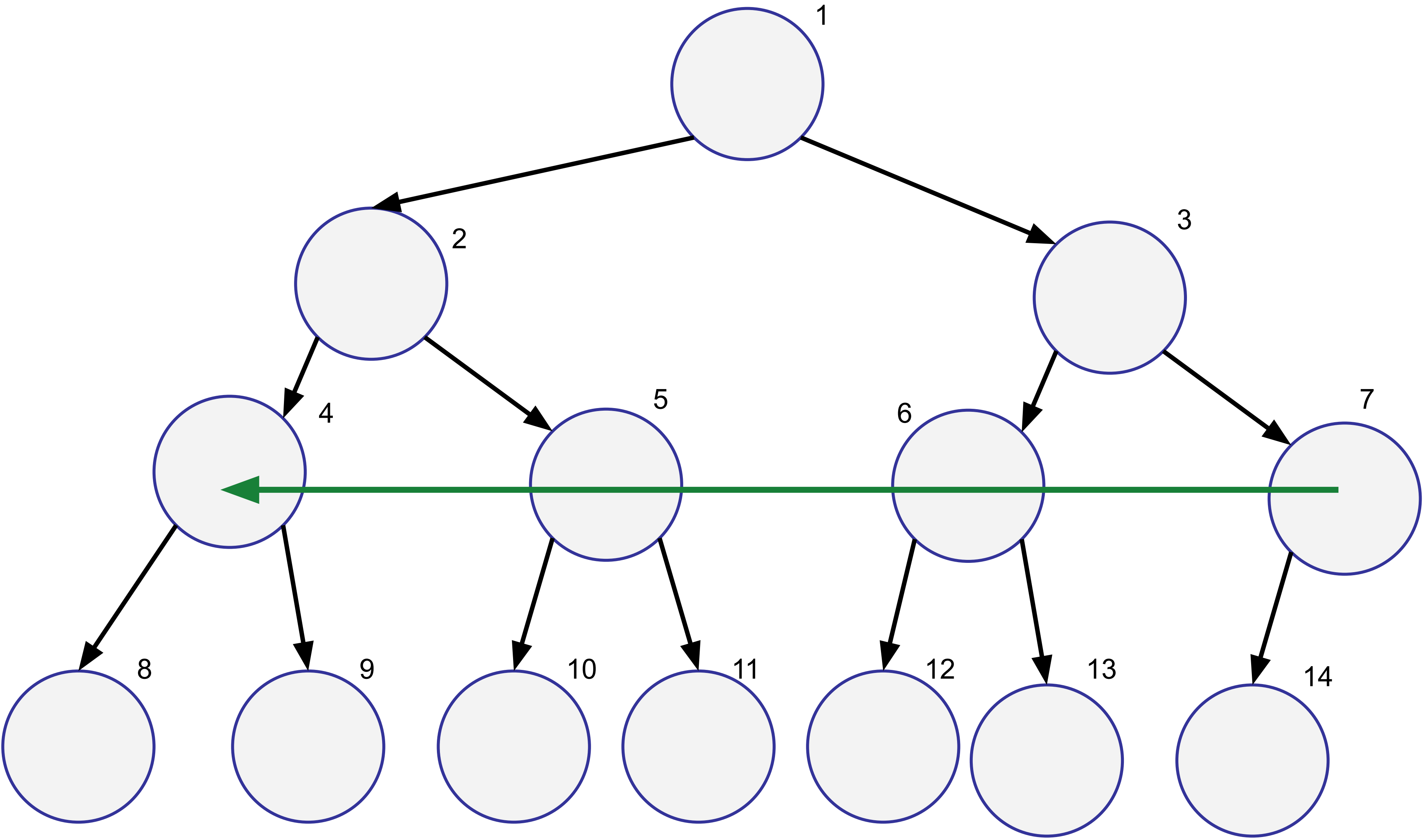
6	11	9	10	1	8	3	
0	1	2	3	4	5	6	7

Binary Heaps:

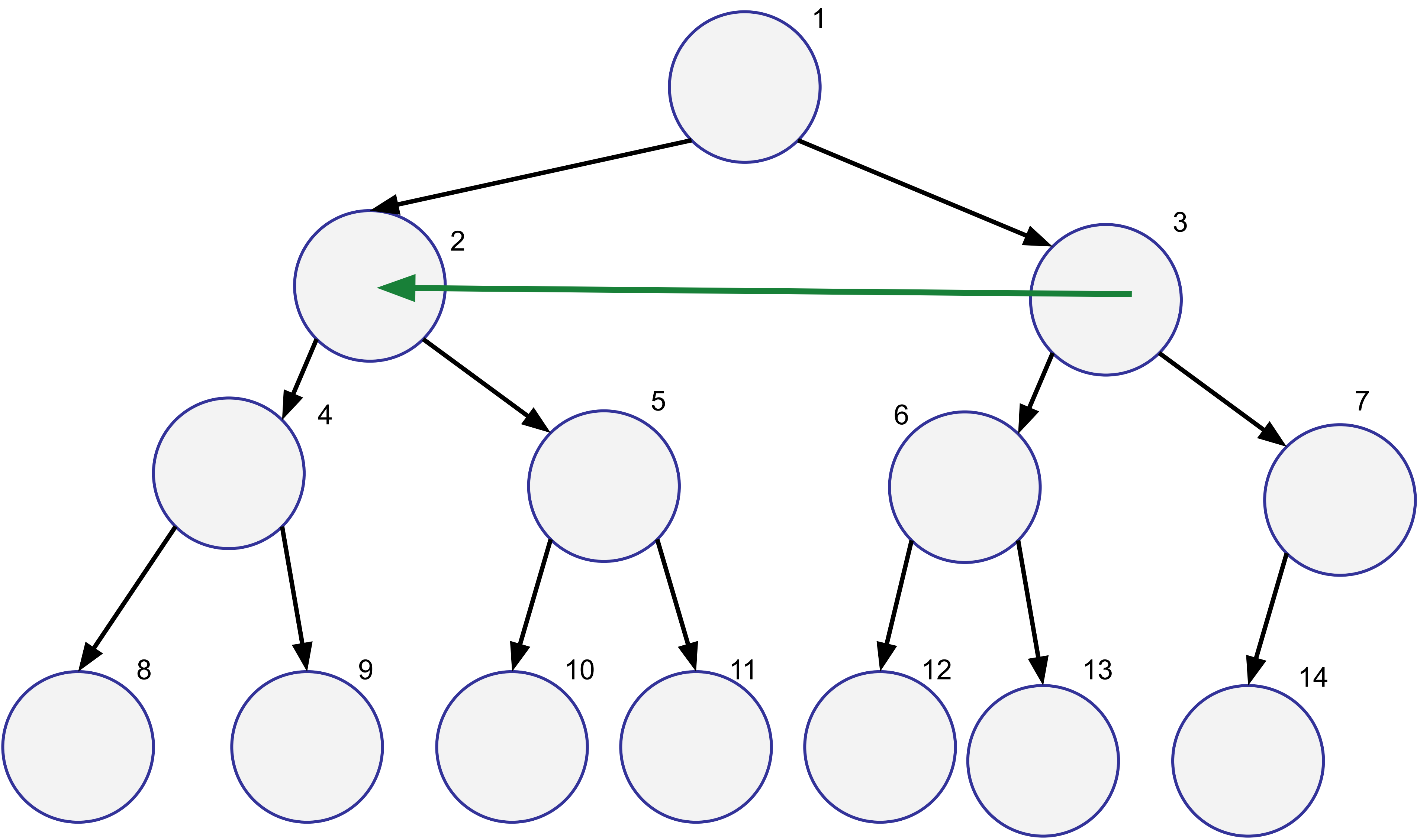
Heapify algorithm:

```
void heapify(){
  for(int idx = size / 2; idx >= 1; --idx){
    bubbleDown(idx);
  }
}
```

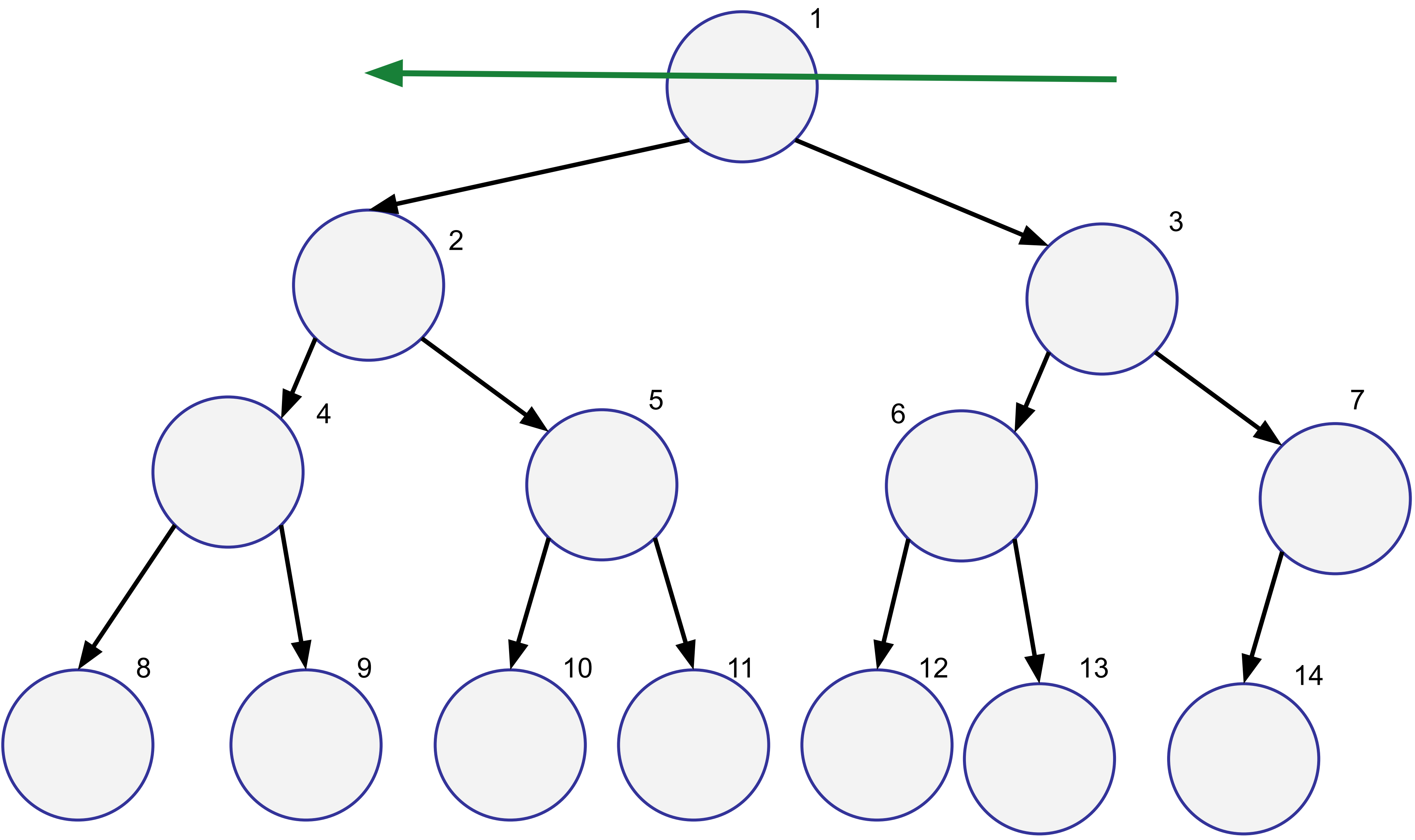
Binary Heaps:



Binary Heaps:



Binary Heaps:

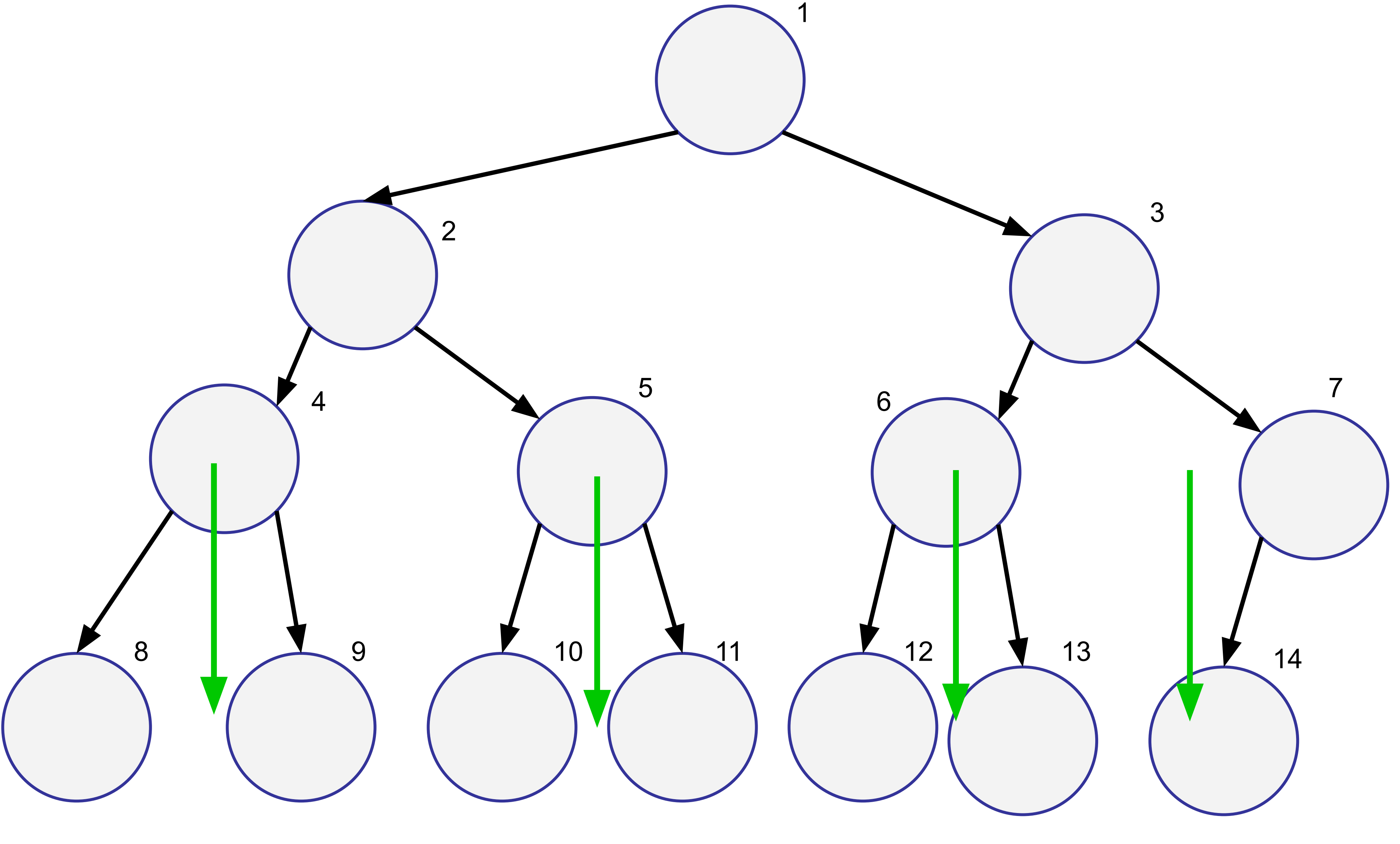


Binary Heaps:

Why is this O(n)? Isn't it O(n log n)?

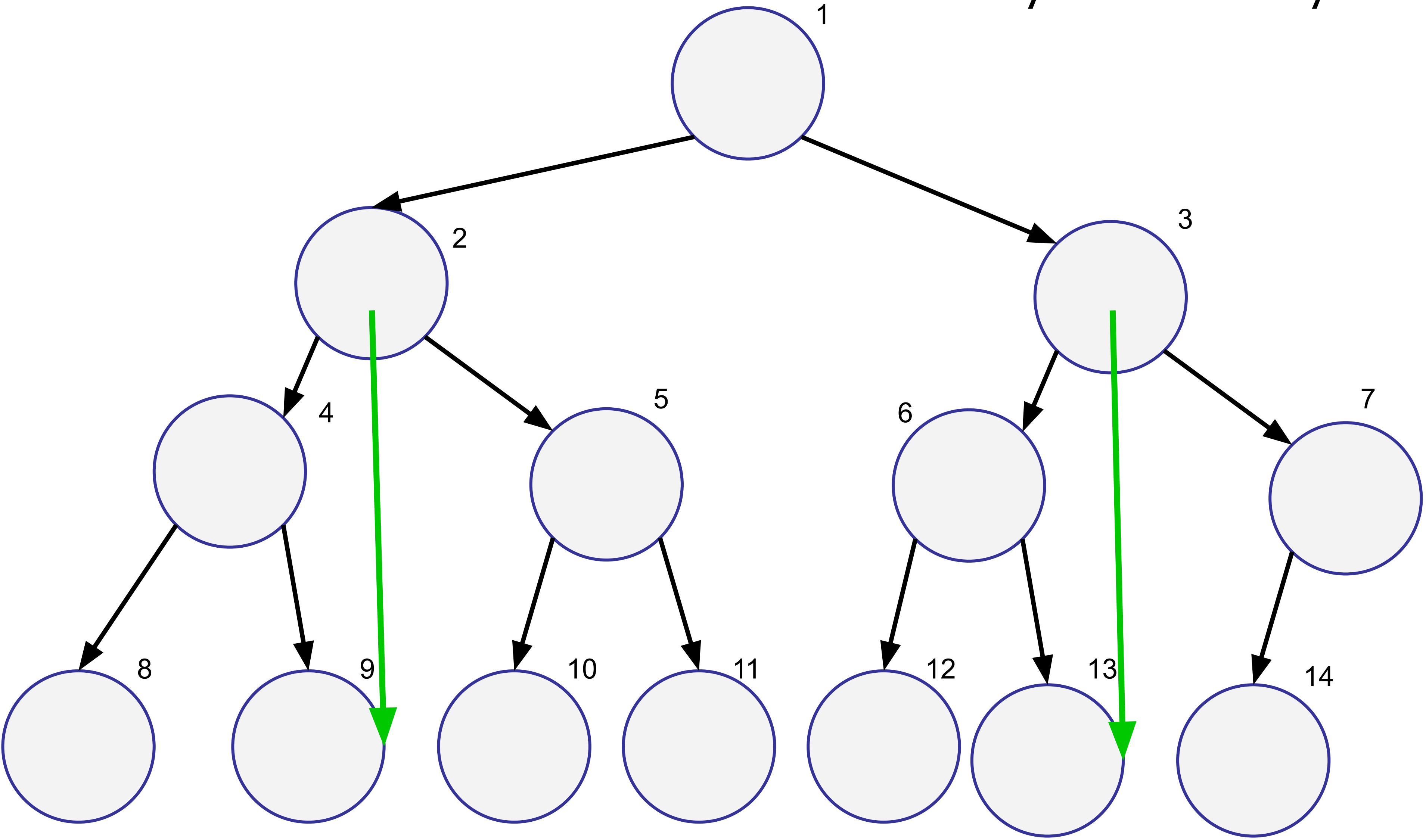
Binary Heaps:

The lowest nodes traverse 1 level



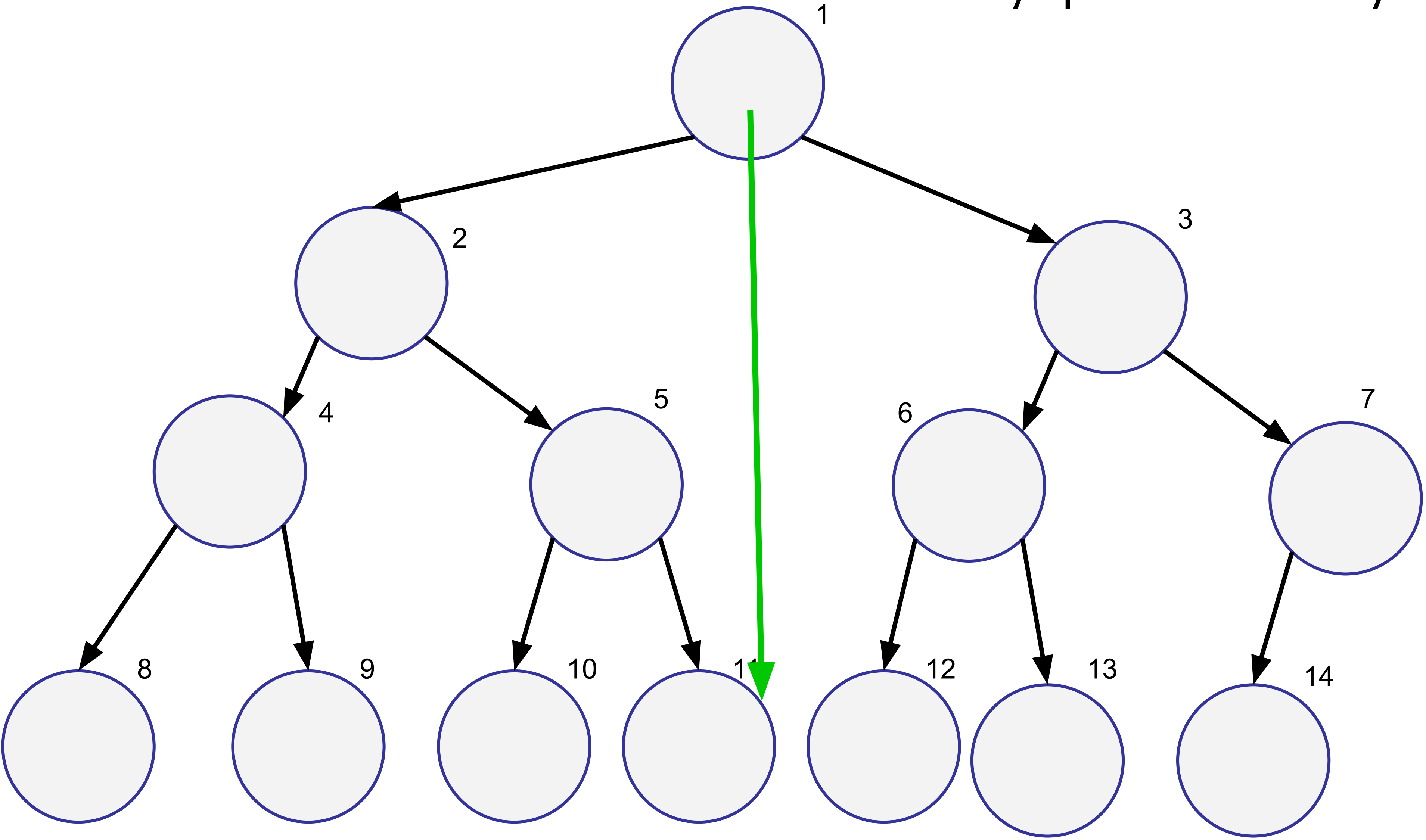
Binary Heaps:

The next nodes traverse 2 levels but only half as many



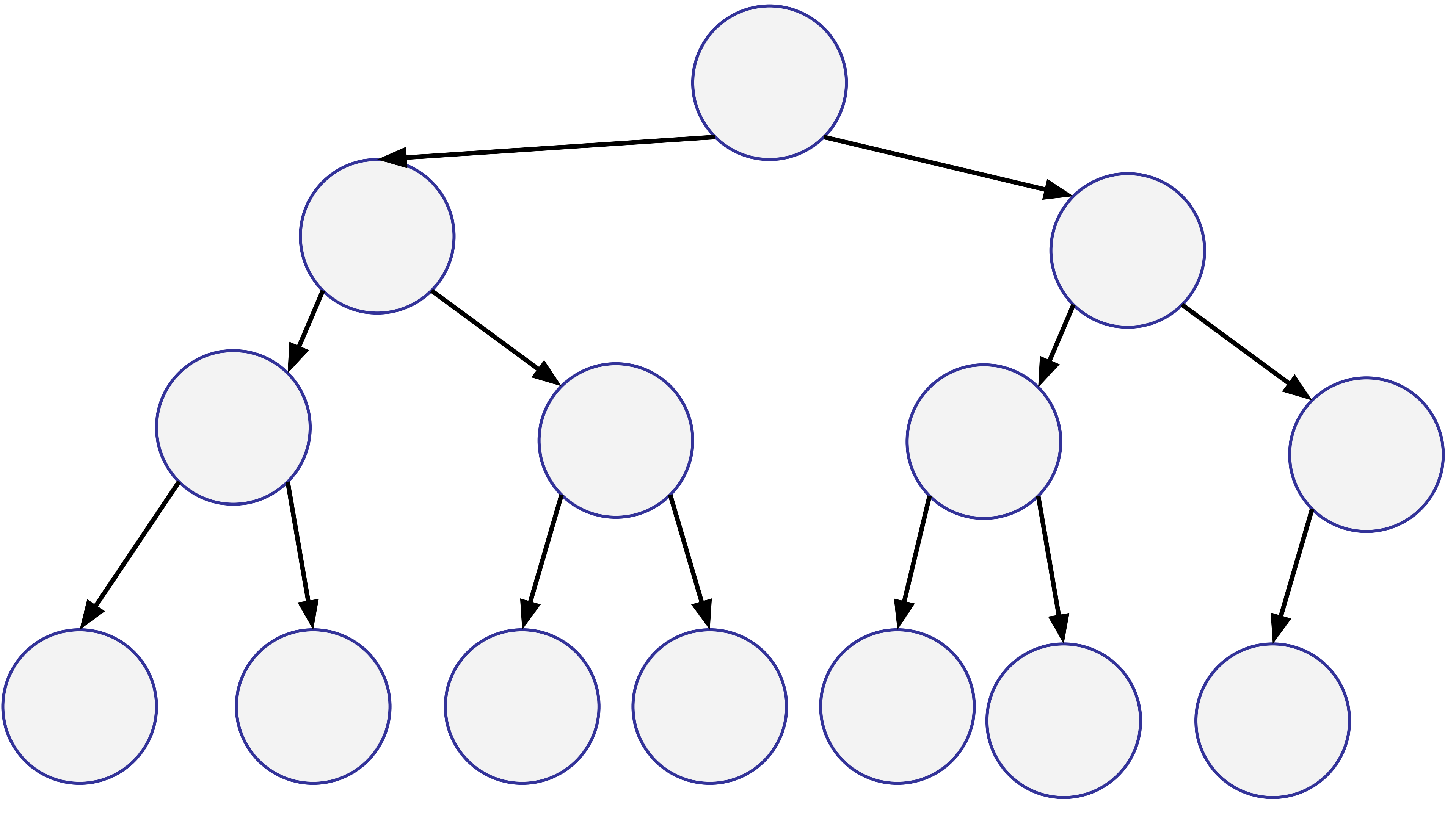
Binary Heaps:

The next nodes traverse 3 levels but only quarter as many



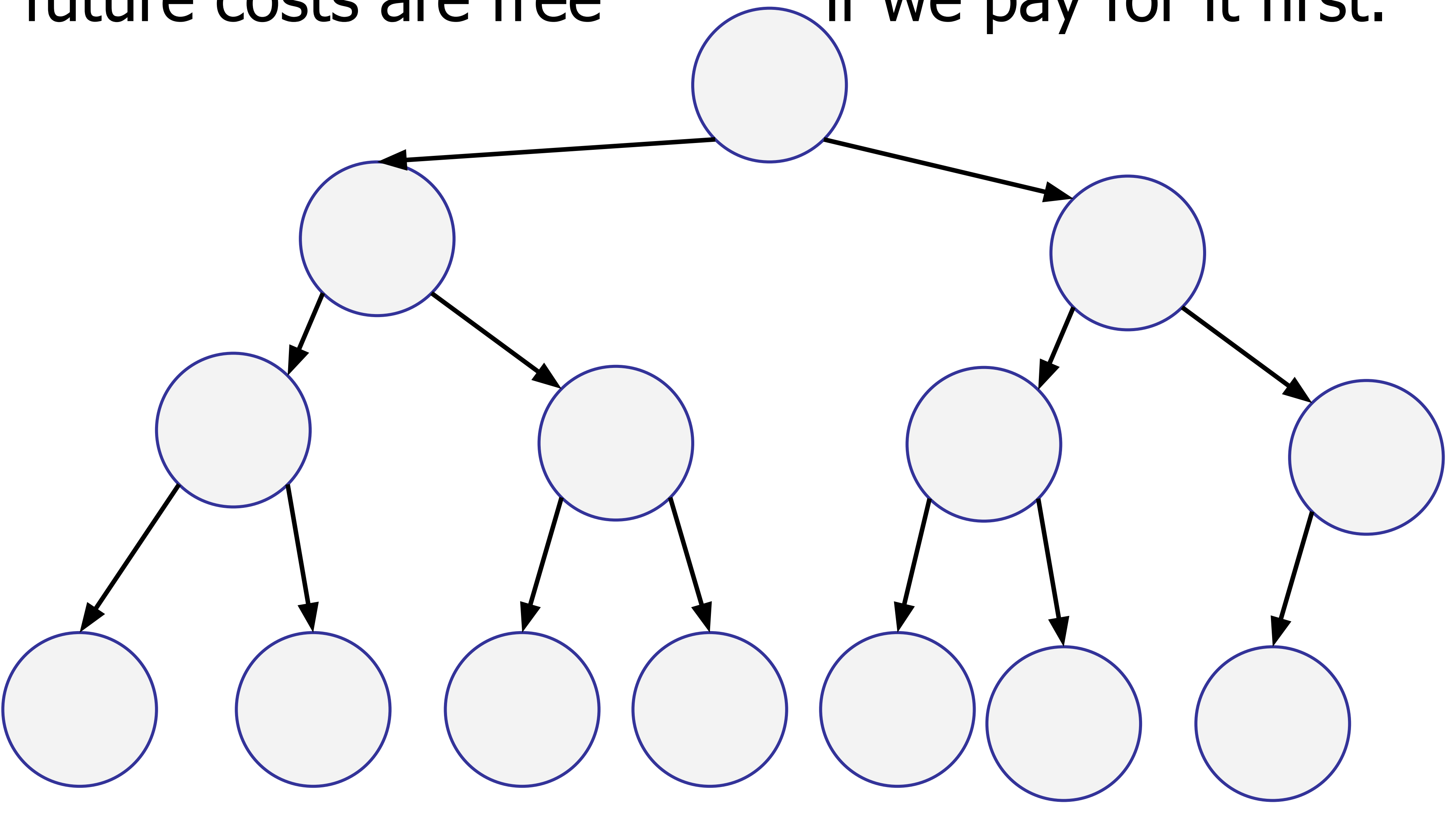
Binary Heaps:

Proof: Via amortised analysis



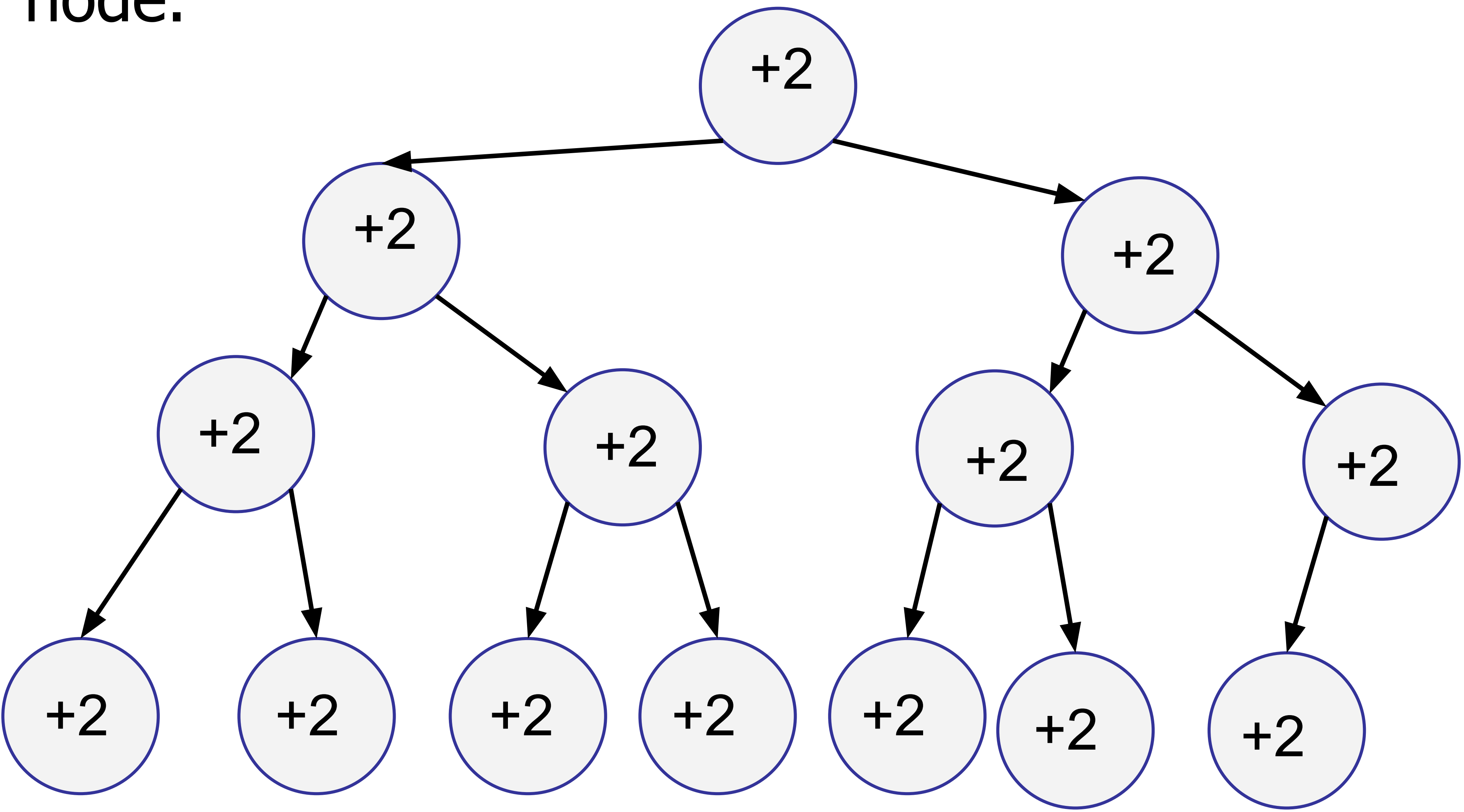
Binary Heaps:

Recall: Amortised analysis helps us pretend future costs are free if we pay for it first.



Binary Heaps:

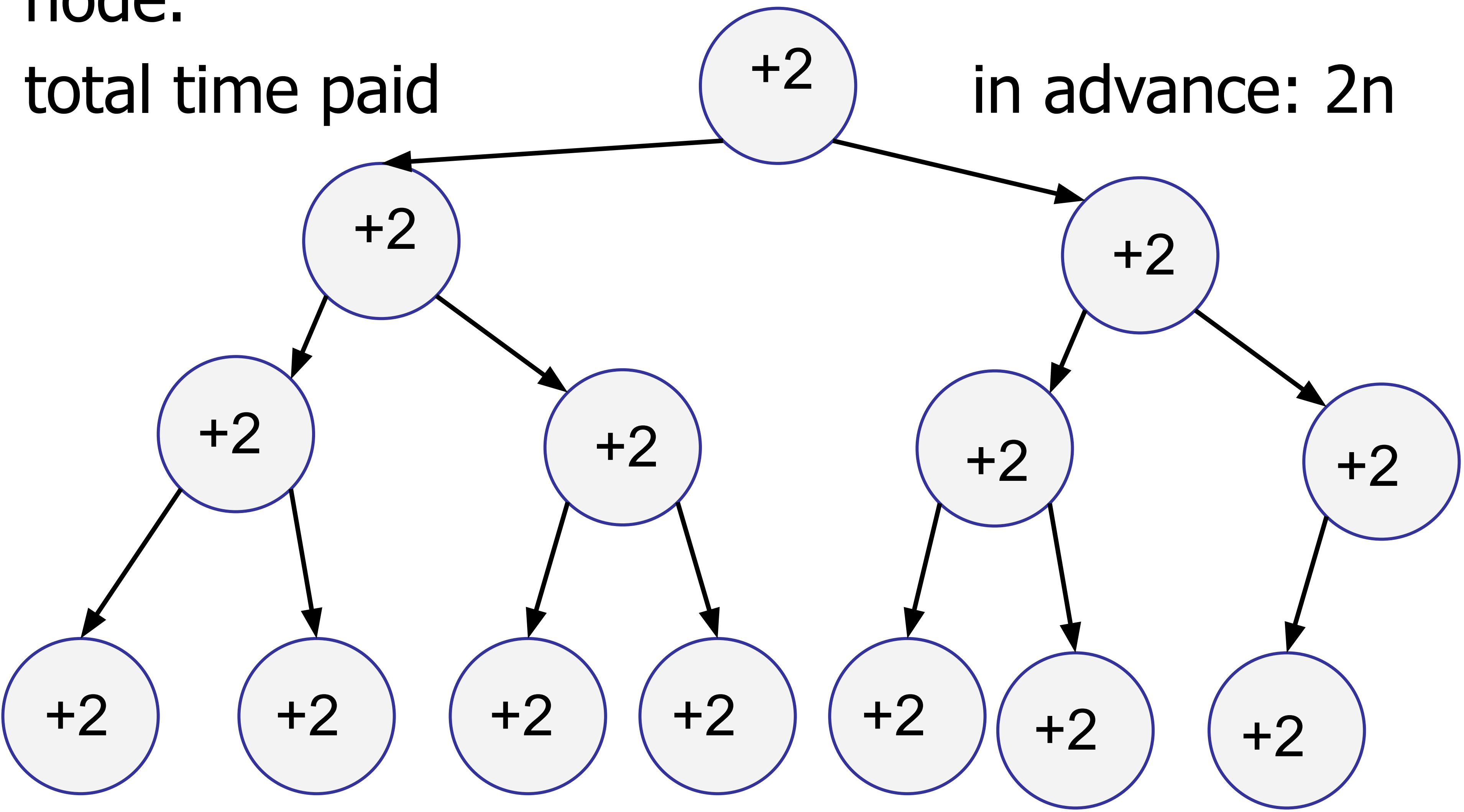
Before heapify, for n nodes, add +2 to each node.



Binary Heaps:

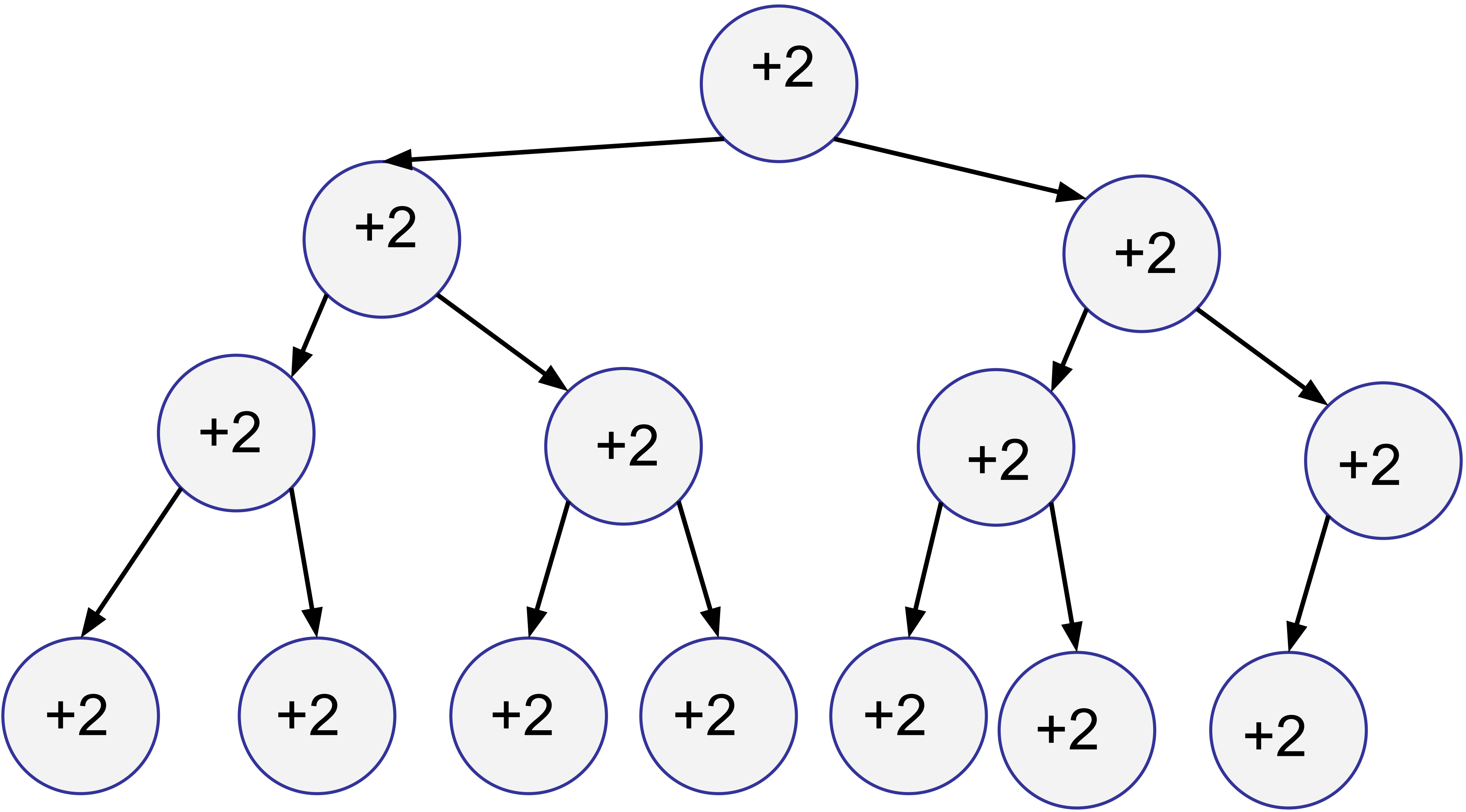
Before heapify, for n nodes, add +2 to each node.

total time paid in advance: 2n



Binary Heaps:

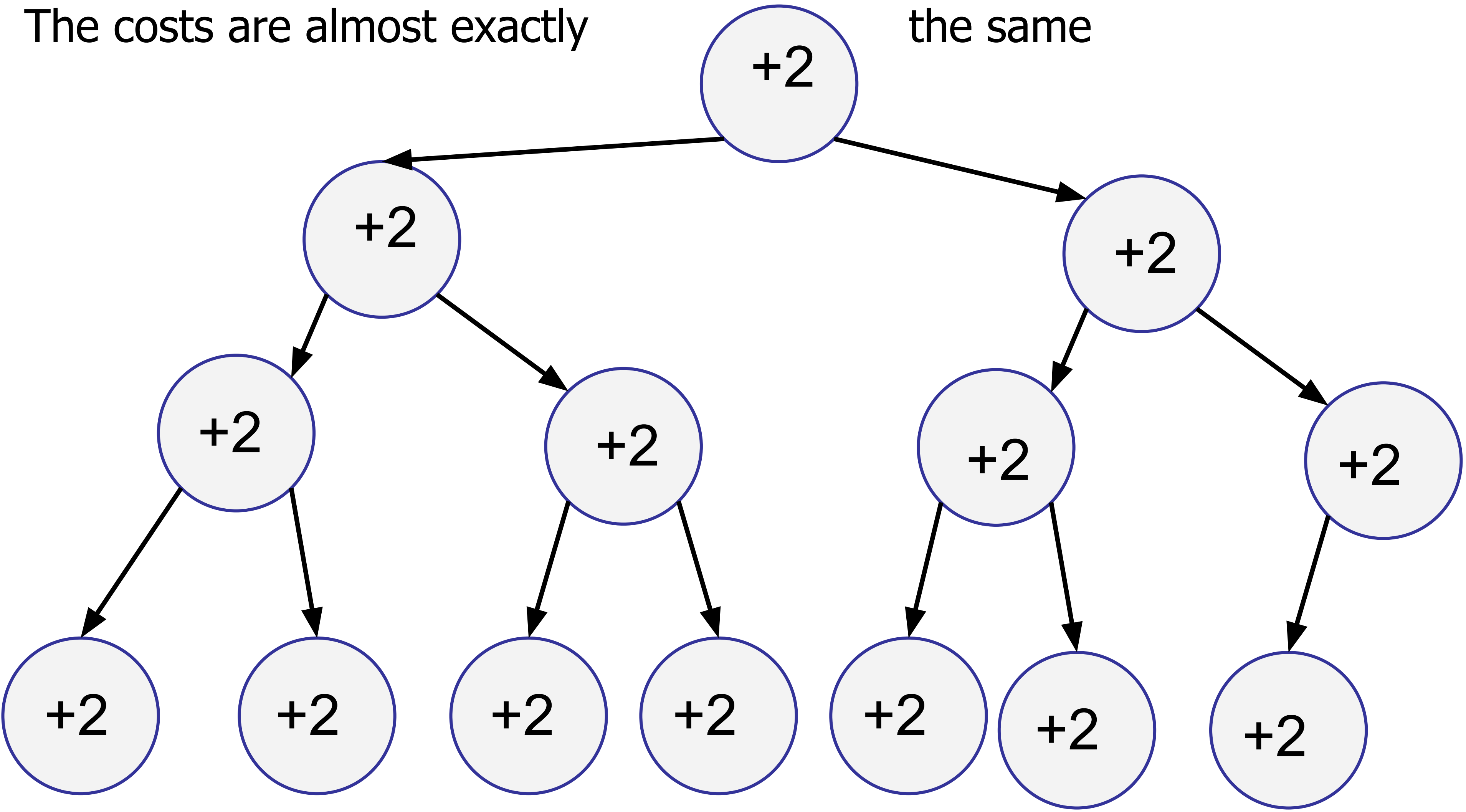
Claim: down all bubbleDown operations are free.



Binary Heaps:

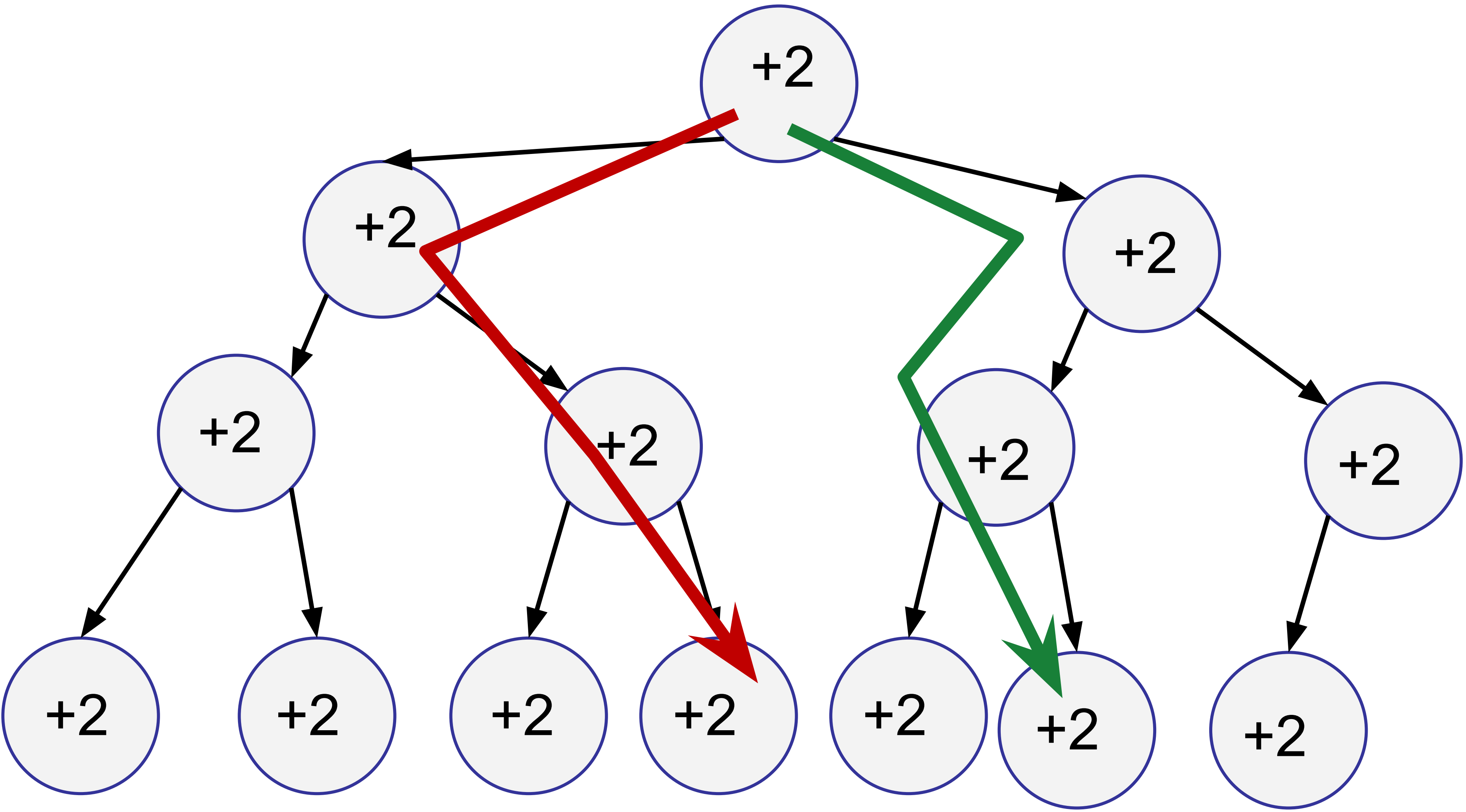
Claim 2: As we traverse the heap, regardless of the path, pretend the path was always “left, right, right, right…”

The costs are almost exactly the same



Binary Heaps:

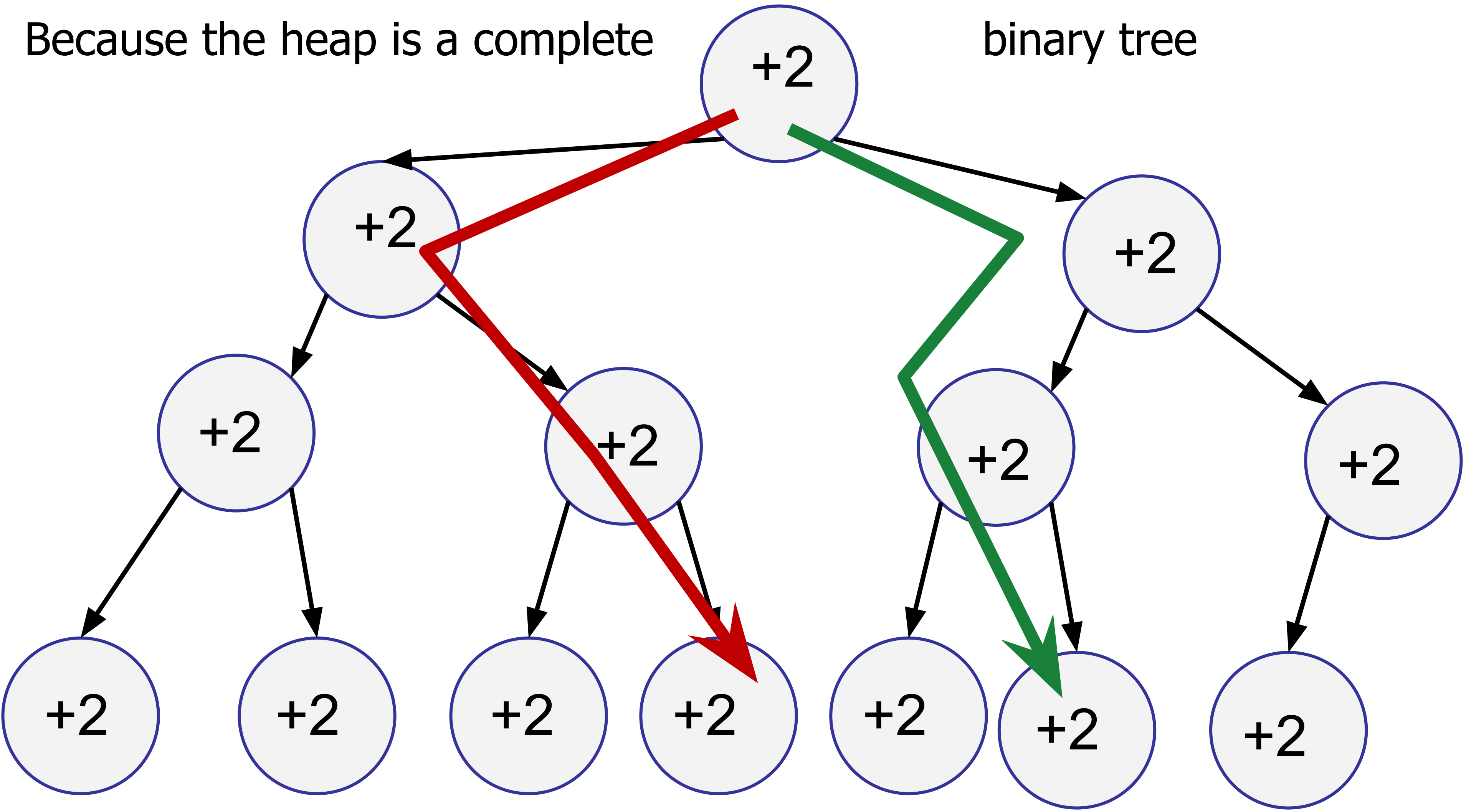
E.g. if bubble down would have done the green path, it costs the same as if it did the red path since only the height matters.



Binary Heaps:

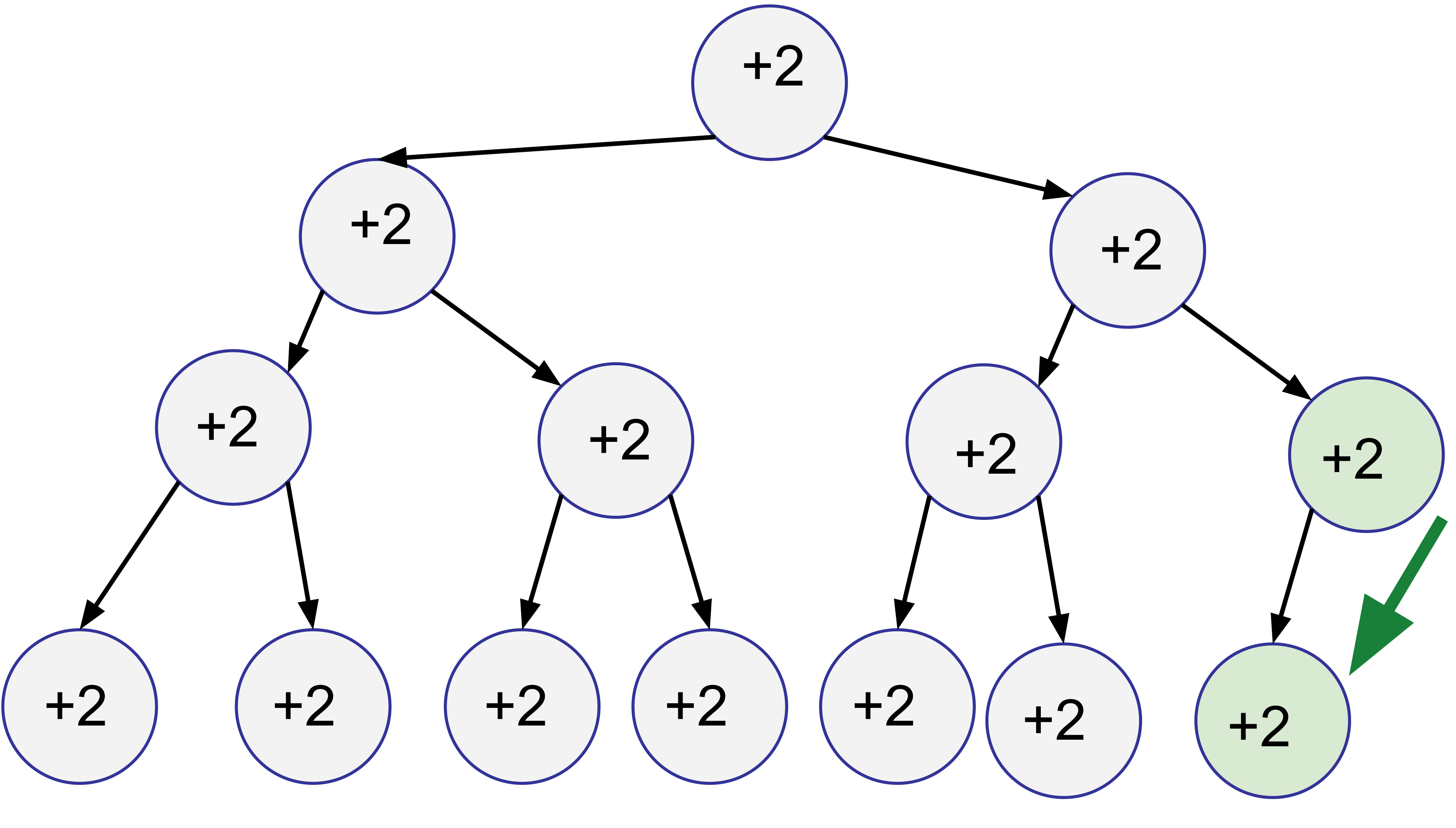
E.g. if bubble down would have done the green path, it costs the same as if it did the red path since only the height matters.

Because the heap is a complete binary tree



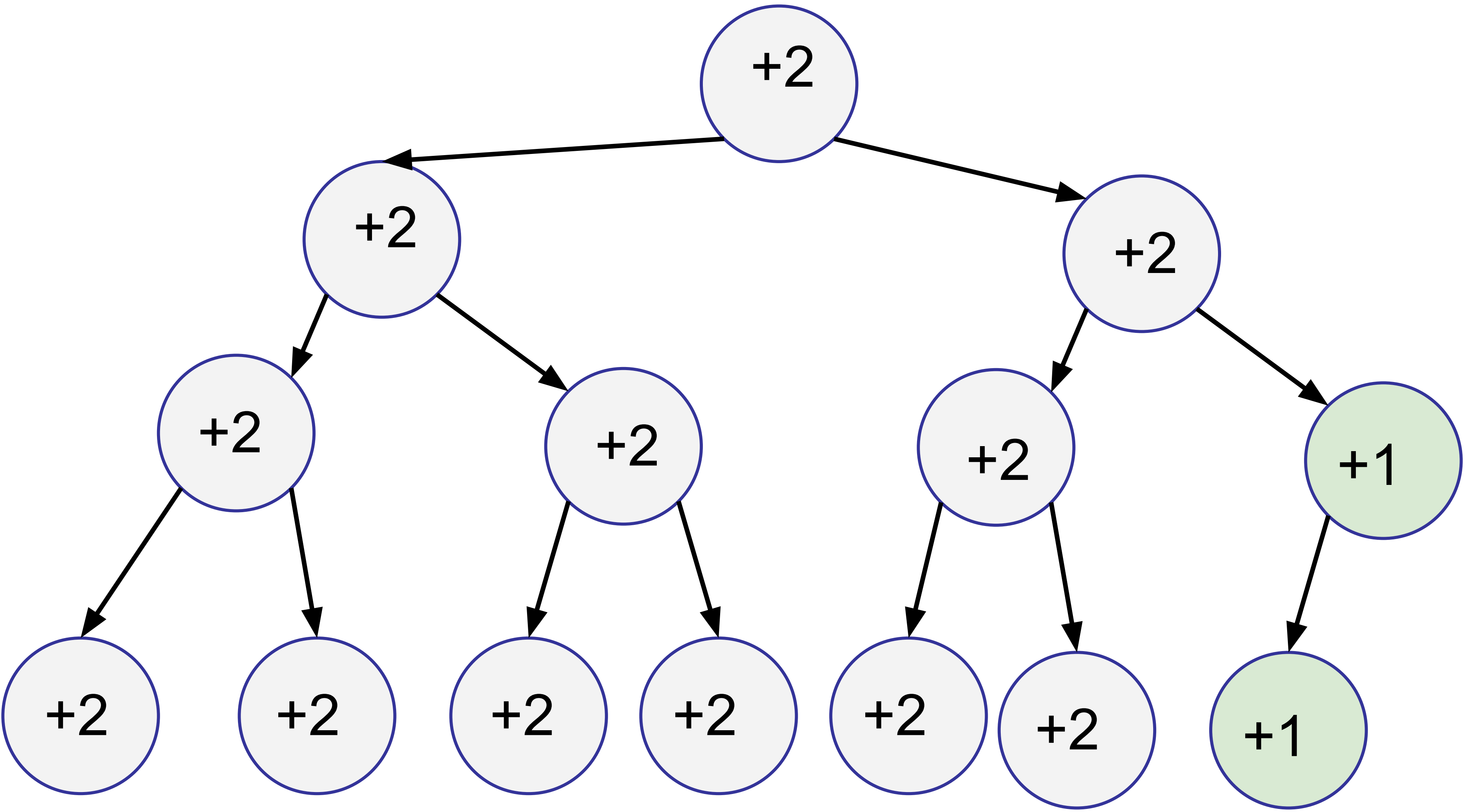
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



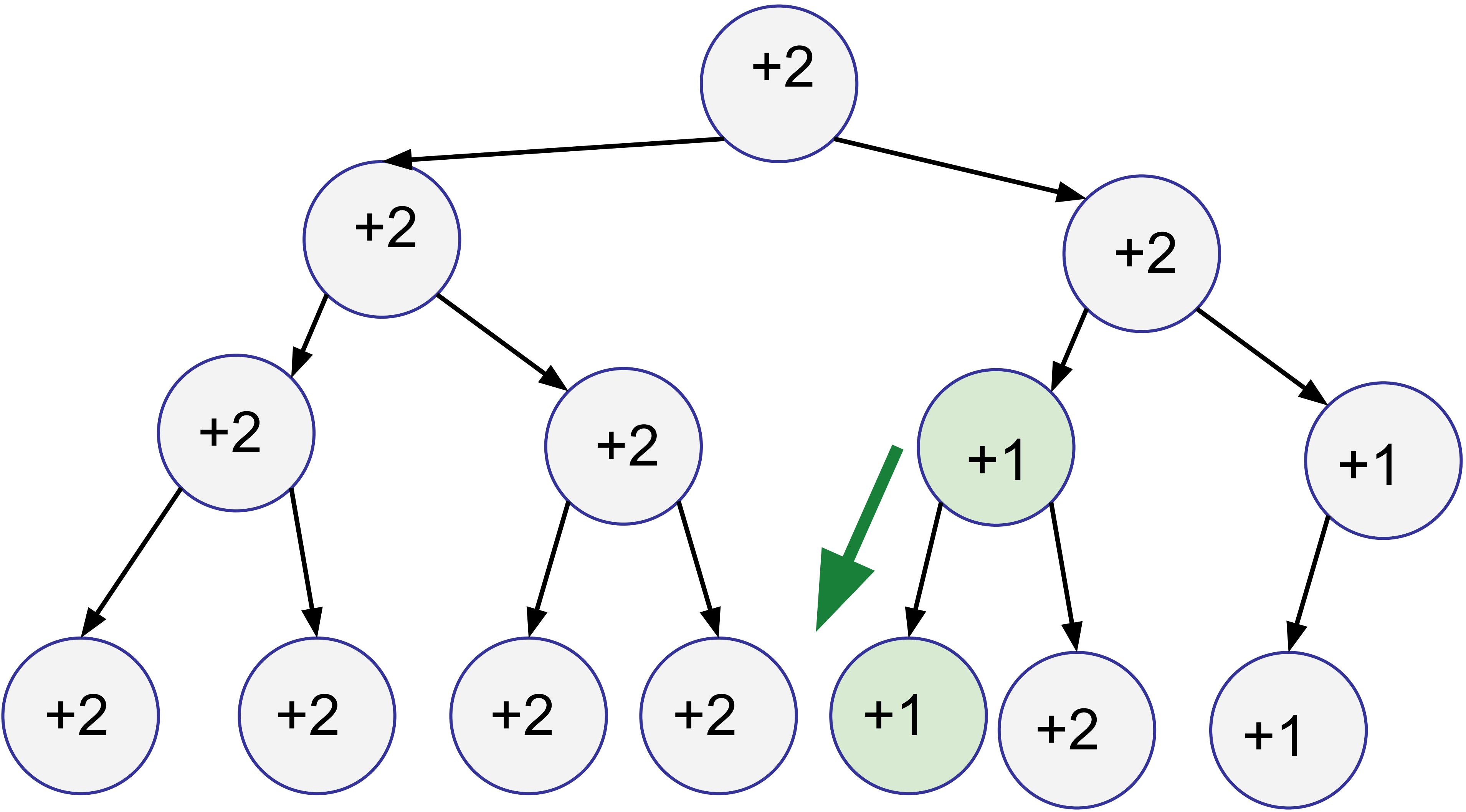
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



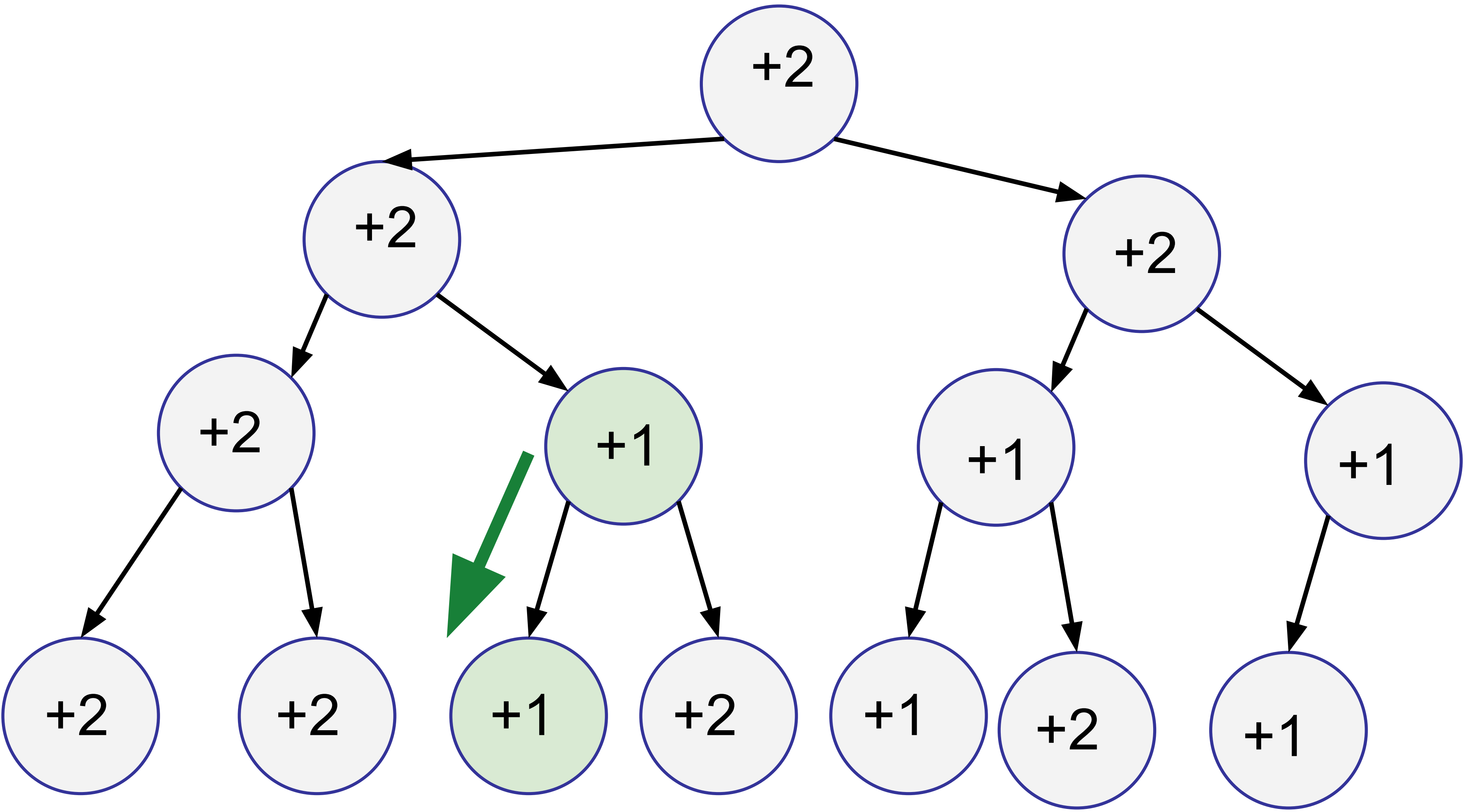
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



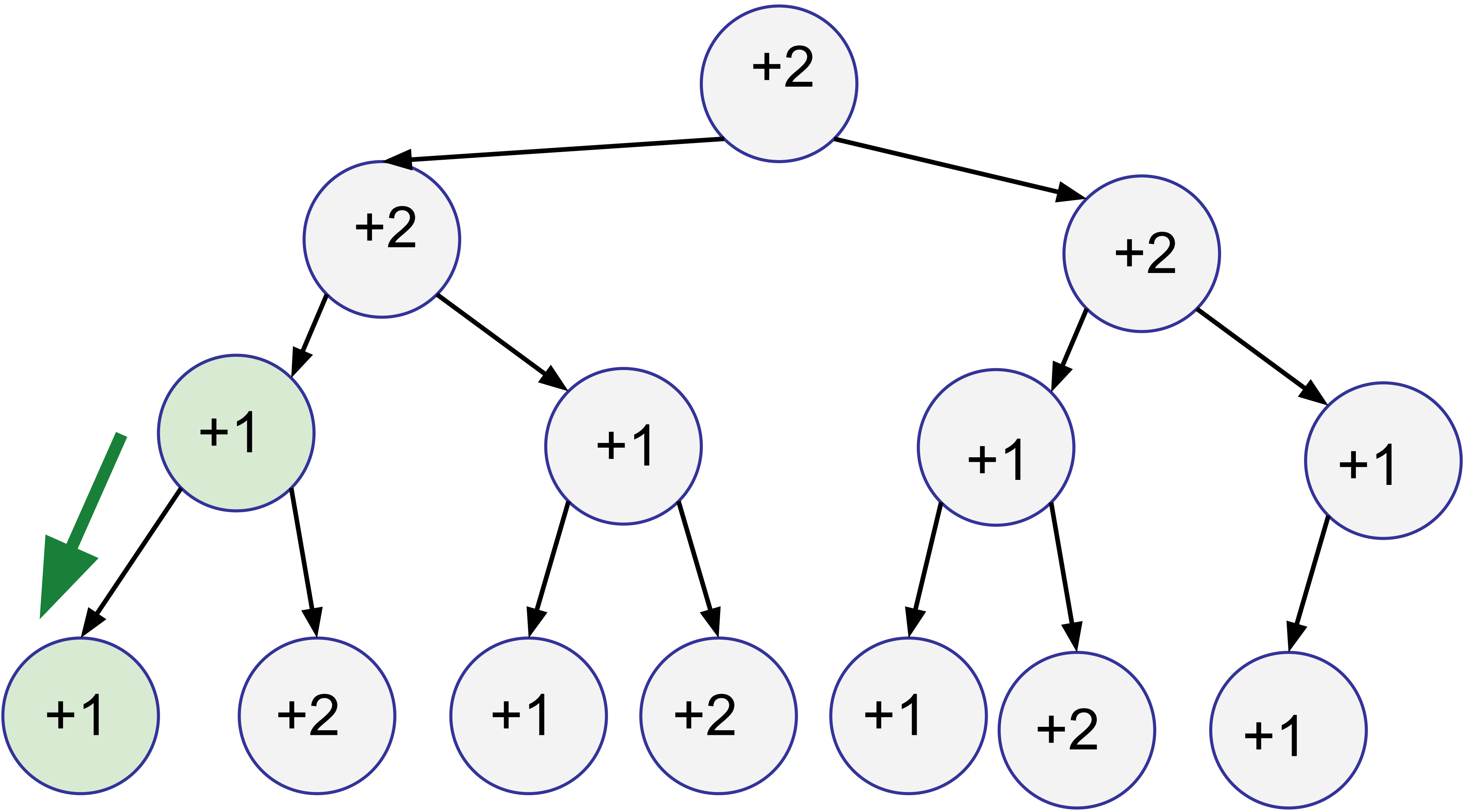
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



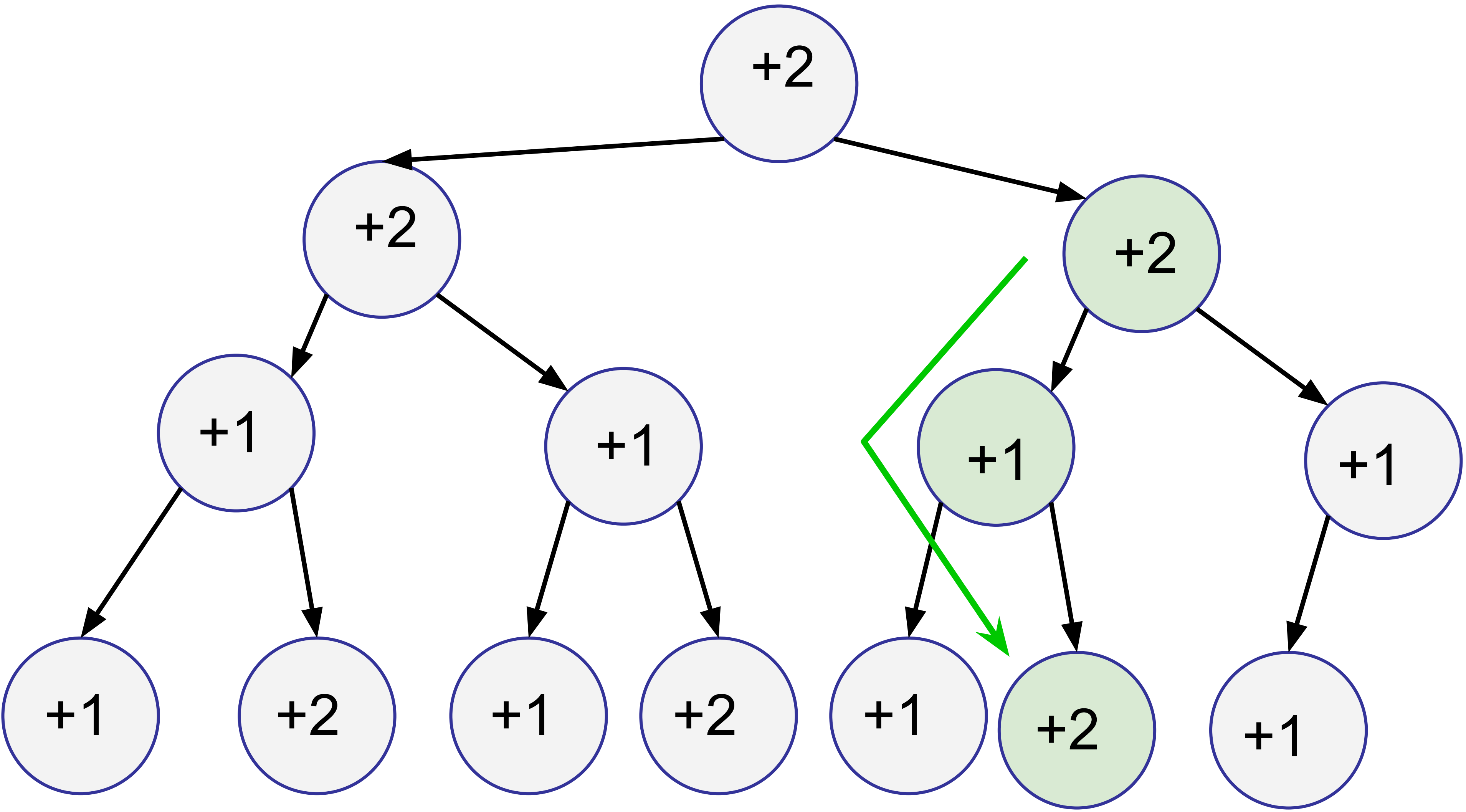
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



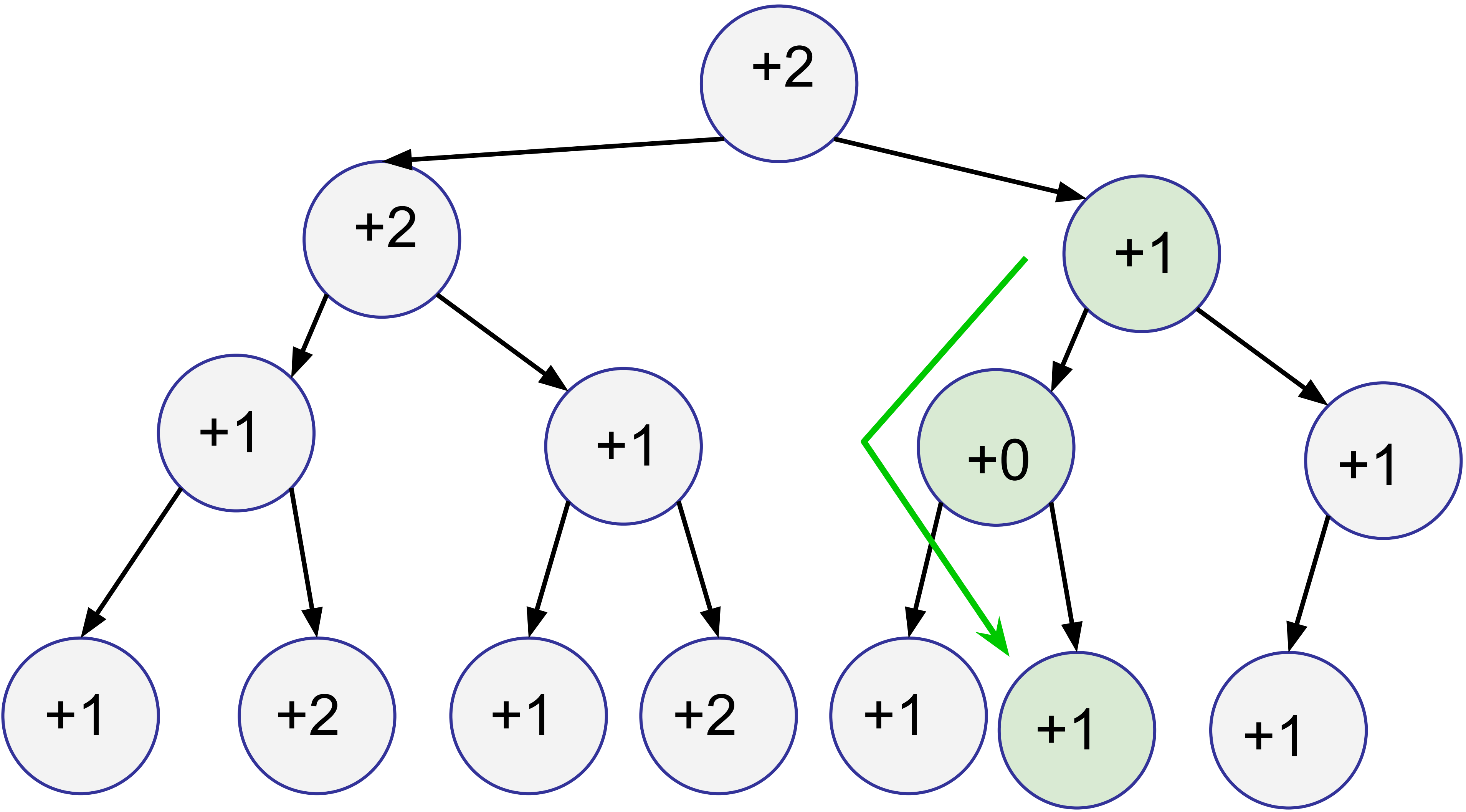
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



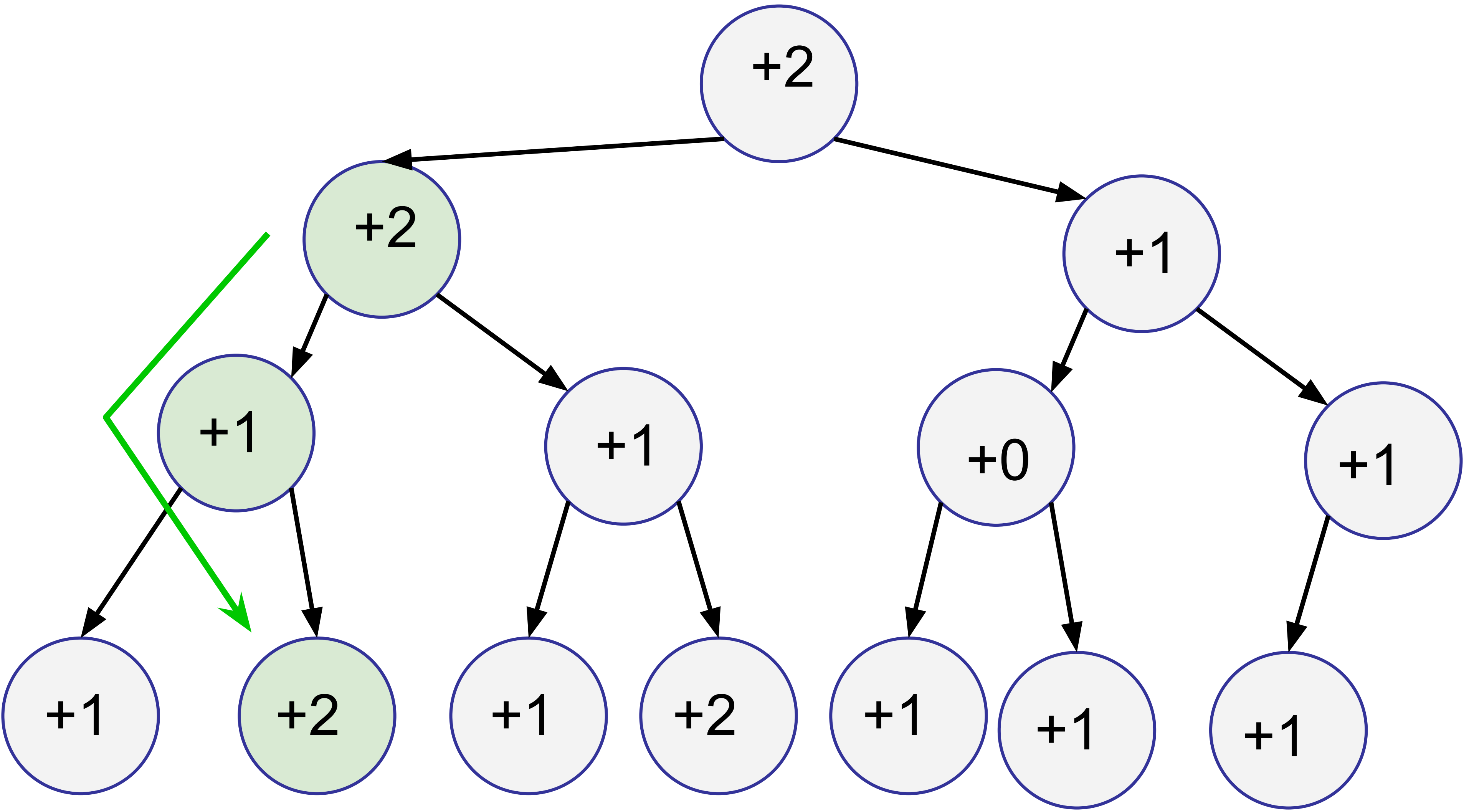
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



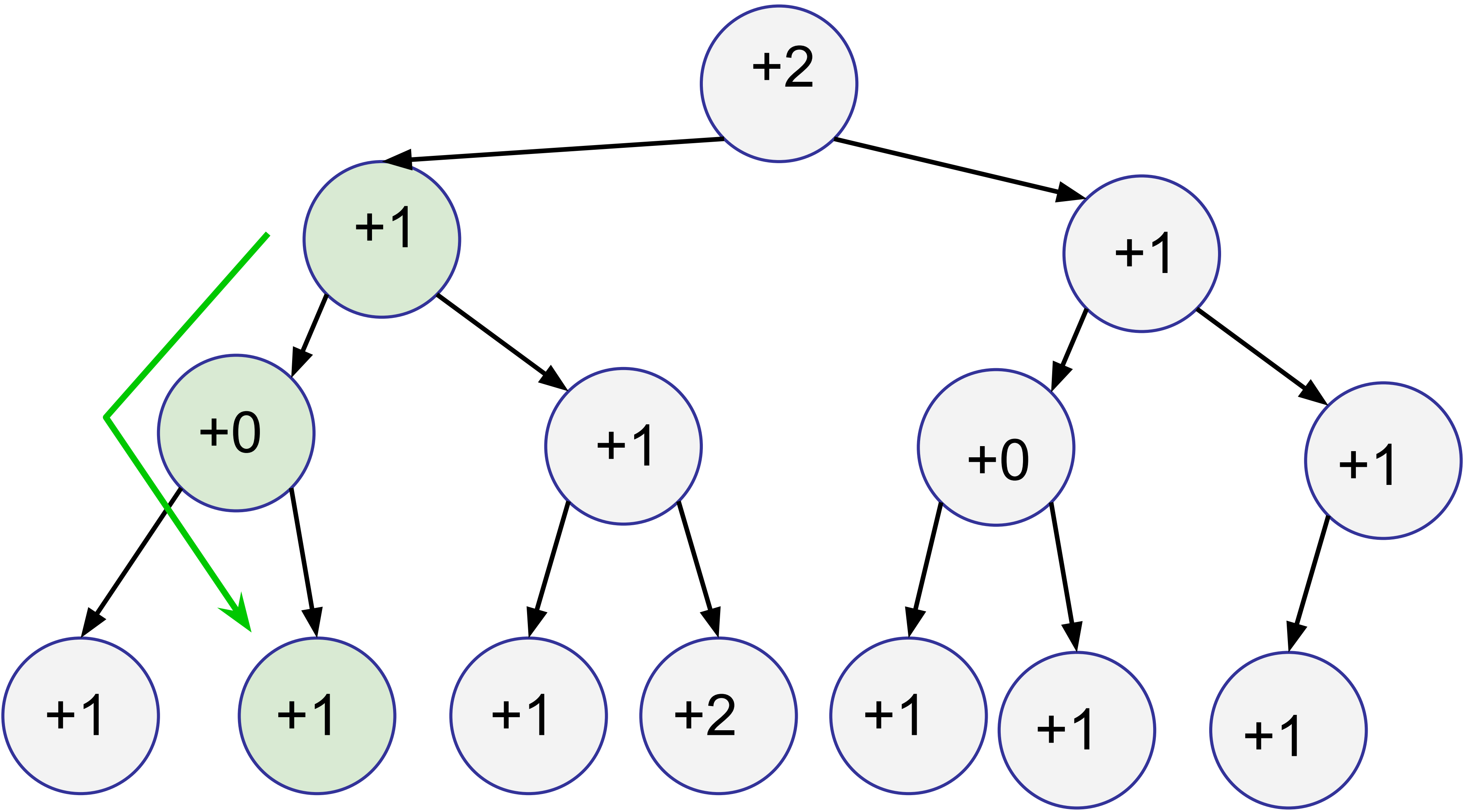
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



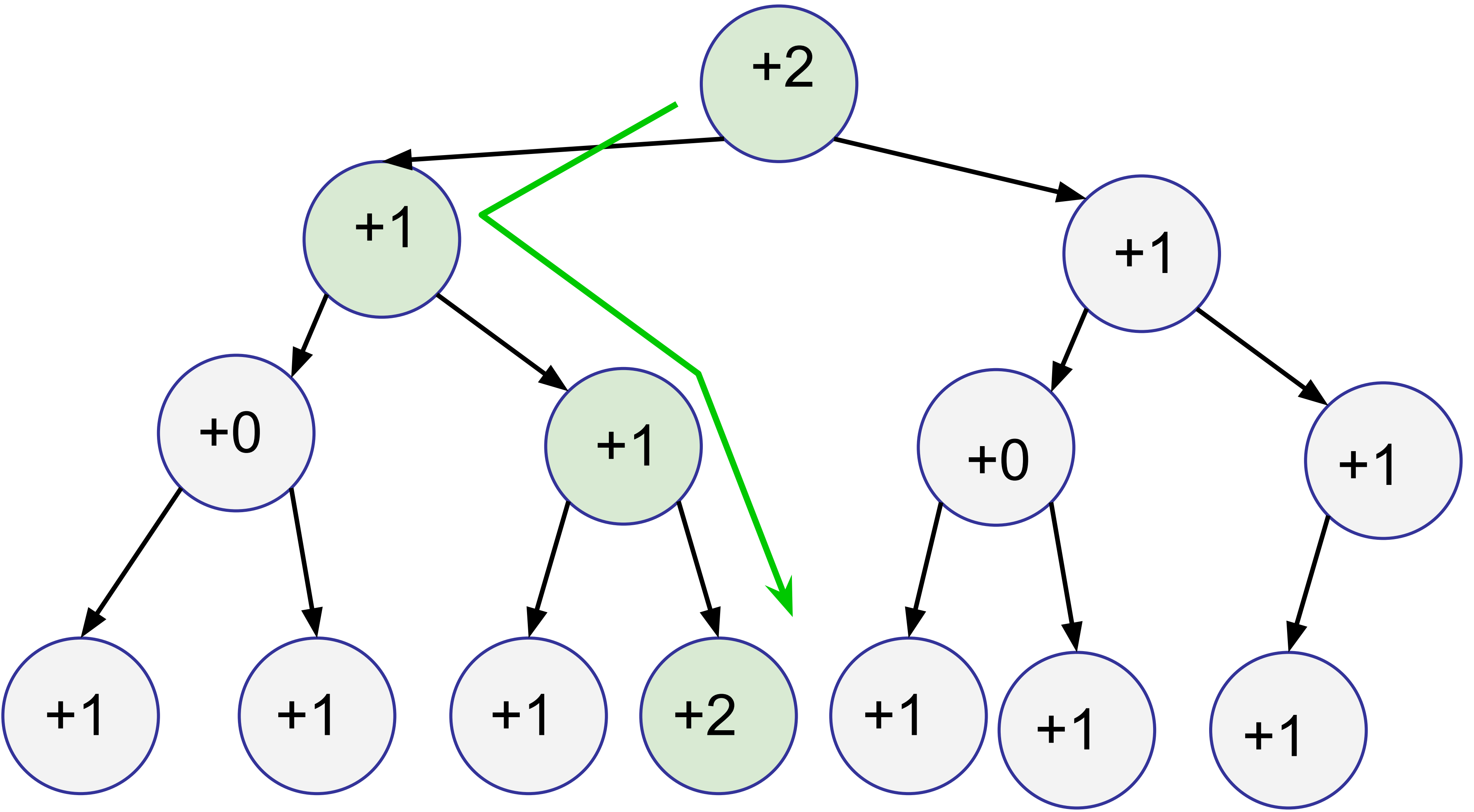
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



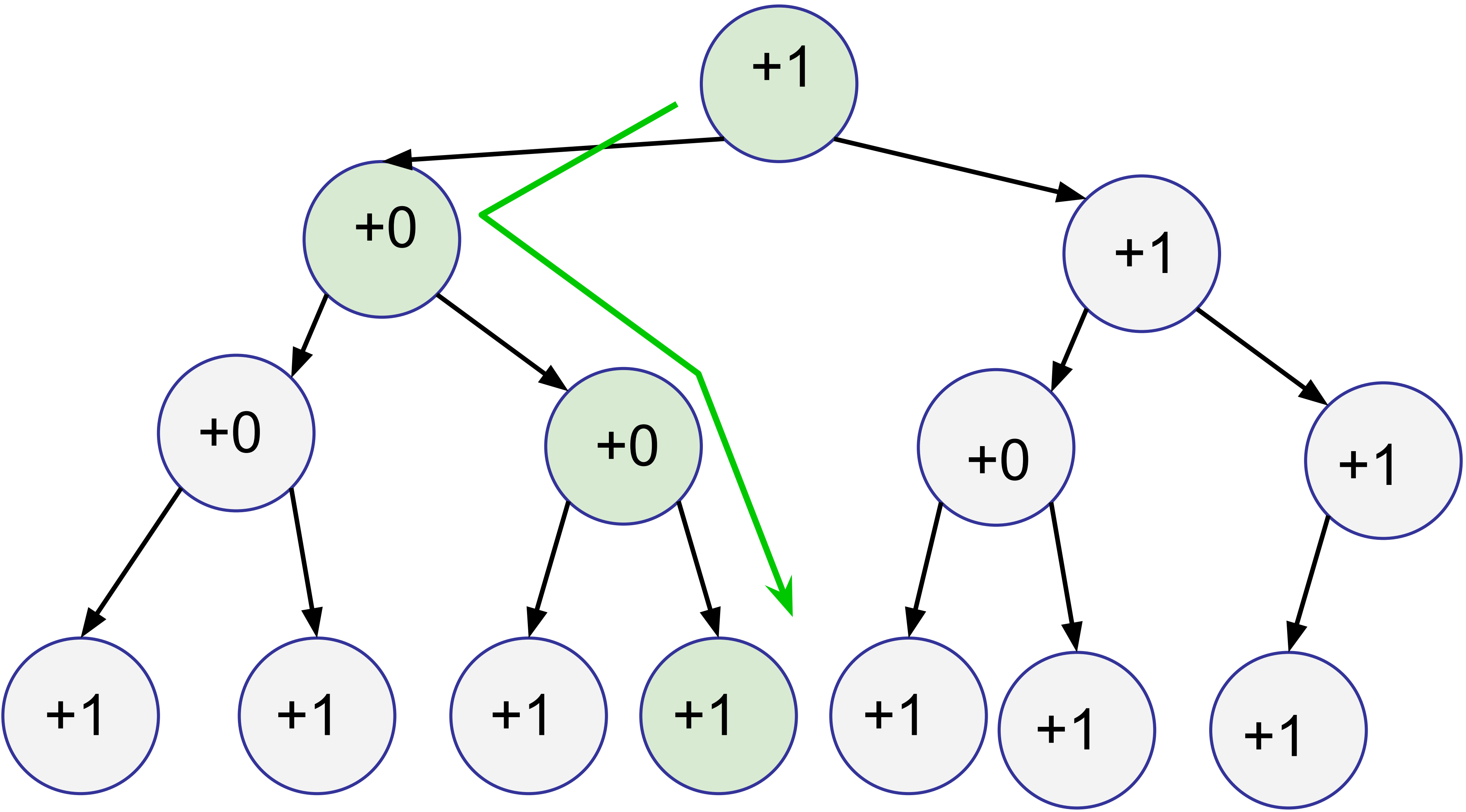
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



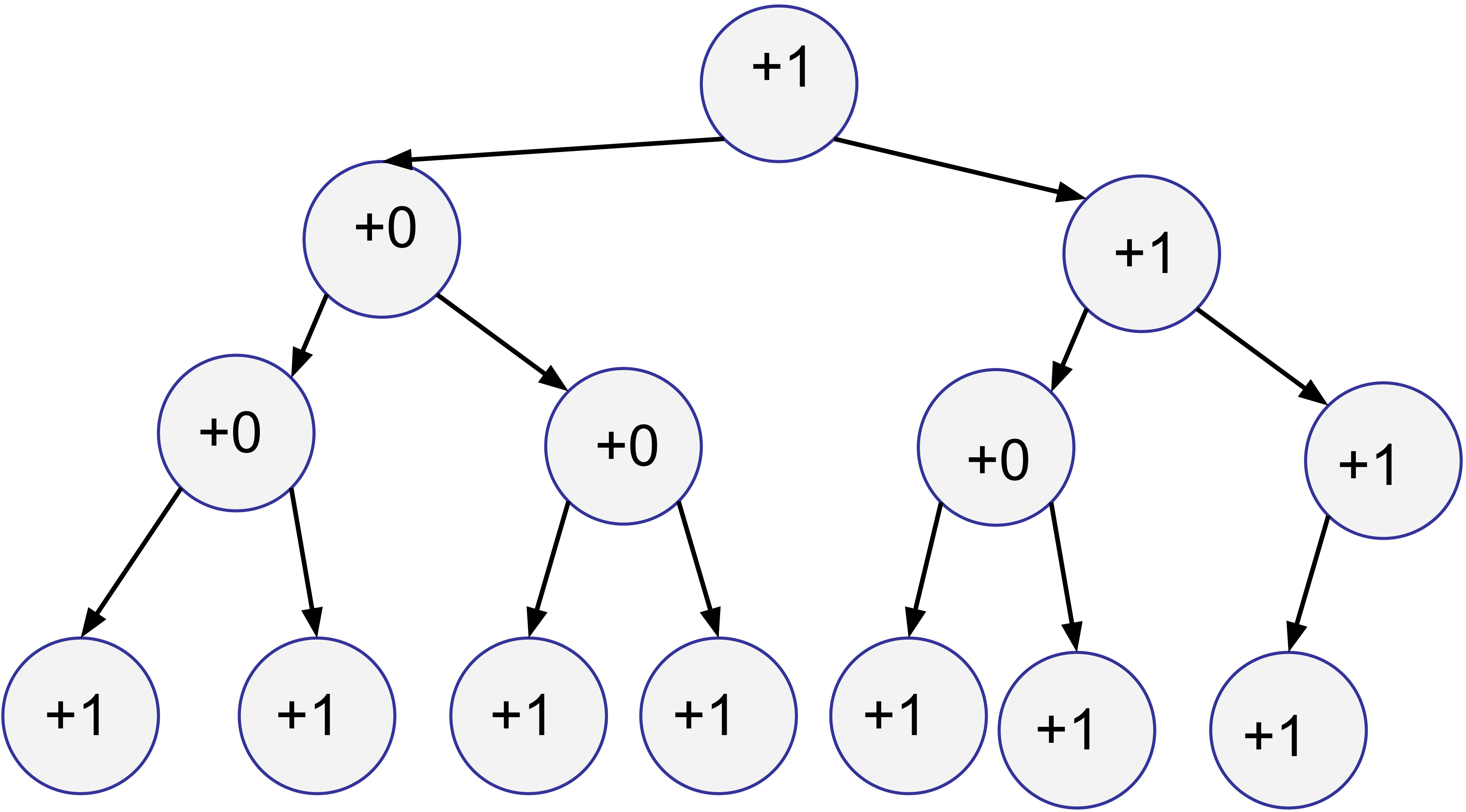
Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



Binary Heaps:

Claim: each bubble down will be paid for based on the time units paid in advance at each node.



Binary Heaps:

Claim: If we traverse this way, we will only ever visit nodes that have at least 1 or 2 units of time for us, never 0.

Proof: **Left as a challenge. Post on coursemology!**

Binary Heaps:

Since we paid 2n units of time in advance to make all subsequent downHeaps free, heapify is free!

Binary Heaps:

Since we paid 2n units of time in advance to make all subsequent downHeaps free, heapify is free!

Let d(i) = cost of downheap starting from index i

total cost of heapify = d(n/2) + d(n/2 - 1) + d(n/2 - 2) + … d(1)

$$= (2n - 2n) + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1))$$

$$= 2n + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1) - 2n)$$

Binary Heaps:

Since we paid 2n units of time in advance to make all subsequent downHeaps free, heapify is free!

Let $d(i)$ = cost of downheap starting from index i

total cost of heapify $= d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1)$

$$= (2n - 2n) + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1))$$

$$= 2n + \boxed{(d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1)} - 2n)$$

each term is proportional to how many nodes the bubbleDown visited

Binary Heaps:

Since we paid 2n units of time in advance to make all subsequent downHeaps free, heapify is free!

Let d(i) = cost of downheap starting from index i

total cost of heapify = d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1)

$$= (2n - 2n) + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1))$$

$$= 2n + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + \ldots d(1) \boxed{- 2n)}$$

redistribute this -2n to make all the d(i) terms 0

Binary Heaps:

Since we paid 2n units of time in advance to make all subsequent downHeaps free, heapify is free!

Let d(i) = cost of downheap starting from index i

total cost of heapify = d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1)

= (2n - 2n) + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1))

= 2n + $\boxed{\text{(d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1) - 2n)}}$

As long as the sum total d(i) cost does not exceed 2n, this term is not positive

Binary Heaps:

Since we paid 2n units of time in advance to make all subsequent downHeaps free, heapify is free!

Let d(i) = cost of downheap starting from index i

total cost of heapify = d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1)

= (2n - 2n) + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1))

= 2n + (d(n/2) + d(n/2 - 1) + d(n/2 - 2) + ... d(1) - 2n)

<= 2n

Today: Heaps and PQs

Priority Queue ADT:

– new API!

Binary Heap:

- new data structure!

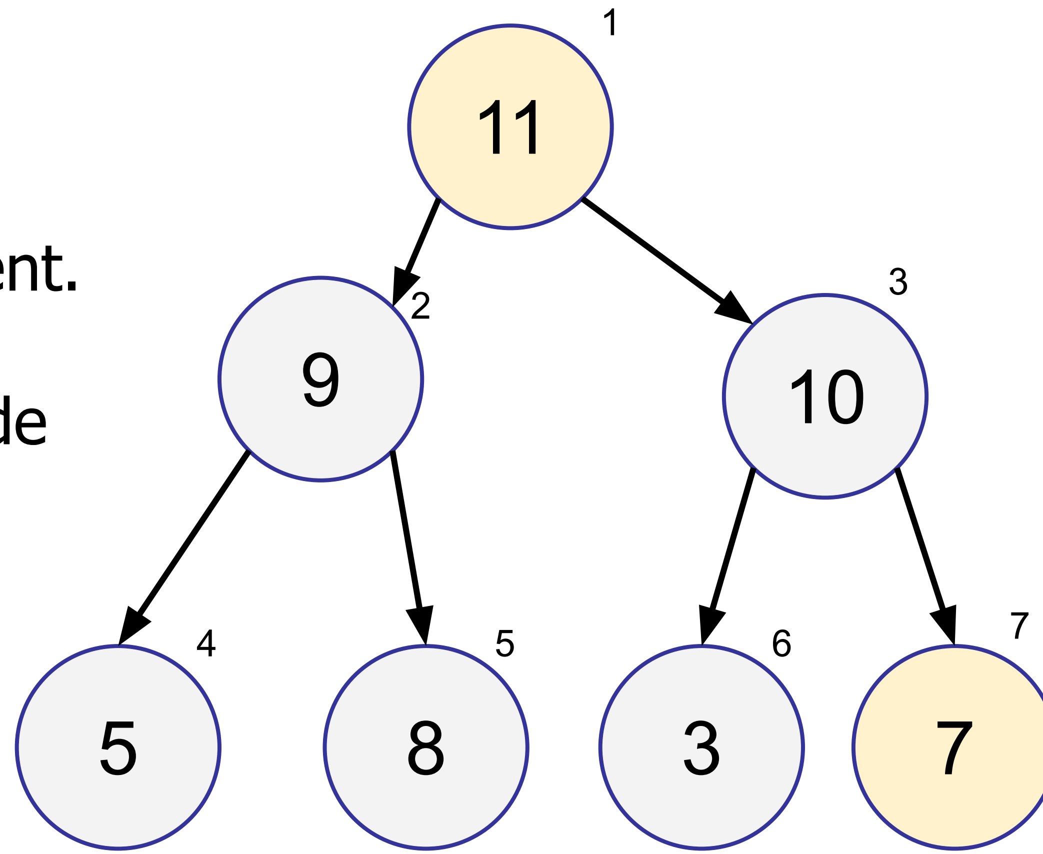
Heapsort:

– new cool sorting algorithm!

Heapsort:

Recall: extractMax

1. swap root with last element.
2. remove new last element
3. bubble down the root node

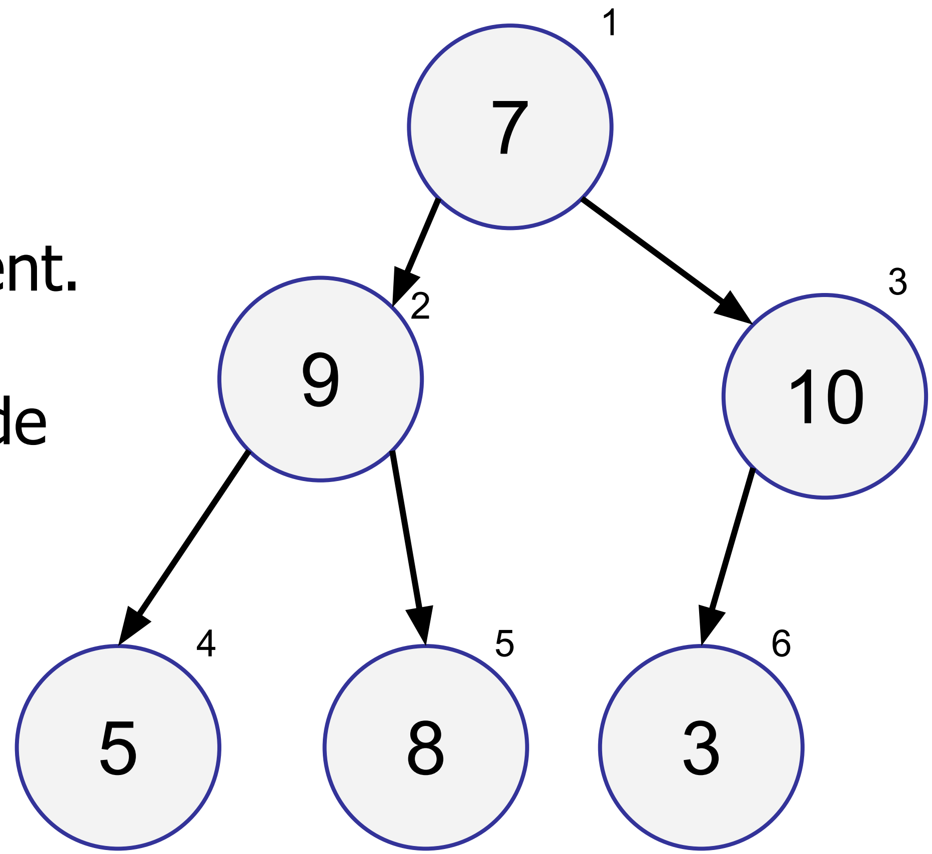


7	11	9	10	5	8	3	7
0	1	2	3	4	5	6	7

Heapsort:

Recall: extractMax

1. swap root with last element.
2. remove new last element
3. bubble down the root node

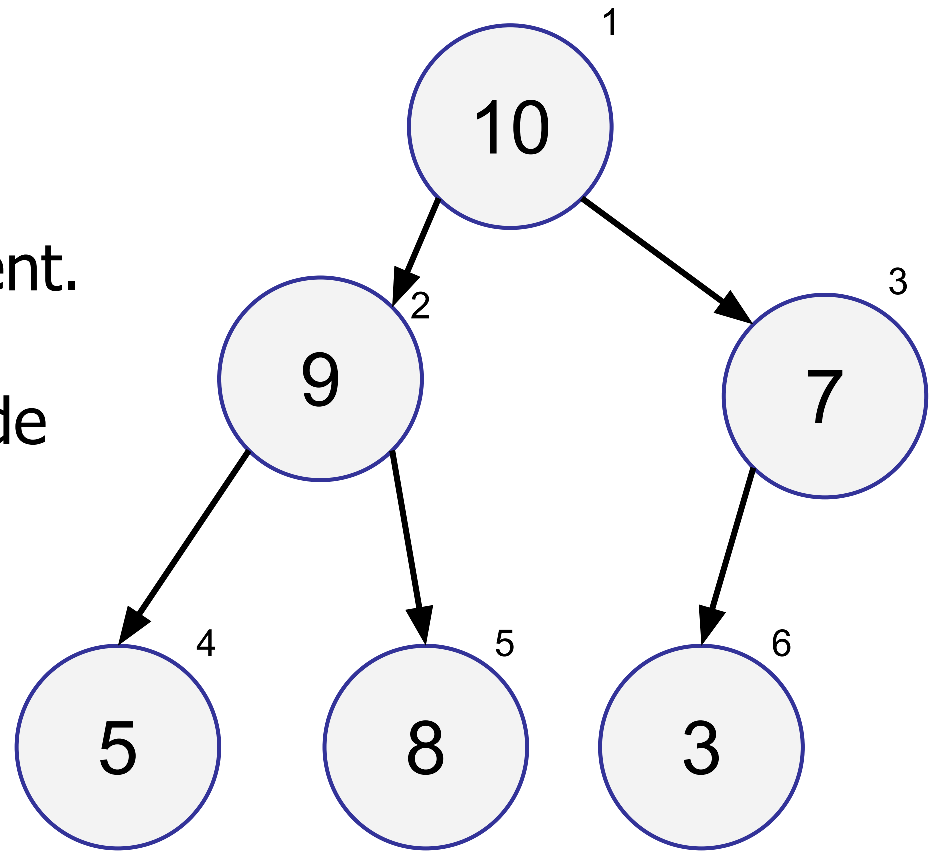


7	7	9	10	5	8	3	11
0	1	2	3	4	5	6	7

Heapsort:

Recall: extractMax

1. swap root with last element.
2. remove new last element
3. bubble down the root node

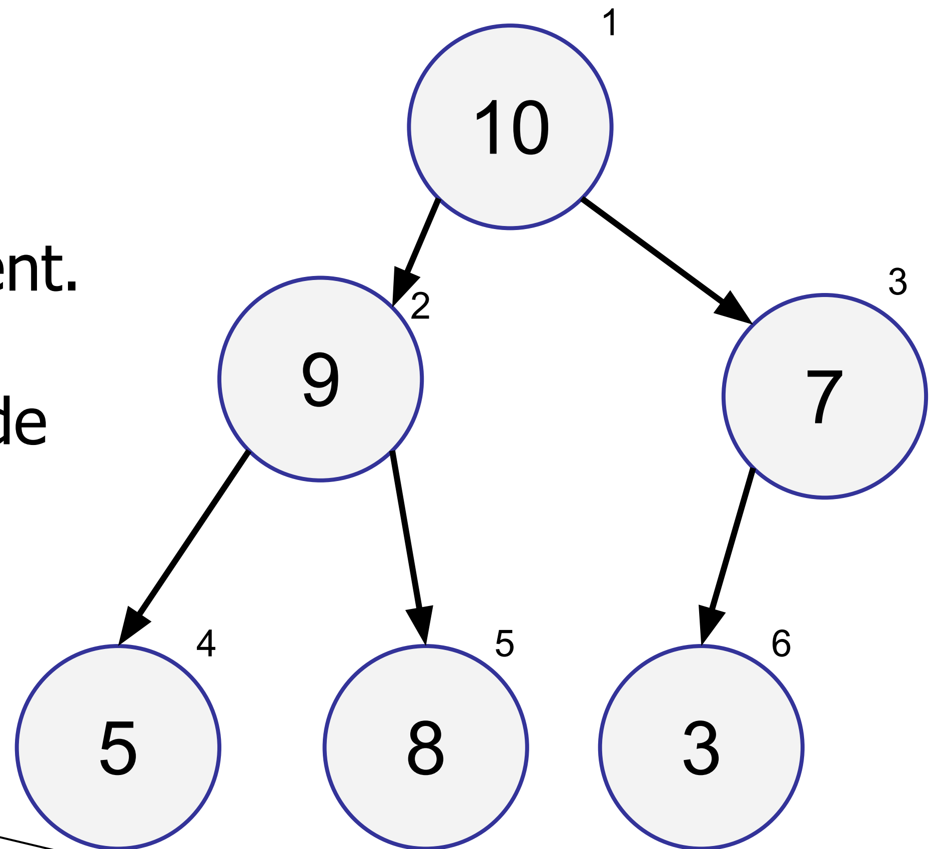


6	10	9	7	5	8	3	11
0	1	2	3	4	5	6	7

Heapsort:

Recall: extractMax

1. swap root with last element.
2. remove new last element
3. bubble down the root node



what happens if we
just kept it in the array?

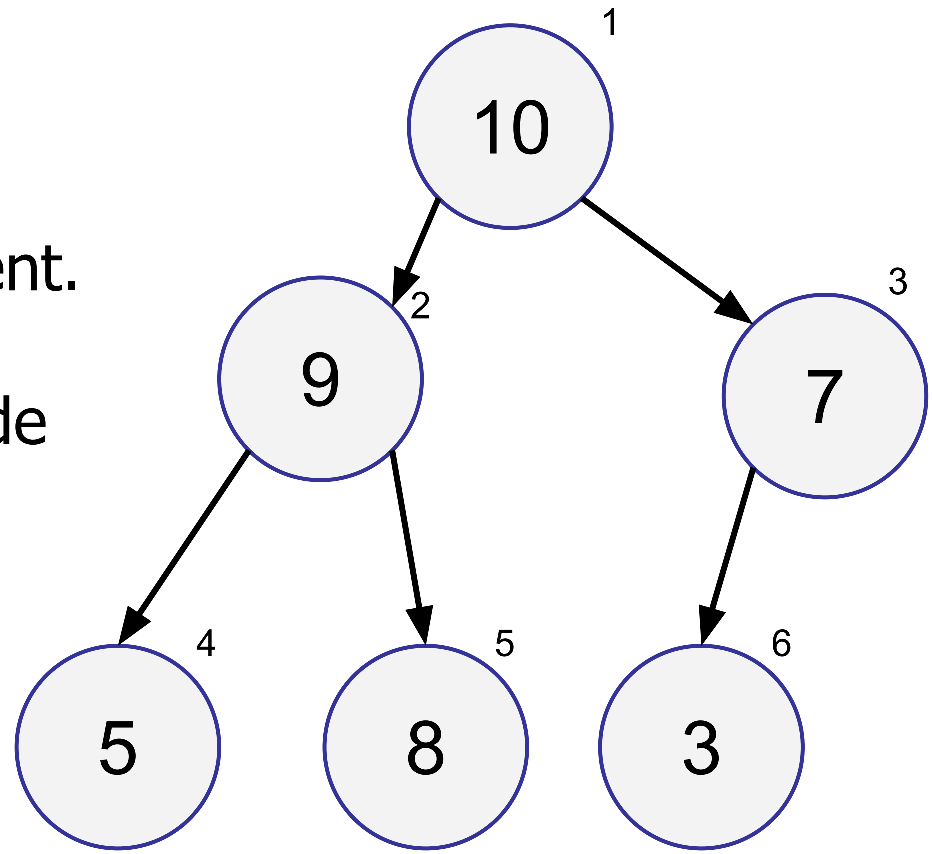
6	10	9	7	5	8	3	11
0	1	2	3	4	5	6	7

Heapsort:

Recall: extractMax

1. swap root with last element.
2. remove new last element
3. bubble down the root node

Where does the next max element go?



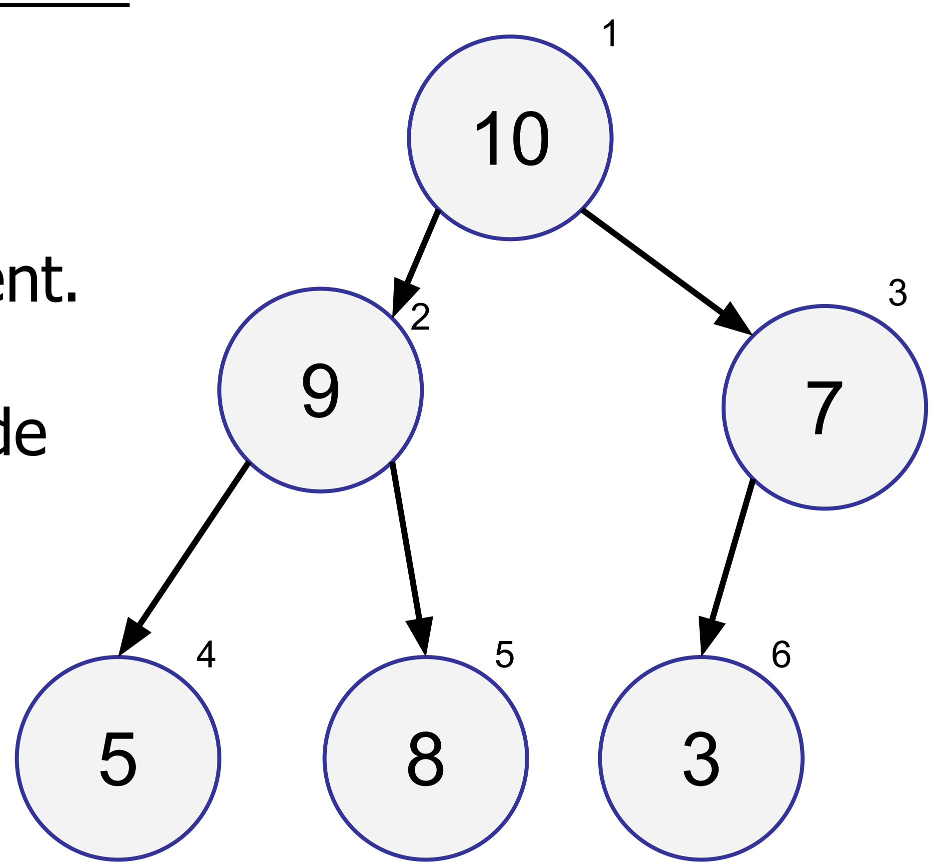
6	10	9	7	5	8	3	11
0	1	2	3	4	5	6	7

Heapsort:

Recall: extractMax

1. swap root with last element.
2. remove new last element
3. bubble down the root node

This is just a hyperefficient selection sort!

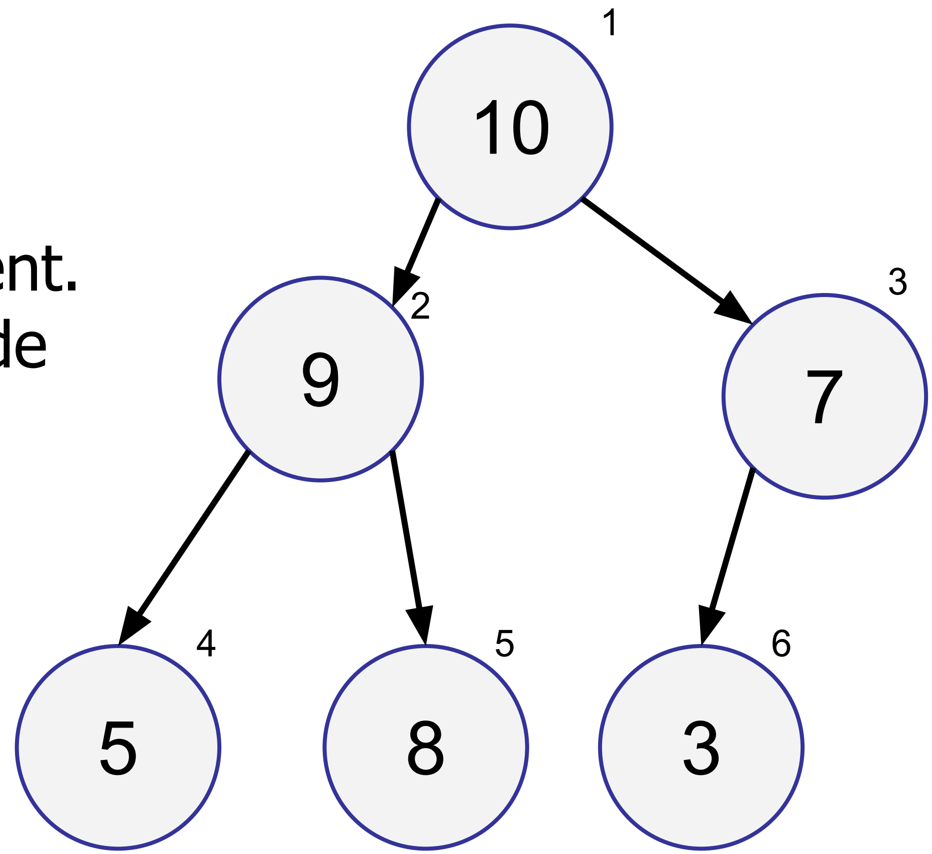


6	10	9	7	5	8	3	11
0	1	2	3	4	5	6	7

Heapsort:

HeapSort

1. swap root with last element.
2. bubble down the root node

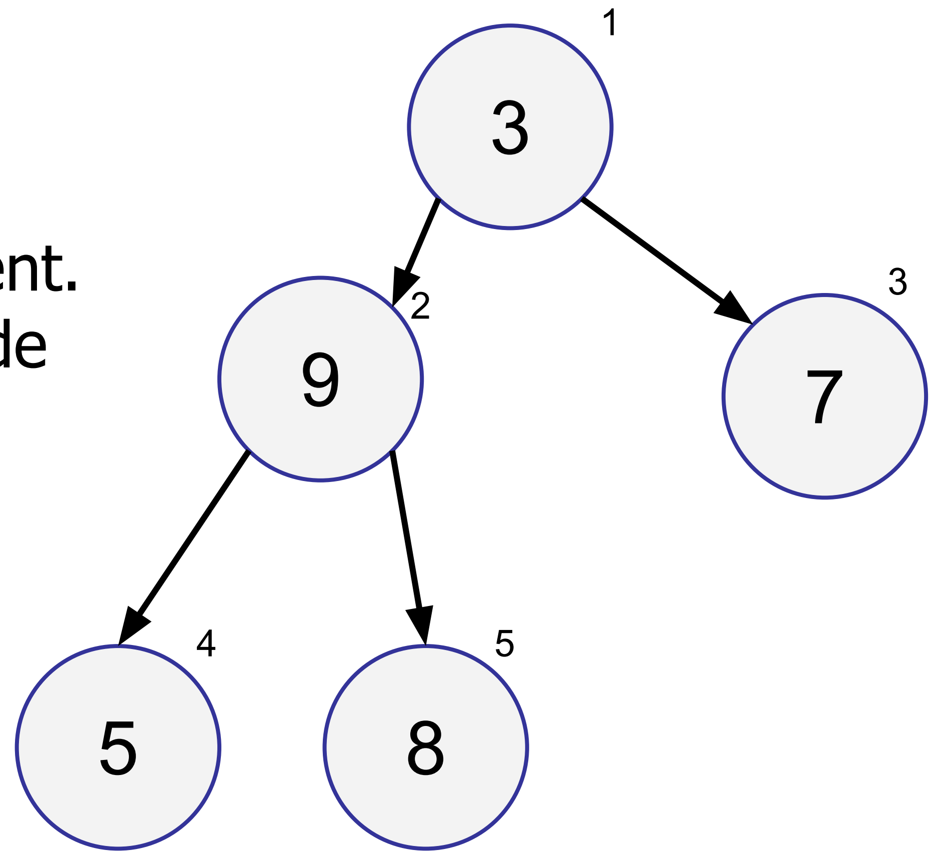


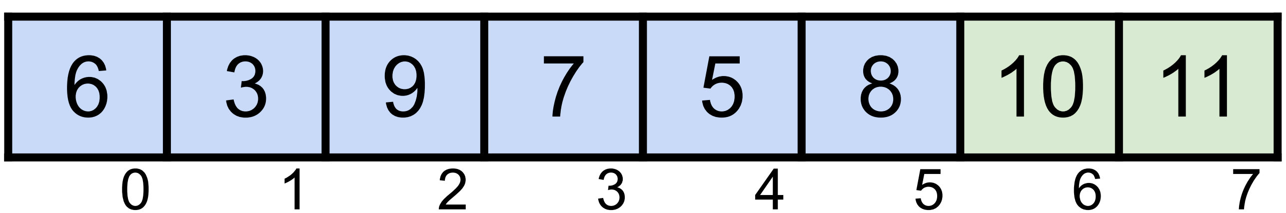
6	10	9	7	5	8	3	11
0	1	2	3	4	5	6	7

Heapsort:

HeapSort

1. swap root with last element.
2. bubble down the root node

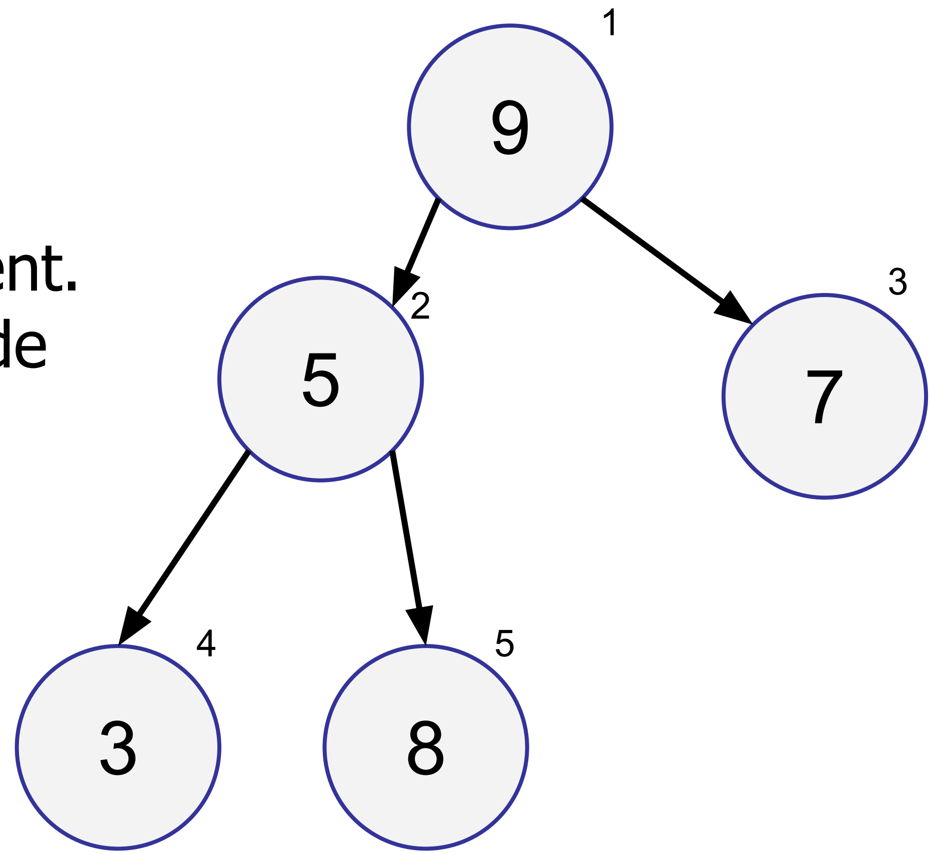




Heapsort:

HeapSort

1. swap root with last element.
2. bubble down the root node

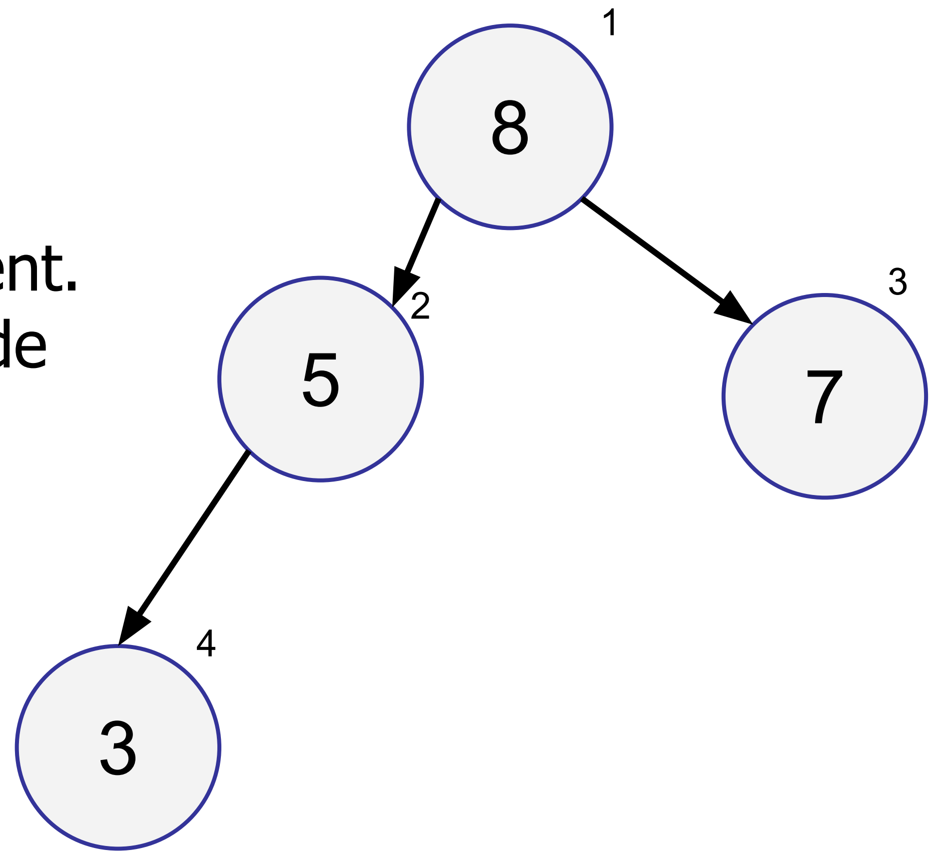


6	9	5	7	3	8	10	11
0	1	2	3	4	5	6	7

Heapsort:

HeapSort

1. swap root with last element.
2. bubble down the root node

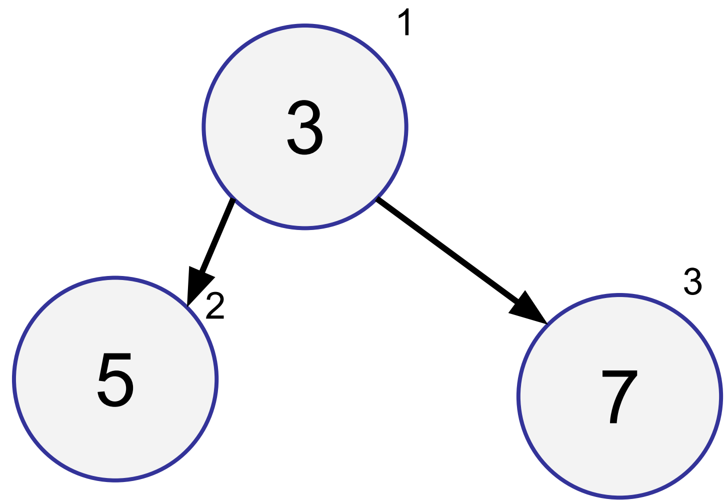


6	8	5	7	3	9	10	11
0	1	2	3	4	5	6	7

Heapsort:

HeapSort

1. swap root with last element.
2. bubble down the root node

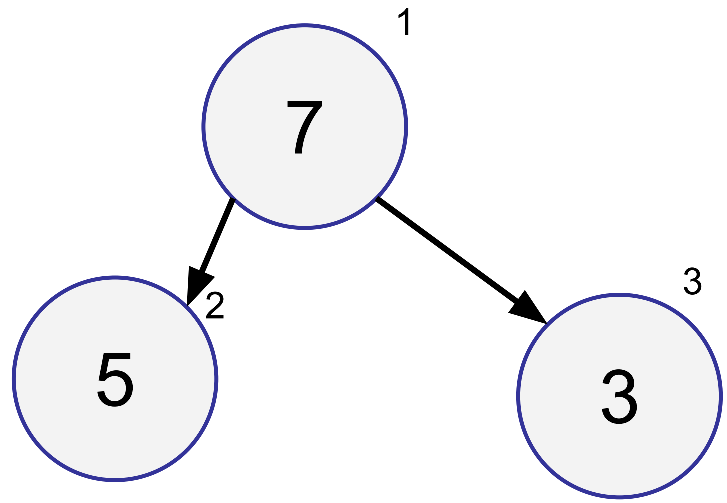


6	3	5	7	8	9	10	11
0	1	2	3	4	5	6	7

Heapsort:

HeapSort

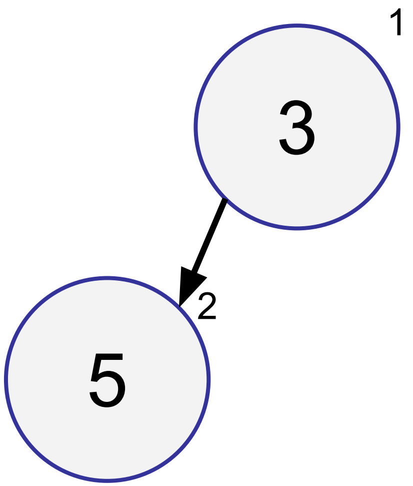
1. swap root with last element.
2. bubble down the root node



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

Heapsort:

HeapSort



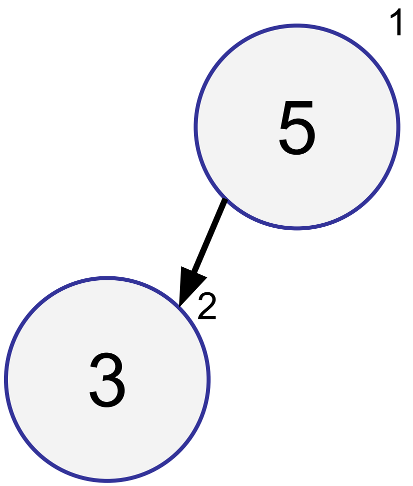
1. swap root with last element.
2. bubble down the root node

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

Heapsort:

HeapSort

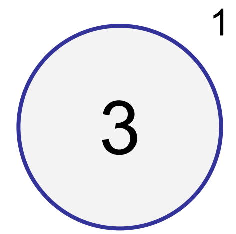
1. swap root with last element.
2. bubble down the root node



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

Heapsort:

HeapSort



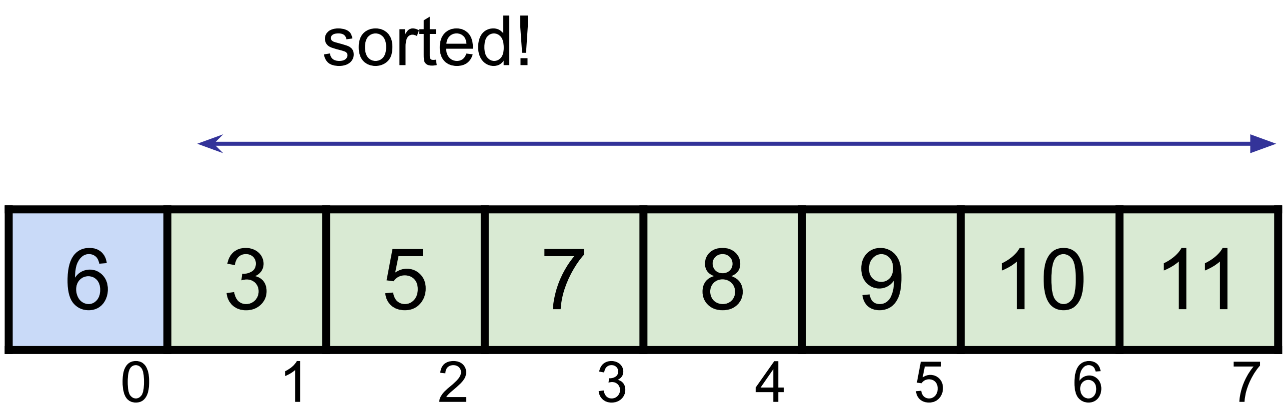
1. swap root with last element.
2. bubble down the root node

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

Heapsort:

HeapSort

1. swap root with last element.
2. bubble down the root node



Heapsort:

Algorithm	In-place?	Expected runtime	Worst case runtime	Stability
Quicksort	Yes	O(n log n)	$O(n^2)$	No
Mergesort	No. O(n) extra space	O(n log n)	O(n log n)	Yes
Heapsort	Yes	O(n log n)	O(n log n)	???

Today: Heaps and PQs

Priority Queue ADT:

– new API!

Binary Heap:

- new data structure!

Heapsort:

– new cool sorting algorithm!

Next Week: Graphs!