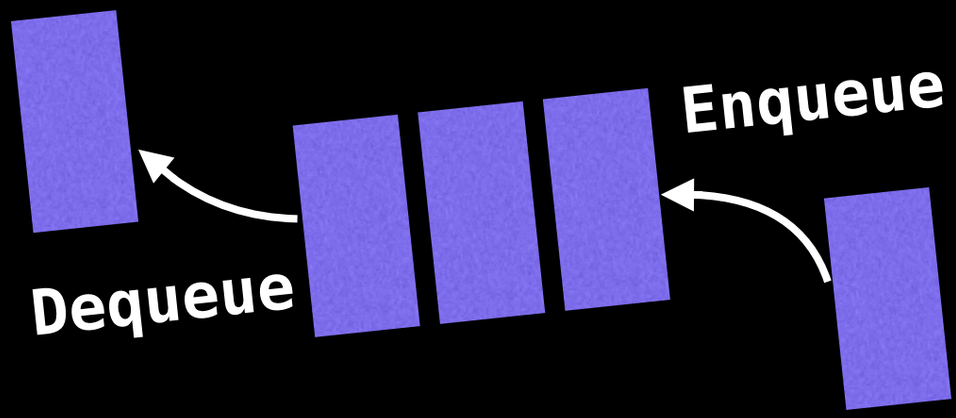
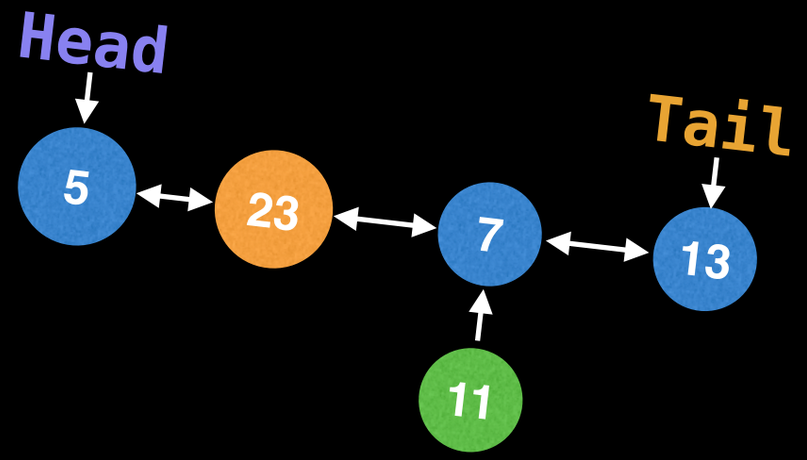
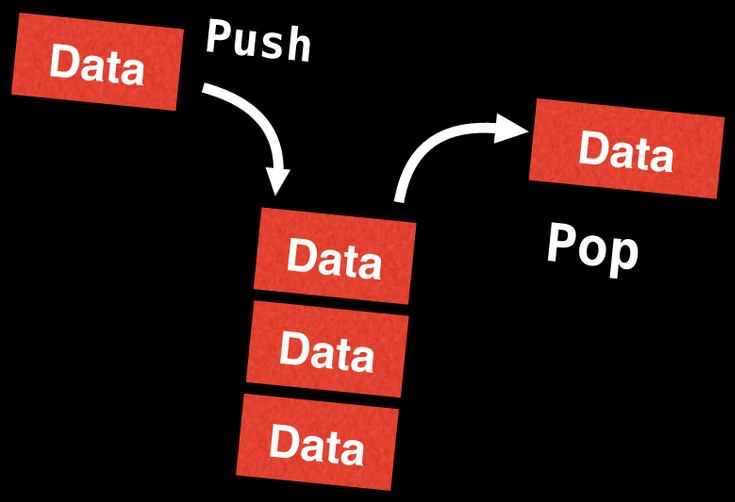
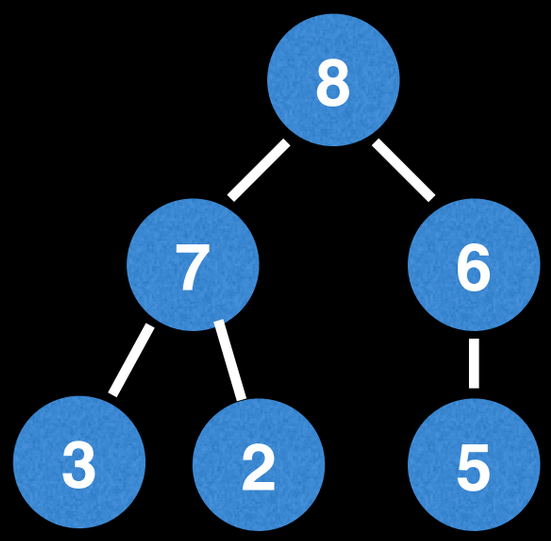
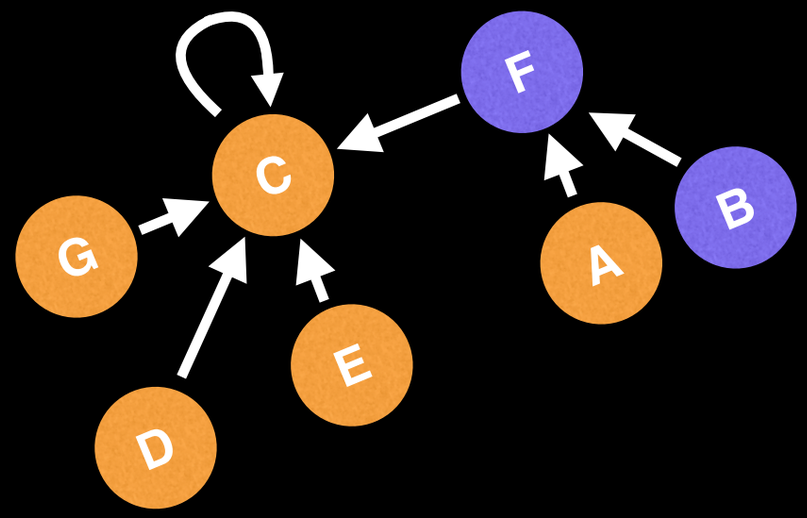




Data Structure Video Series







Indexed Priority Queues

William Fiset

Priority Queue Videos

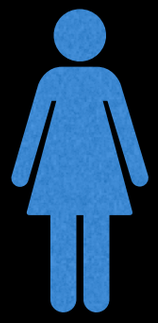
Display of previous PQ vids

What is an Indexed Priority Queue?

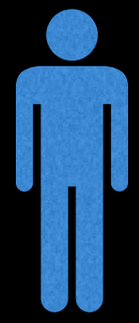
An **Indexed Priority Queue** is a traditional priority queue variant which on top of the regular PQ operations supports **quick updates and deletions** of **key-value pairs**.

Example

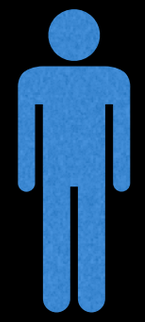
Suppose a hospital has a waiting room with **N** people which need attention with different levels of priority:



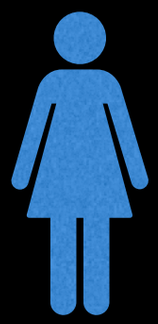
Mary is in labour
Priority: 9



Akarsh has a paper cut
Priority: 1

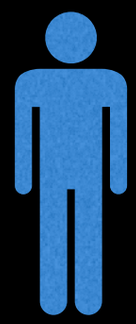


James has an arrow in his leg
Priority: 7



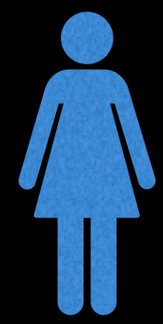
Naida’s stomach hurts

Priority: 3



Richard has a fractured wrist

Priority: 5



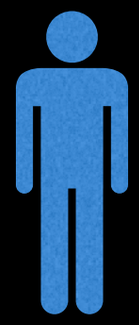
Leah’s stomach hurts

Priority: 3

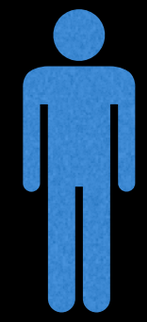
Example

The hospital serves Mary first.

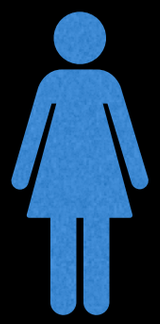
Mary is in labour
Priority: 9



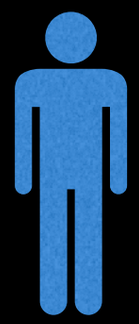
Akarsh has a paper cut
Priority: 1



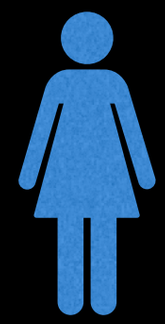
James has an arrow in his leg
Priority: 7



Naida’s stomach hurts
Priority: 3



Richard has a fractured wrist
Priority: 5



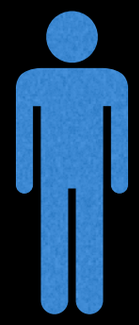
Leah’s stomach hurts
Priority: 3

Example

Followed by James

#1

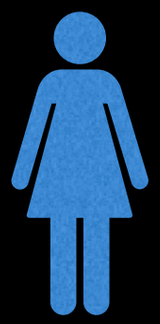
Mary is in labour
Priority: 9



Akarsh has a paper cut
Priority: 1

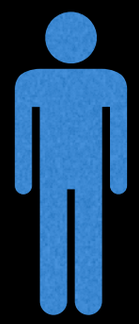
#2

James has an arrow in his leg
Priority: 7



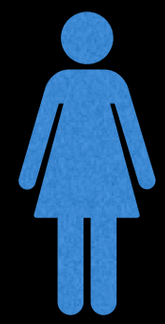
Naida’s stomach hurts

Priority: 3



Richard has a fractured wrist

Priority: 5



Leah’s stomach hurts

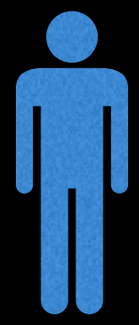
Priority: 3

Example

Then something happens, Naida’s condition worsens as she starts vomiting. Her priority gets updated to 6

#1

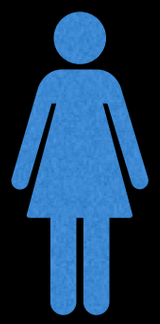
Mary is in labour
Priority: 9



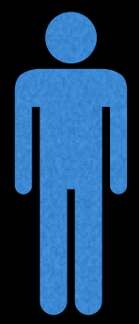
Akarsh has a paper cut
Priority: 1

#2

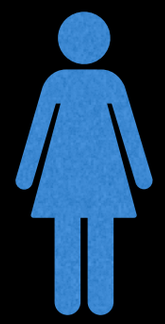
James has an arrow in his leg
Priority: 7



Naida’s stomach hurts + vomiting
Priority: 6



Richard has a fractured wrist
Priority: 5



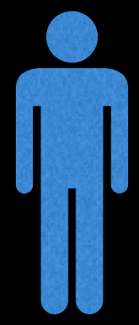
Leah’s stomach hurts
Priority: 3

Example

Naida gets served when the hospital is finished with James.

#1

Mary is in labour
Priority: 9



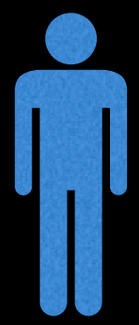
Akarsh has a paper cut
Priority: 1

#2

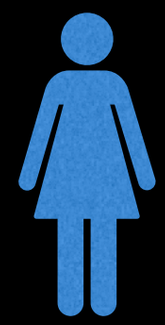
James has an arrow in his leg
Priority: 7

#3

Naida’s stomach hurts + vomiting
Priority: 6



Richard has a fractured wrist
Priority: 5



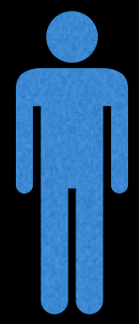
Leah’s stomach hurts
Priority: 3

Example

Richard gets impatient while Naida gets served and leaves to go to the clinic down the street.

#1

Mary is in labour
Priority: 9



Akarsh has a paper cut
Priority: 1

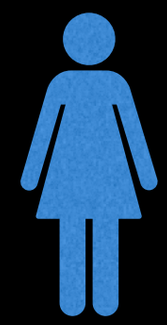
#2

James has an arrow in his leg
Priority: 7

#3

Naida’s stomach hurts + vomiting
Priority: 6

Richard has a fractured wrist
Priority: 5



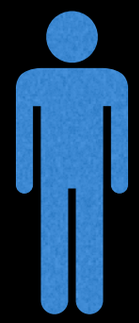
Leah’s stomach hurts
Priority: 3

Example

While Akarsh goes to take a drink of water he slips and cracks his skull open. His priority is increased to 10

#1

Mary is in labour
Priority: 9



Akarsh has a paper cut + open skull
Priority: 10

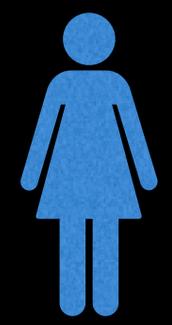
#2

James has an arrow in his leg
Priority: 7

#3

Naida’s stomach hurts + vomiting
Priority: 6

Richard has a fractured wrist
Priority: 5



Leah’s stomach hurts
Priority: 3

Example

Akarsh gets served next.

#1 

Mary is in labour
Priority: 9

#4 

Akarsh has a paper cut + open skull
Priority: 10

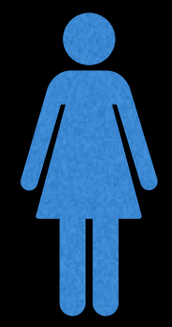
#2 

James has an arrow in his leg
Priority: 7

#3 

Naida’s stomach hurts + vomiting
Priority: 6

Richard has a fractured wrist
Priority: 5



Leah’s stomach hurts
Priority: 3

Example

Followed by Leah.

#1 

Mary is in labour
Priority: 9

#4 

Akarsh has a paper cut + open skull
Priority: 10

#2 

James has an arrow in his leg
Priority: 7

#3 

Naida’s stomach hurts + vomiting
Priority: 6

Richard has a fractured wrist
Priority: 5

#5 

Leah’s stomach hurts
Priority: 3

Example

Followed by Leah.

#1 

Mary is in labour
Priority: 9

#4 

Akarsh has a paper cut + open skull
Priority: 10

#2 

James has an arrow in his leg
Priority: 7

#3 

Naida’s stomach hurts + vomiting
Priority: 6

Richard has a fractured wrist
Priority: 5

#5 

Leah’s stomach hurts
Priority: 3

Usefulness of IPQ

In the hospital example, we saw that it was very important to be able to **dynamically update the priority** (value) of certain people (keys).

The **Indexed Priority Queue (IPQ)** data structure lets us do this efficiently. The first step to using an IPQ is to assign index values to all the keys forming a bidirectional mapping.

Create a bidirectional mapping between your **N** keys and the domain [0, **N**) using a bidirectional **hashtable**.

Key		Key Index (ki)
“Mary”	⟷	0
“Akarsh“	⟷	1
“James”	⟷	2
“Naida”	⟷	3
“Richard”	⟷	4
“Leah”	⟷	5

NOTE: This assumes you know how many keys you will have in your IPQ, but this mapping can be constructed dynamically as well

Reason for mapping

Q: Why are we mapping keys to indexes in the domain [0, N)?

A: Typically priority queues are implemented as heaps under the hood which internally use arrays which we want to facilitate indexing into.

NOTE: Often the keys themselves are integers in the range [0, N) so there’s no need for the mapping, but it’s handy to be able to support any type of key (like names).

IPQ ADT Interface

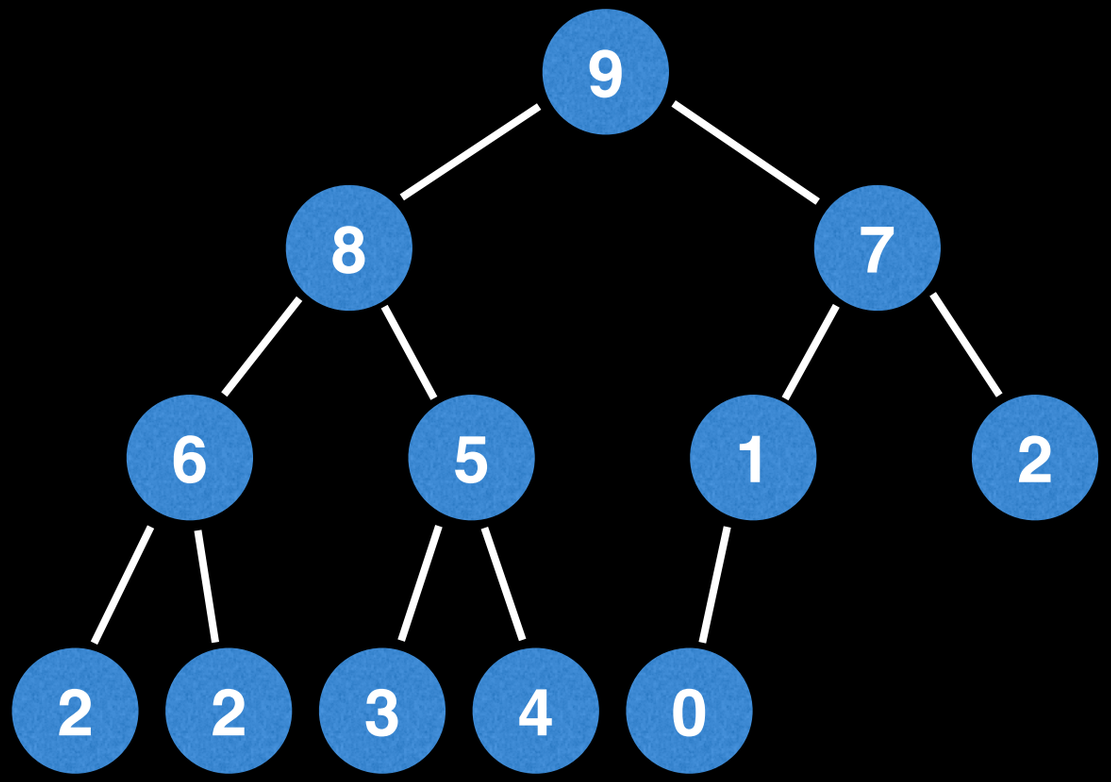
If ‘k’ is the key we want to update
first get the key’s index: **ki = map[k]**,
then use ‘ki’ with the IPQ

```
delete(ki)
valueOf(ki)
contains(ki)
peekMinKeyIndex()
pollMinKeyIndex()
peekMinValue()
pollMinValue()
insert(ki, value)
update(ki, value)
decreaseKey(ki, value)
increaseKey(ki, value)
```

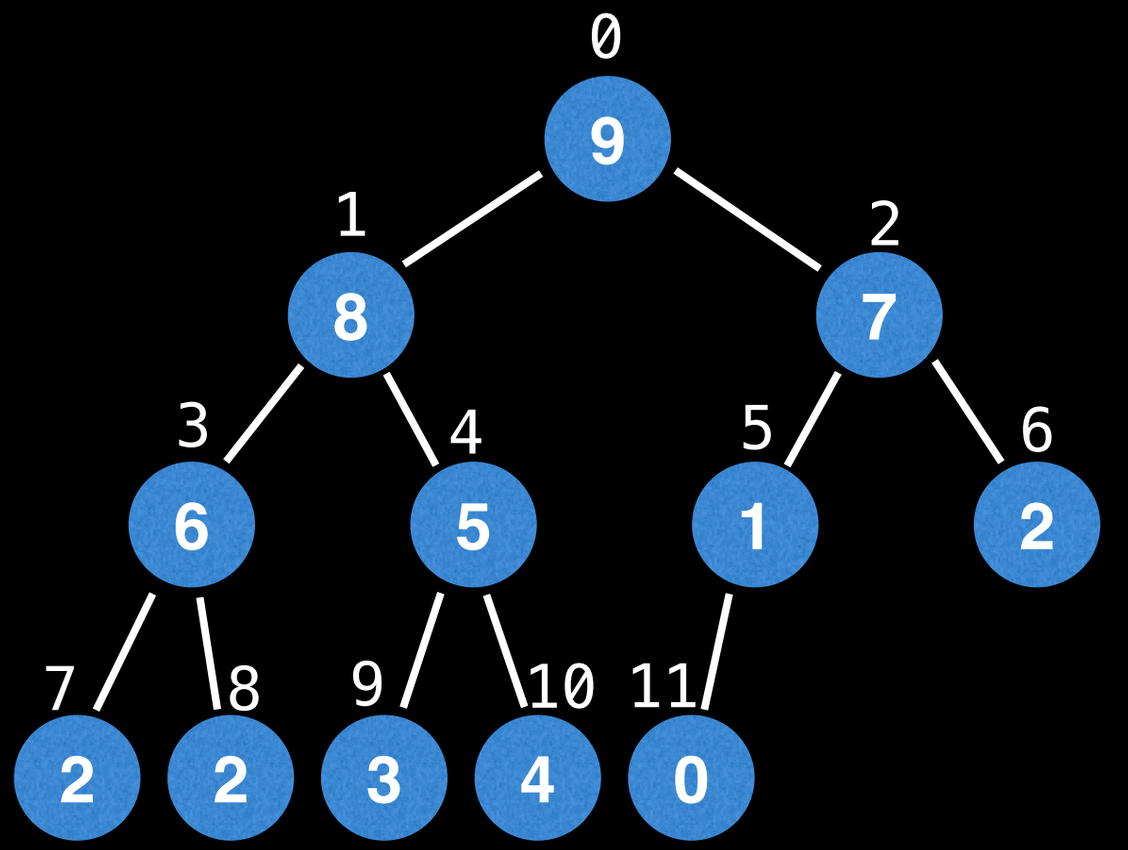
IPQ as a binary heap

Operation	Indexed Binary Heap PQ
`delete(ki)`	O(log(n))
`valueOf(ki)`	O(1)
`contains(ki)`	O(1)
`peekMinKeyIndex()`	O(1)
`pollMinKeyIndex()`	O(log(n))
`peekMinValue()`	O(1)
`pollMinValue()`	O(log(n))
`insert(ki, value)`	O(log(n))
`update(ki, value)`	O(log(n))
`decreaseKey(ki, value)`	O(log(n))
`increaseKey(ki, value)`	O(log(n))

Refresher on the binary heap DS

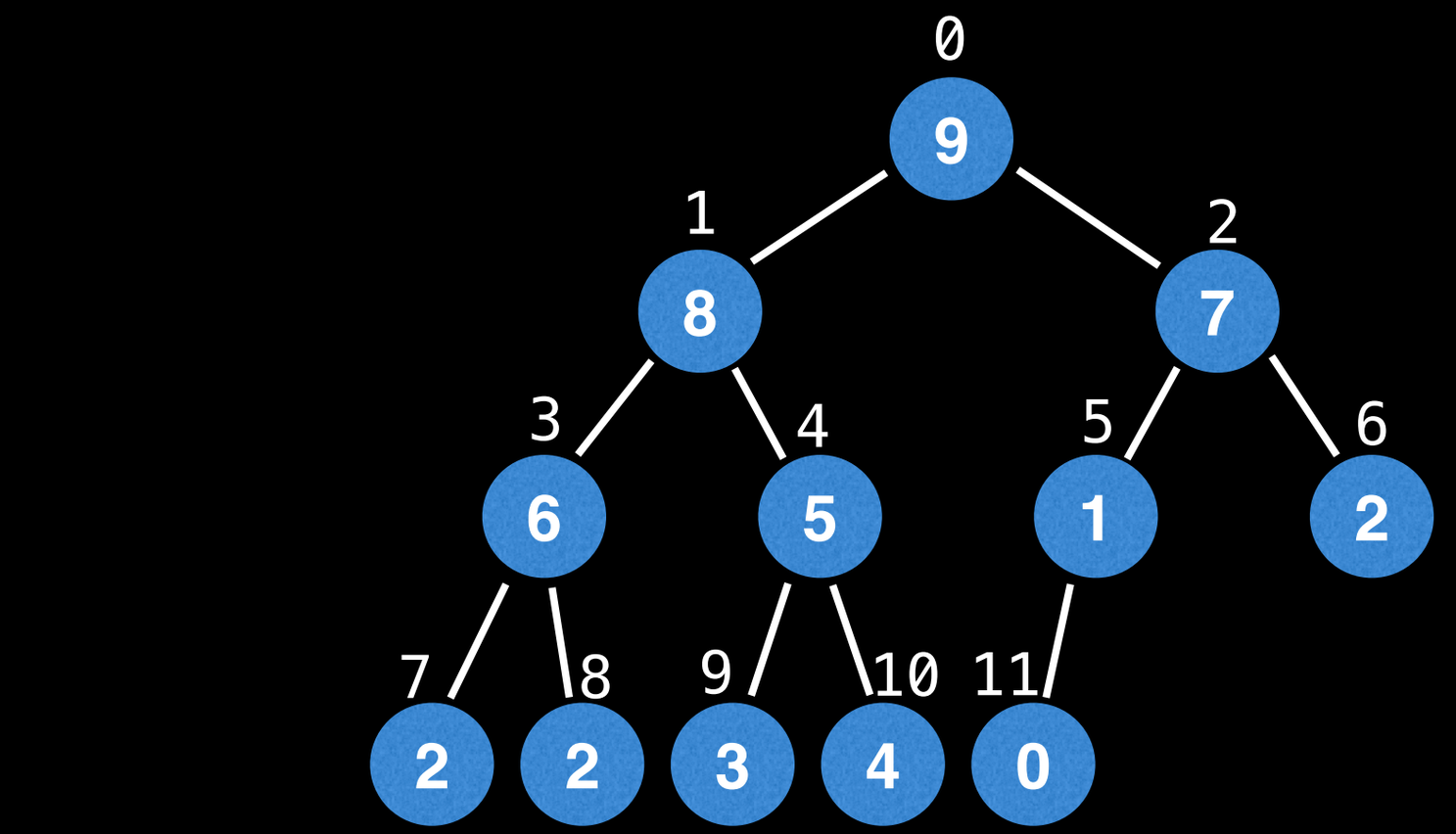


Refresher on the binary heap DS



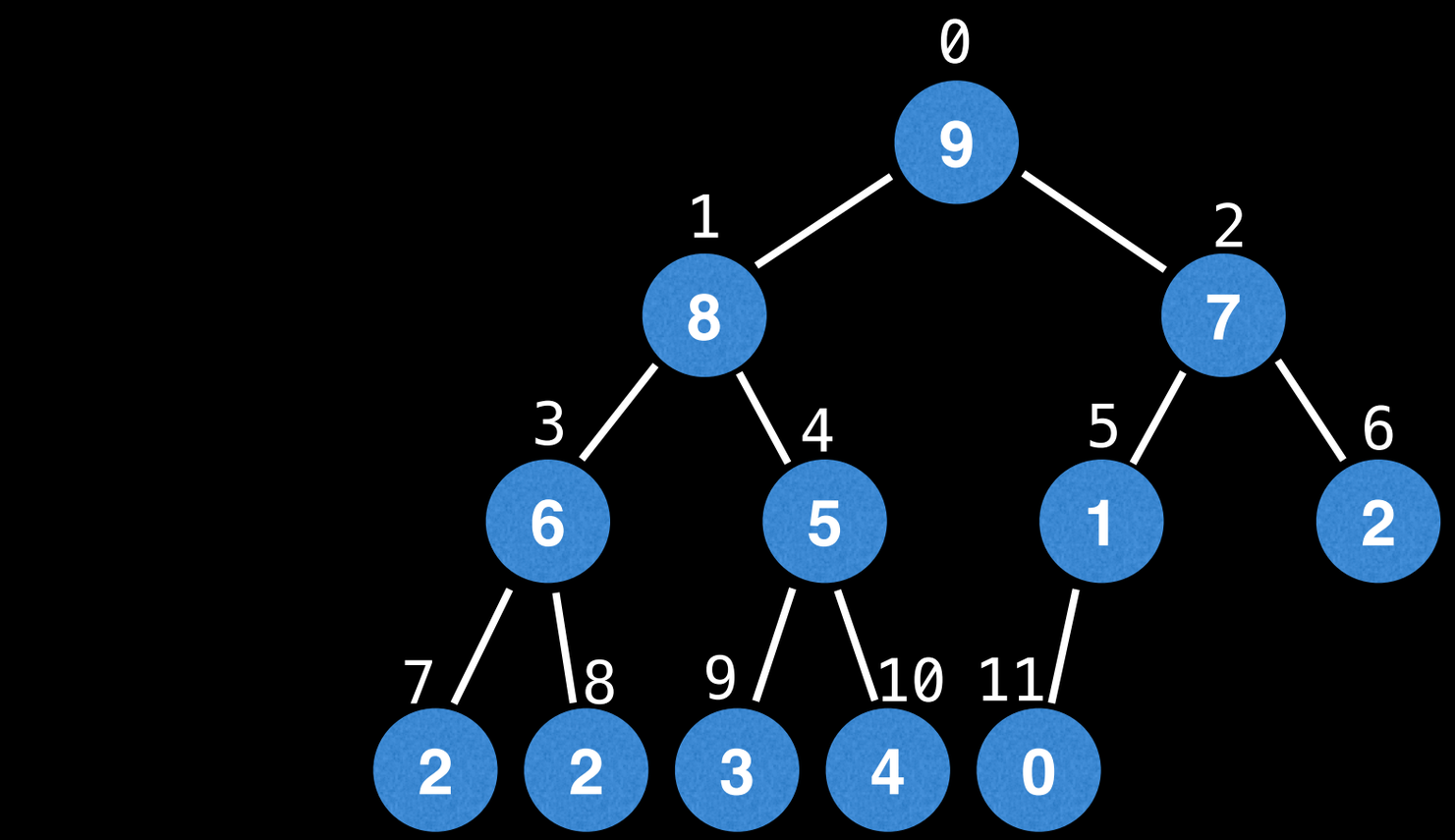
Recall that a very common way to represent a binary heap is with an array since every node is indexed sequentially.

Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	2	2	3	4	0	n/a	n/a	n/a

Refresher on the binary heap DS

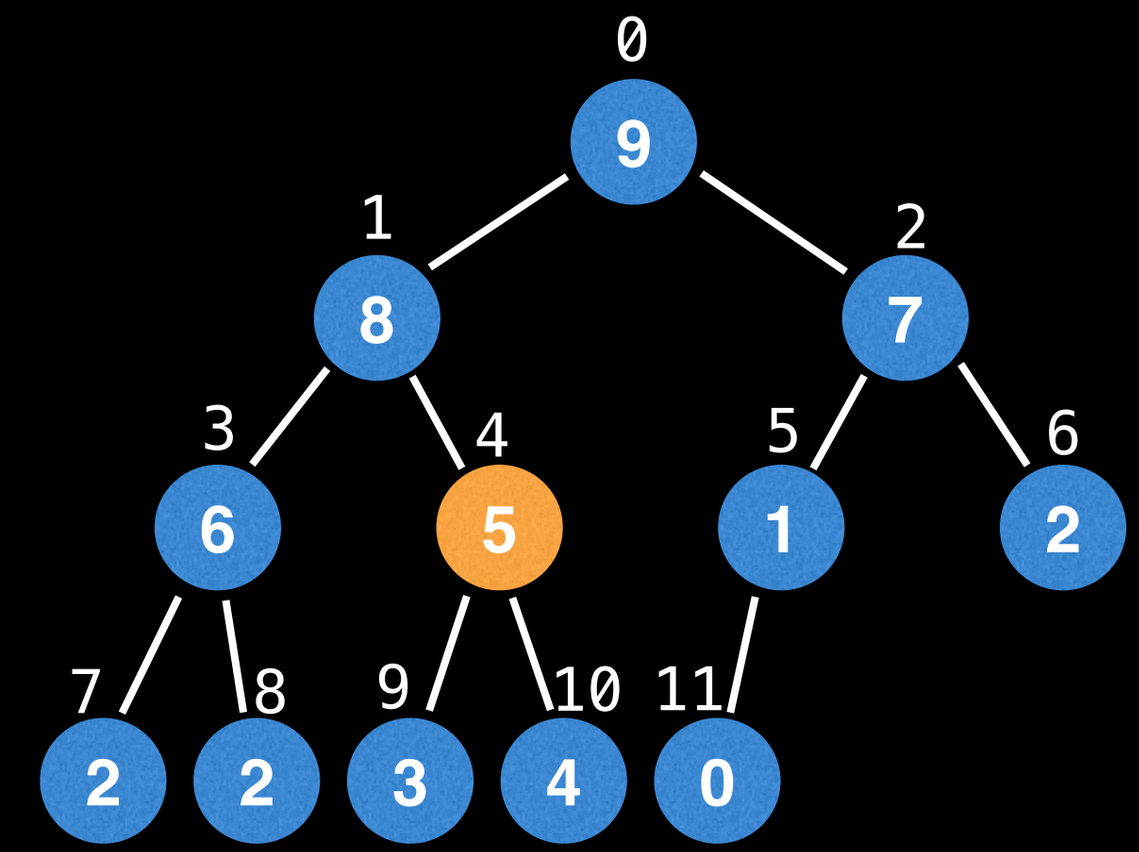


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	2	2	3	4	0	n/a	n/a	n/a

Let *i* be the current node.

Left child index: 2i + 1
Right child index: 2i + 2
(zero based)

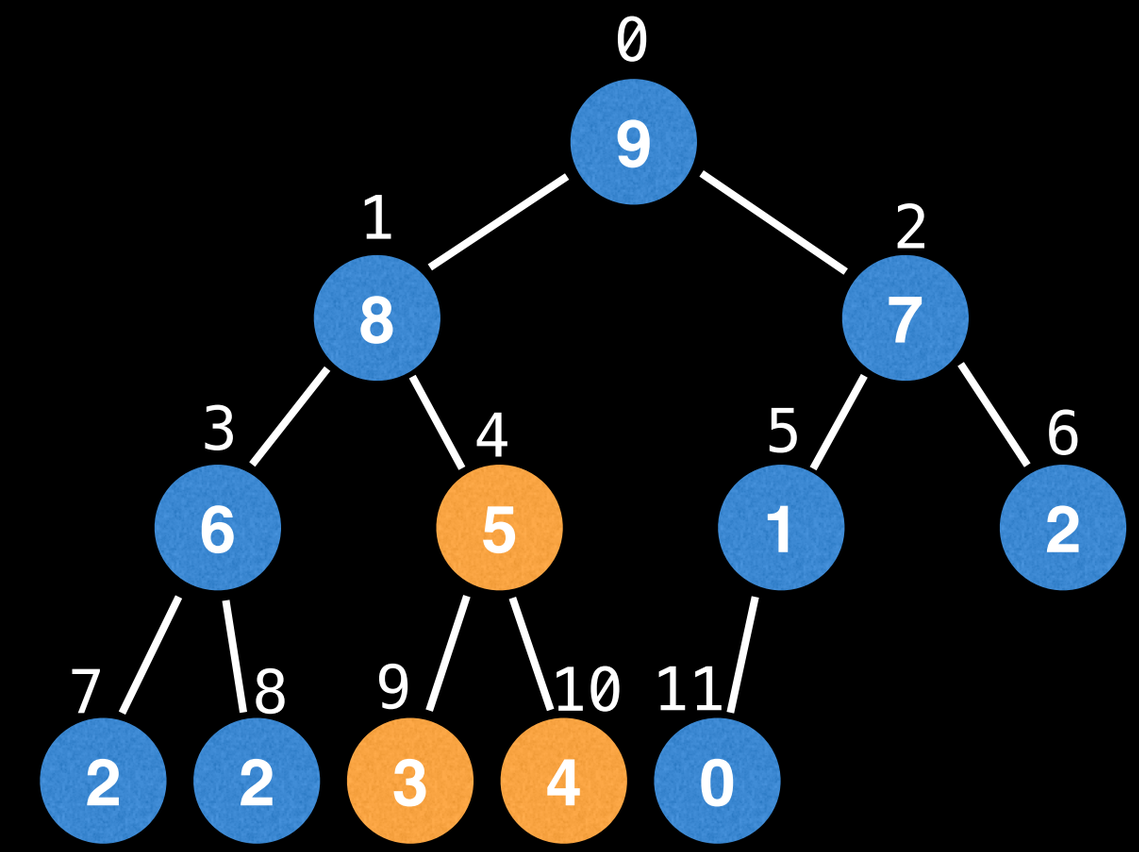
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	2	2	3	4	0	n/a	n/a	n/a

Q: What are the children of the node at index 4?

Refresher on the binary heap DS

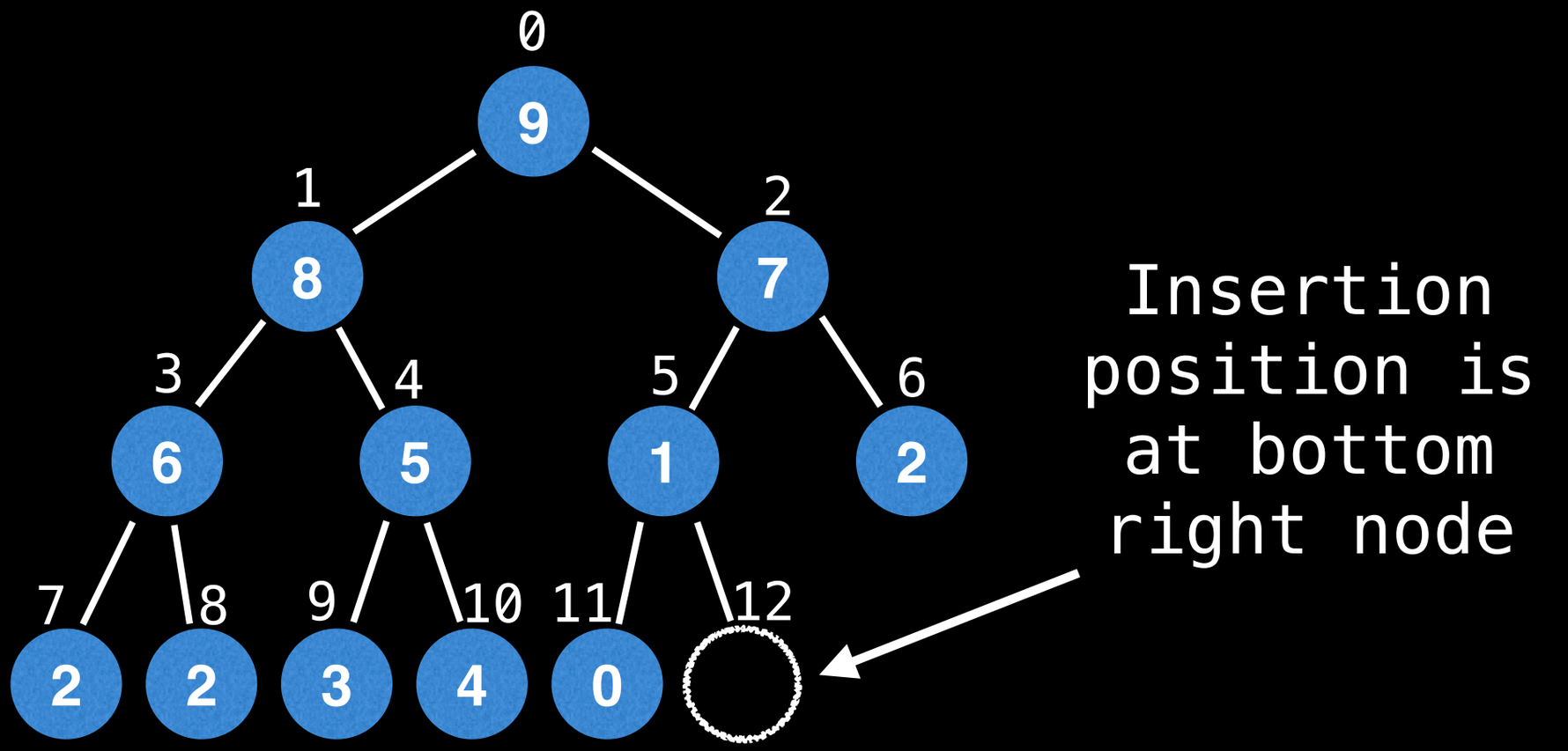


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	2	2	3	4	0	n/a	n/a	n/a

Q: What are the children of the node at index 4?

Left child = 2*4 + 1 = 9
Right child = 2*4 + 2 = 10

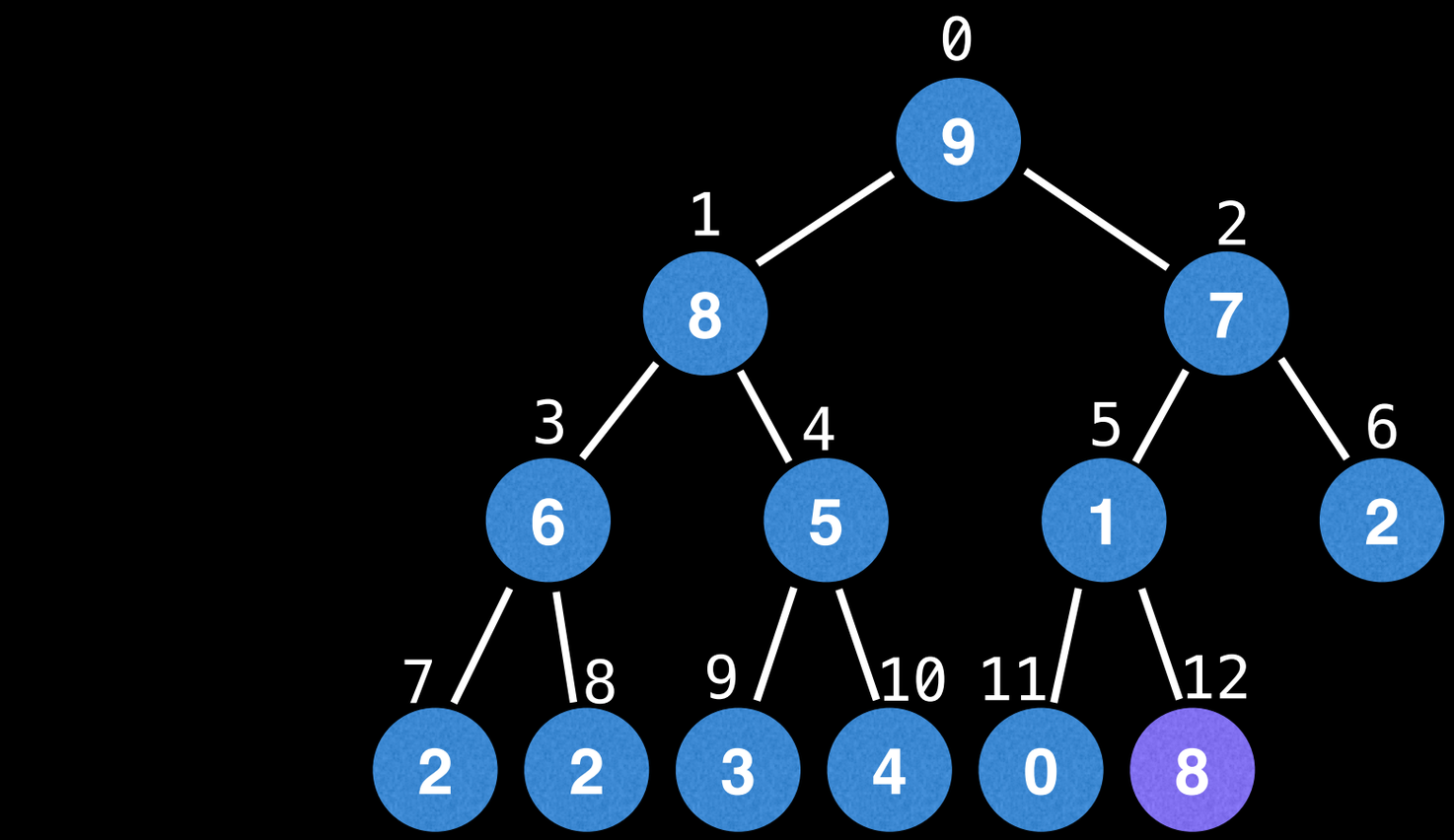
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	2	2	3	4	0	n/a	n/a	n/a

Place values you want to insert into the PQ at the **insertion position at the bottom right** of the binary tree. Doing this ensures a **complete tree structure** is always maintained.

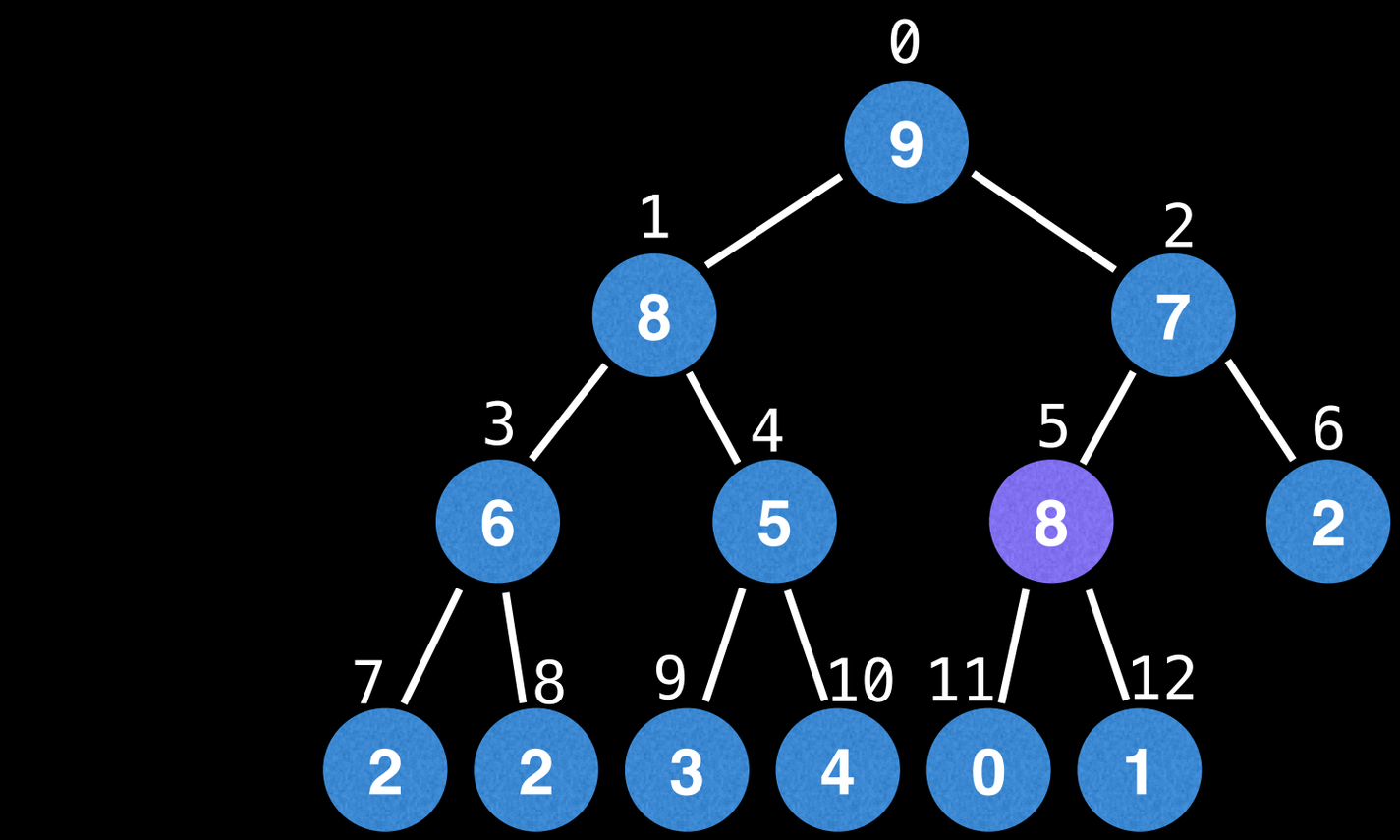
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	2	2	3	4	0	8	n/a	n/a

Suppose we insert the value 8. This would violate the **heap invariant**, so we bubble/sift up the value until the invariant is met.

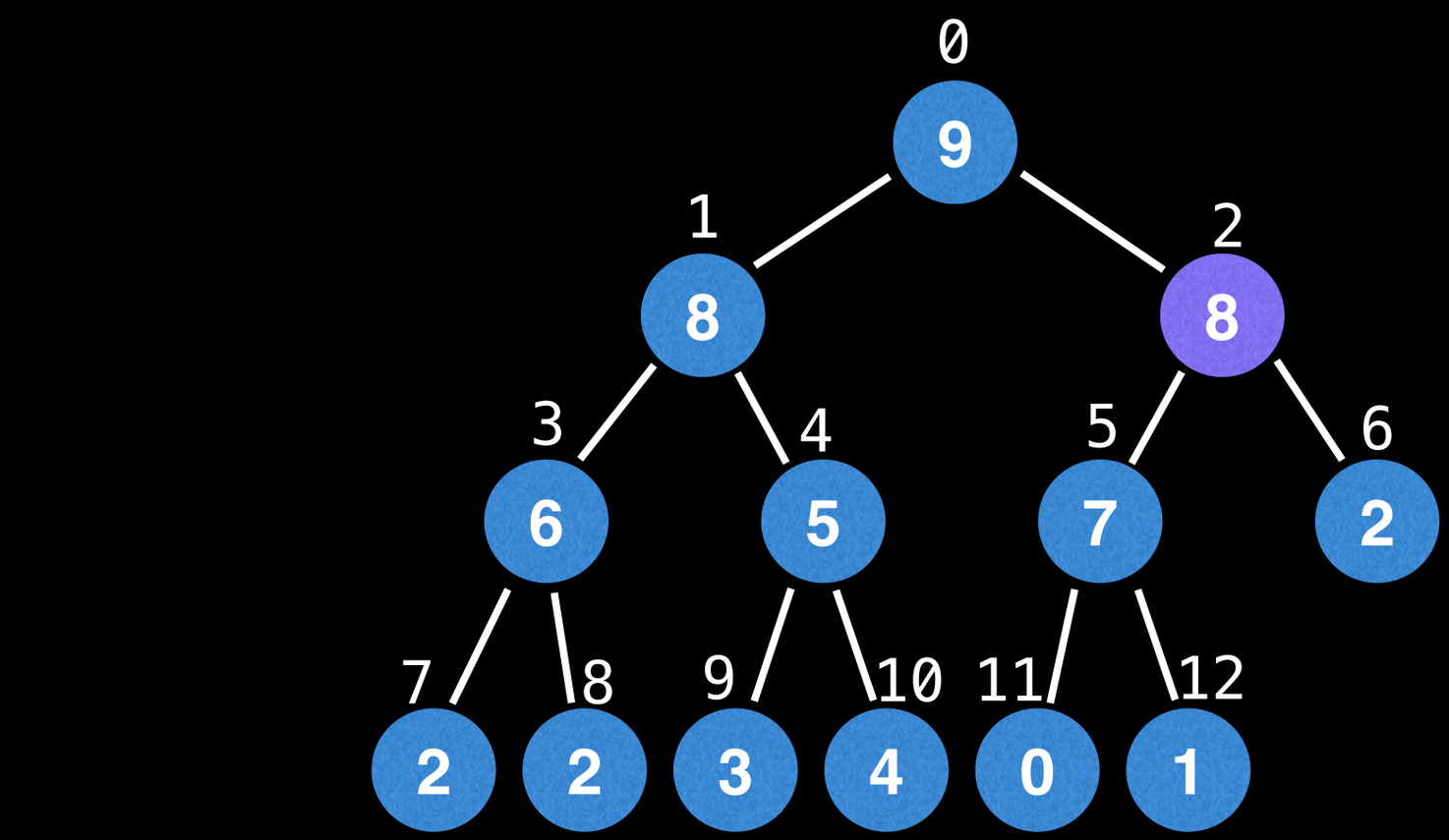
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	8	2	2	2	3	4	0	1	n/a	n/a

Suppose we insert the value 8. This would violate the **heap invariant**, so we bubble/sift up the value until the invariant is met.

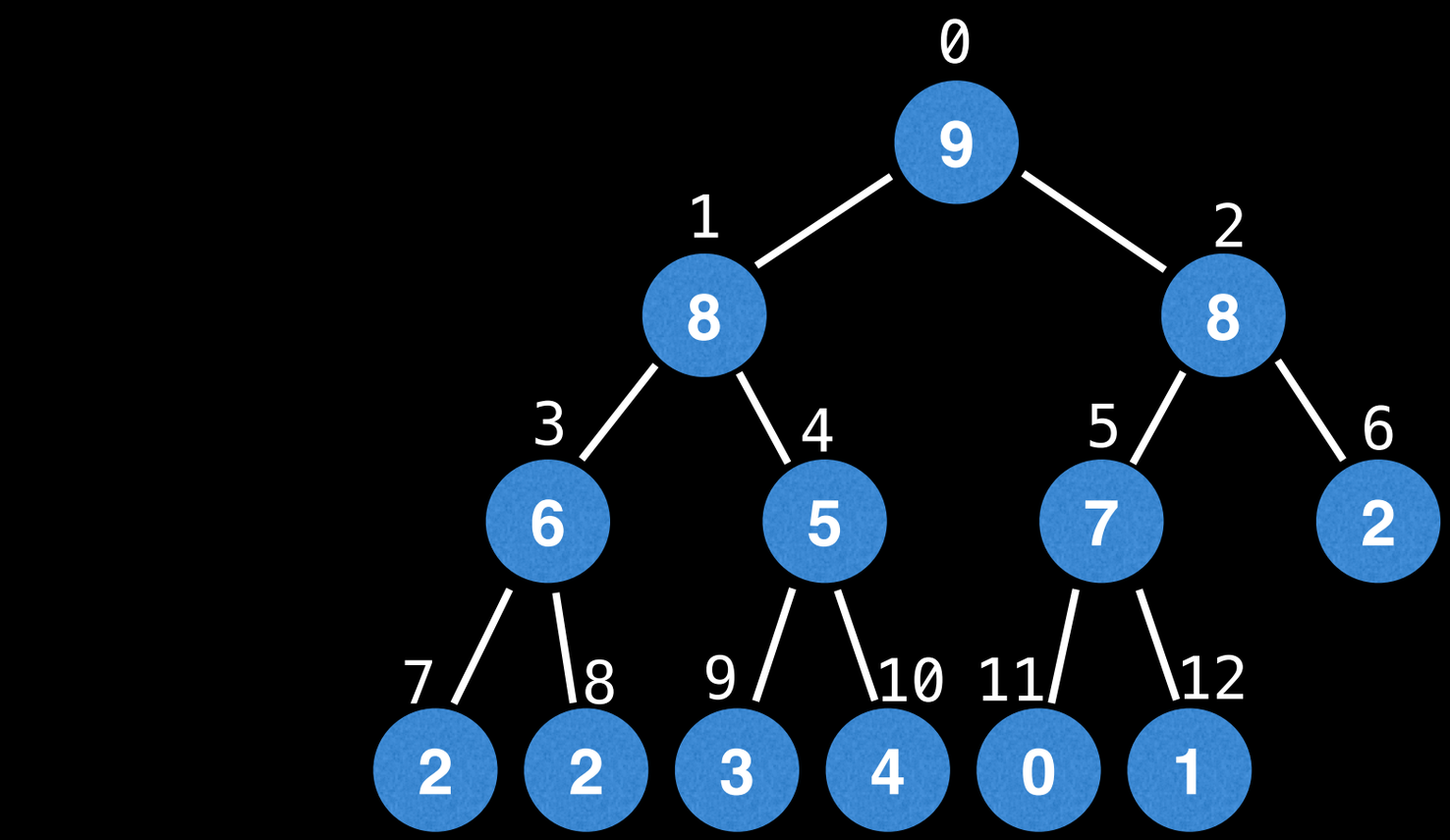
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Suppose we insert the value 8. This would violate the **heap invariant**, so we bubble/sift up the value until the invariant is met.

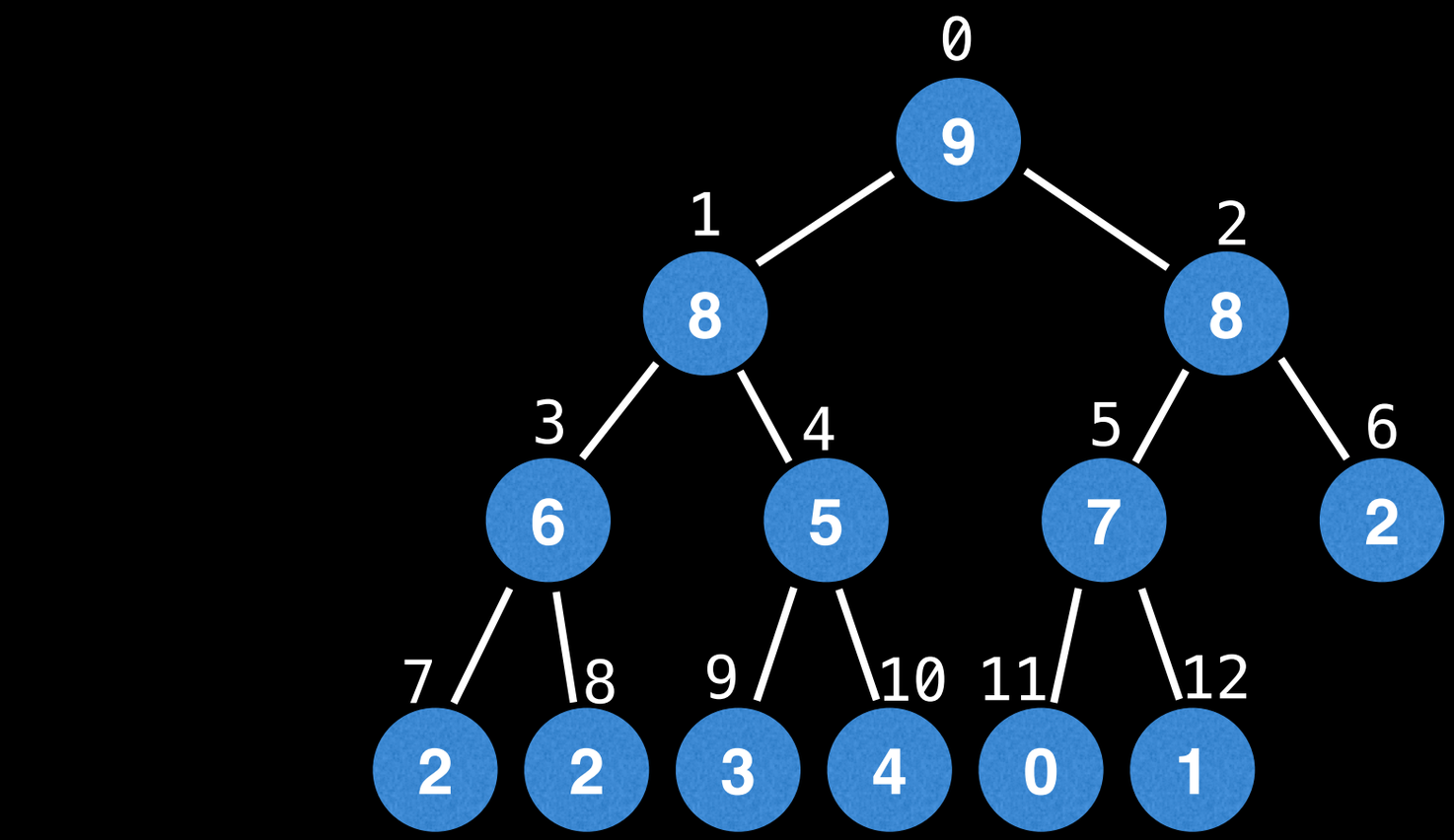
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Suppose we insert the value 8. This would violate the **heap invariant**, so we bubble/sift up the value until the invariant is met.

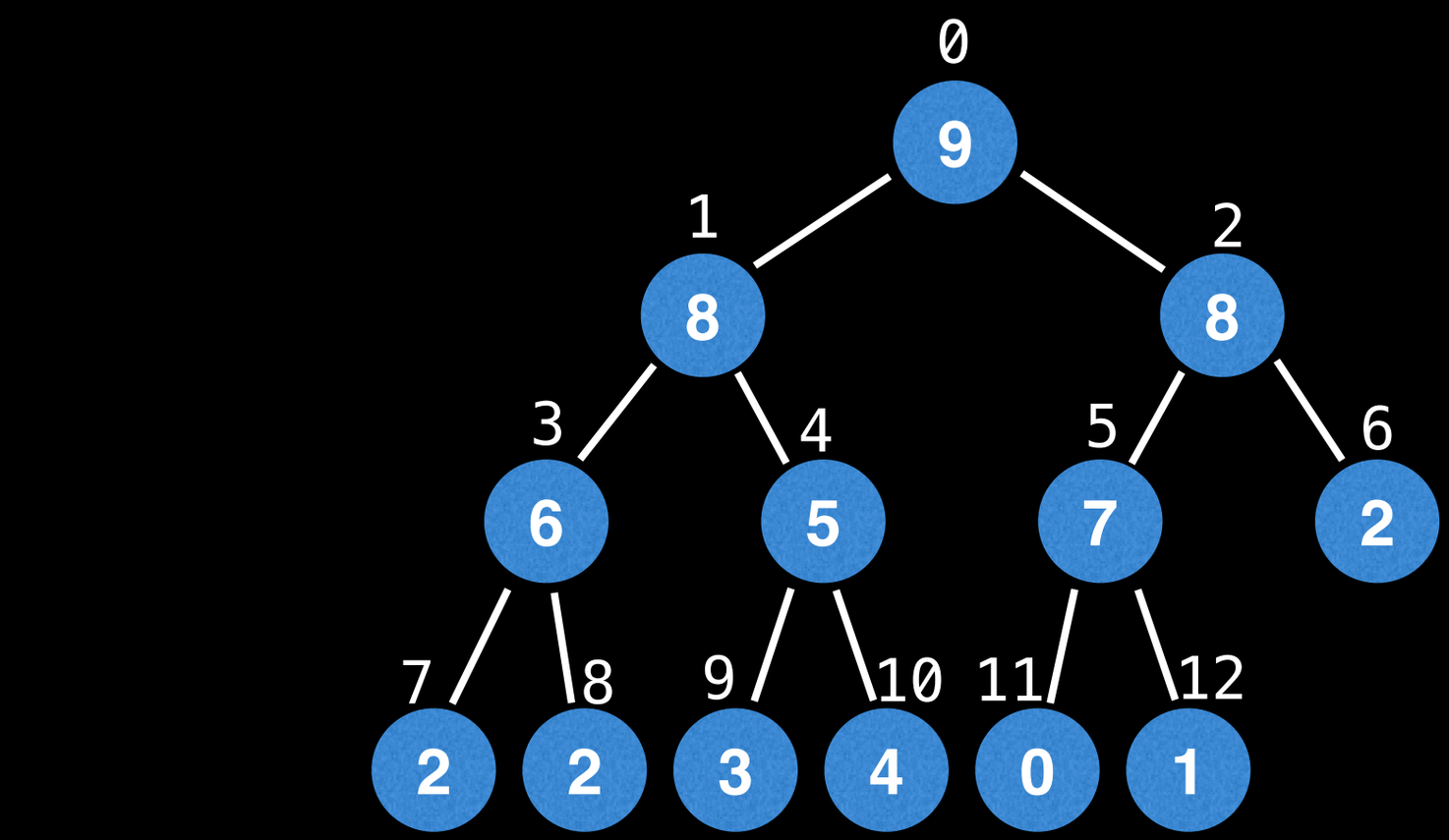
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

In a traditional PQ, to remove items, search for the element you want to remove and then swap with last node, perform removal and finally bubble up or down the swapped value.

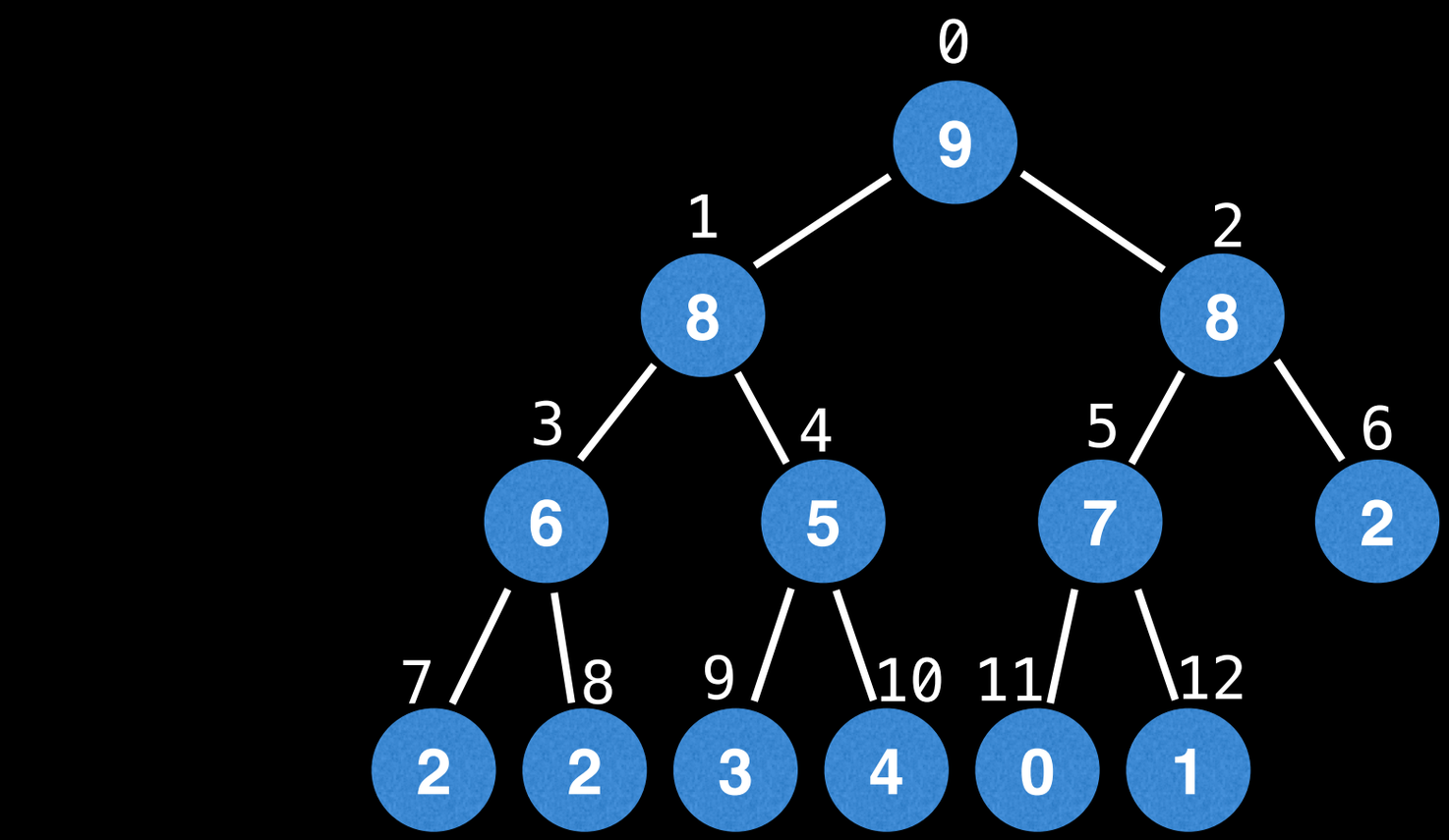
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Remove node with value 5.

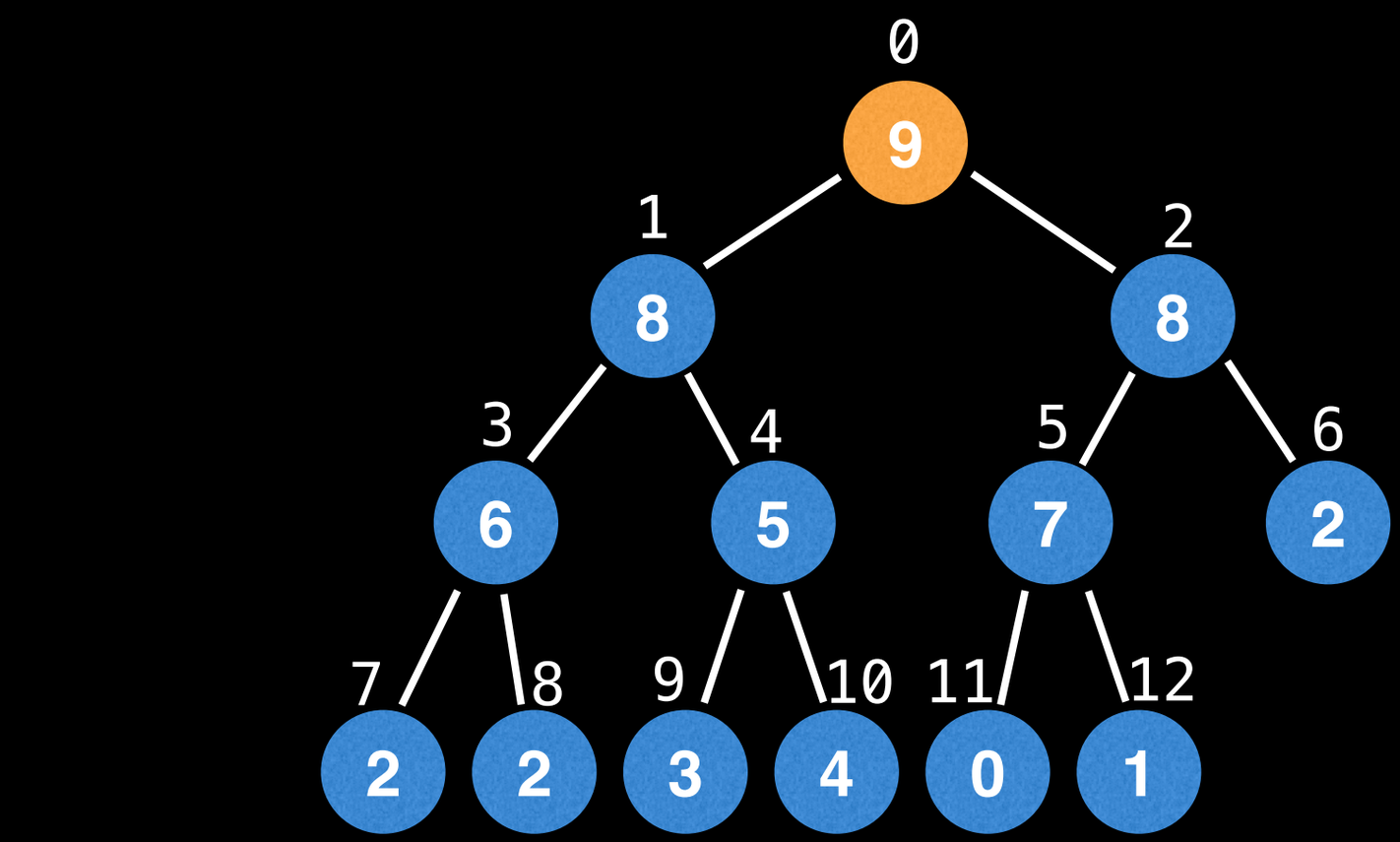
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Search for node with value 5.

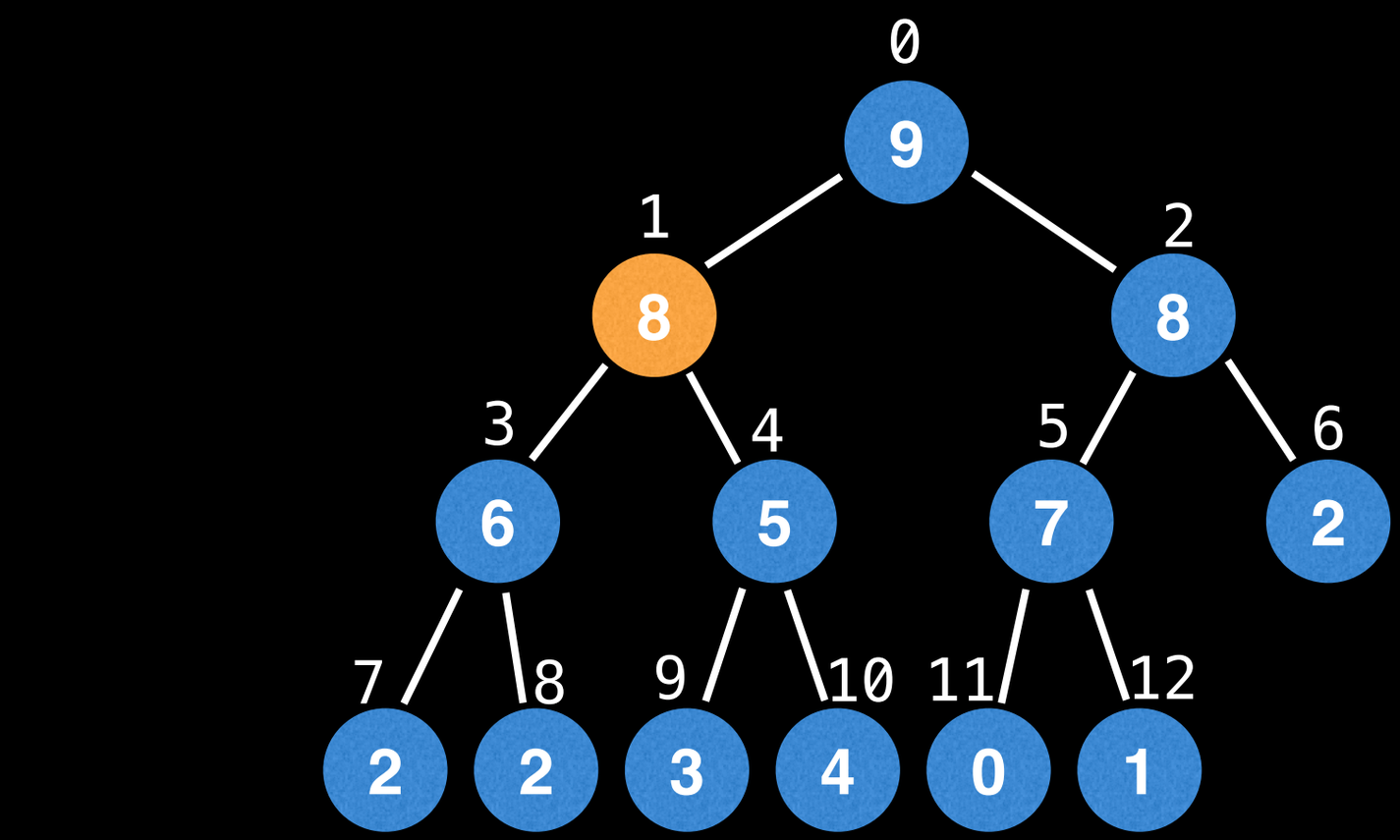
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Search for node with value 5.

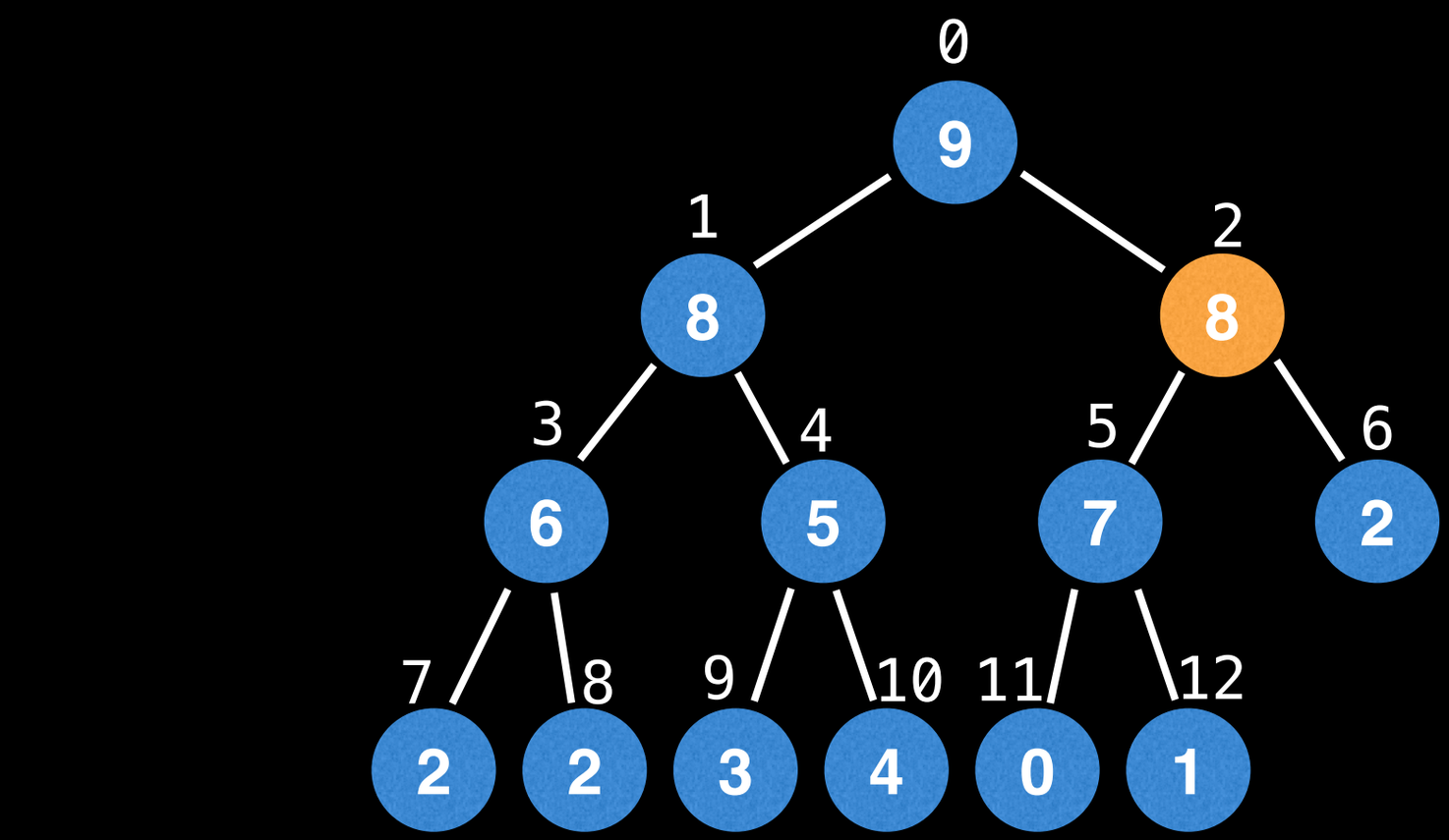
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Search for node with value 5.

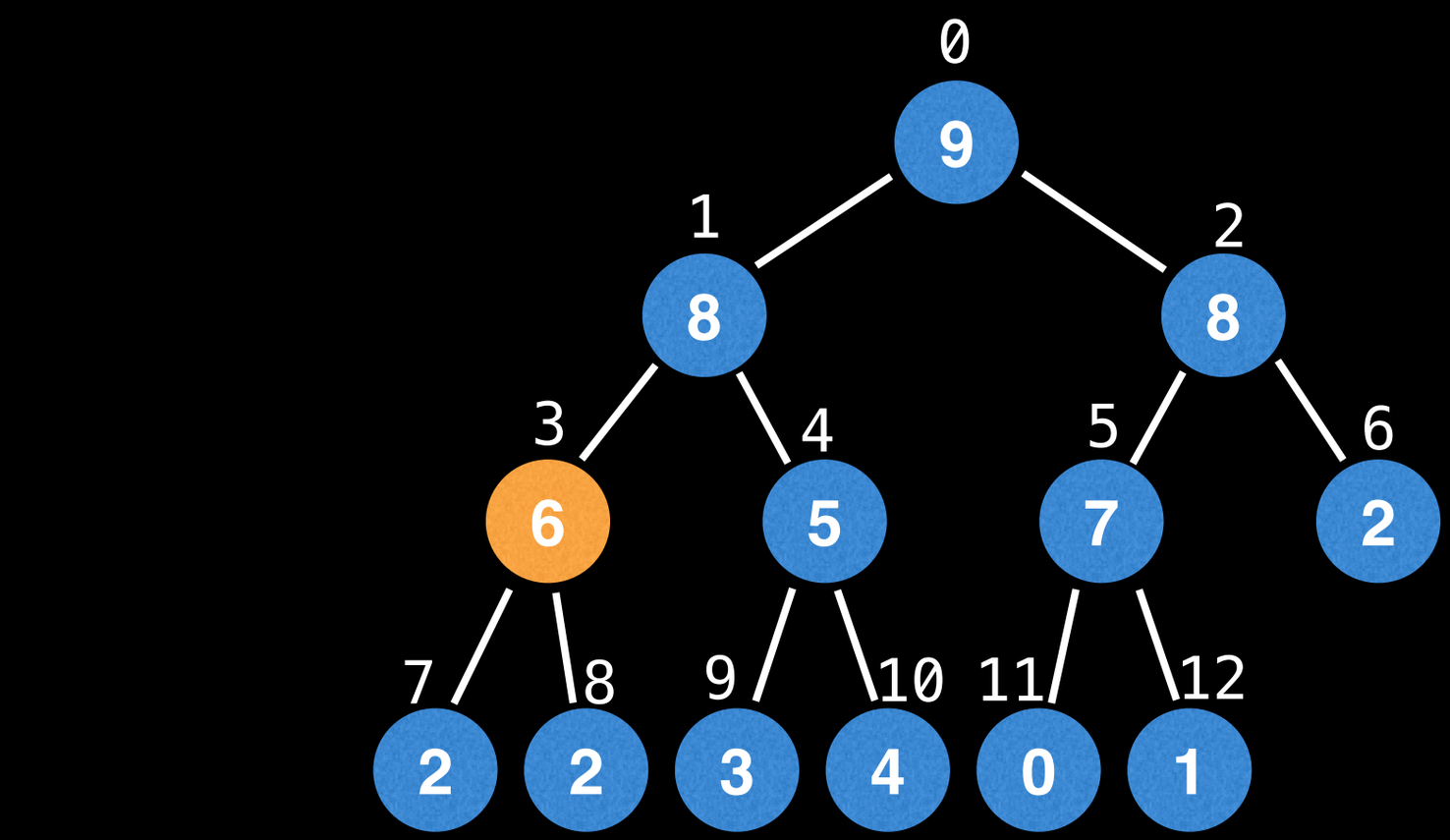
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Search for node with value 5.

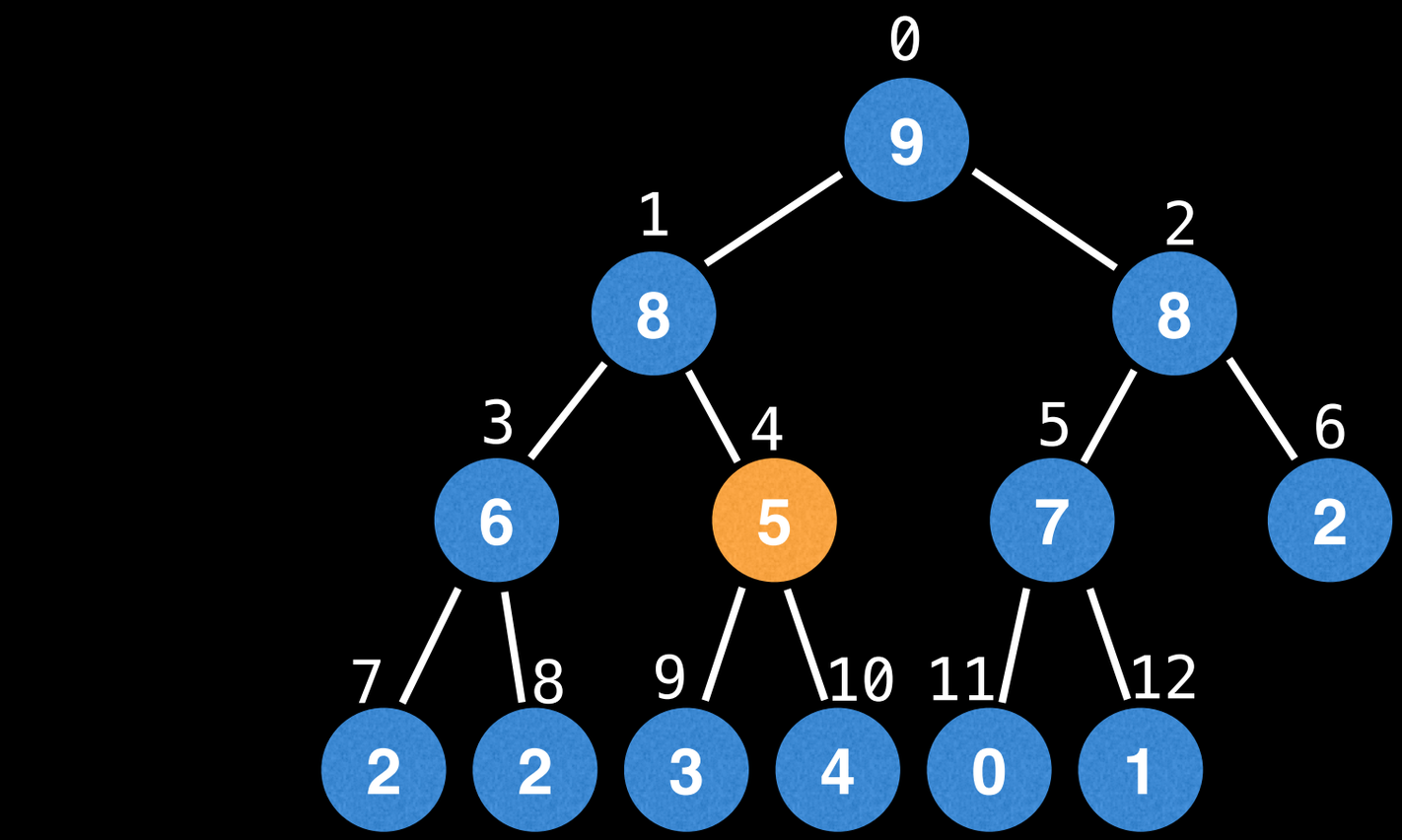
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Search for node with value 5.

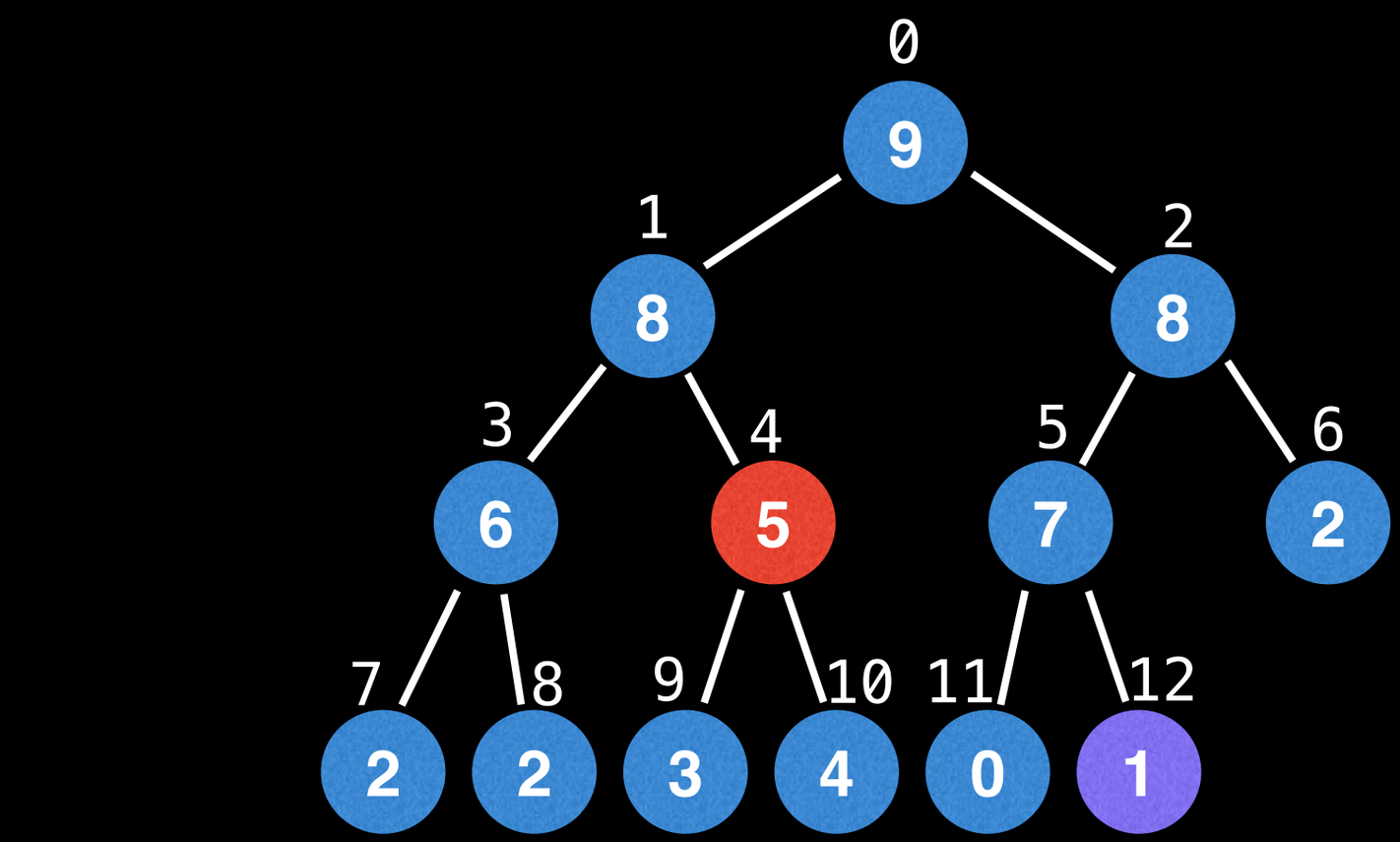
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Search for node with value 5.

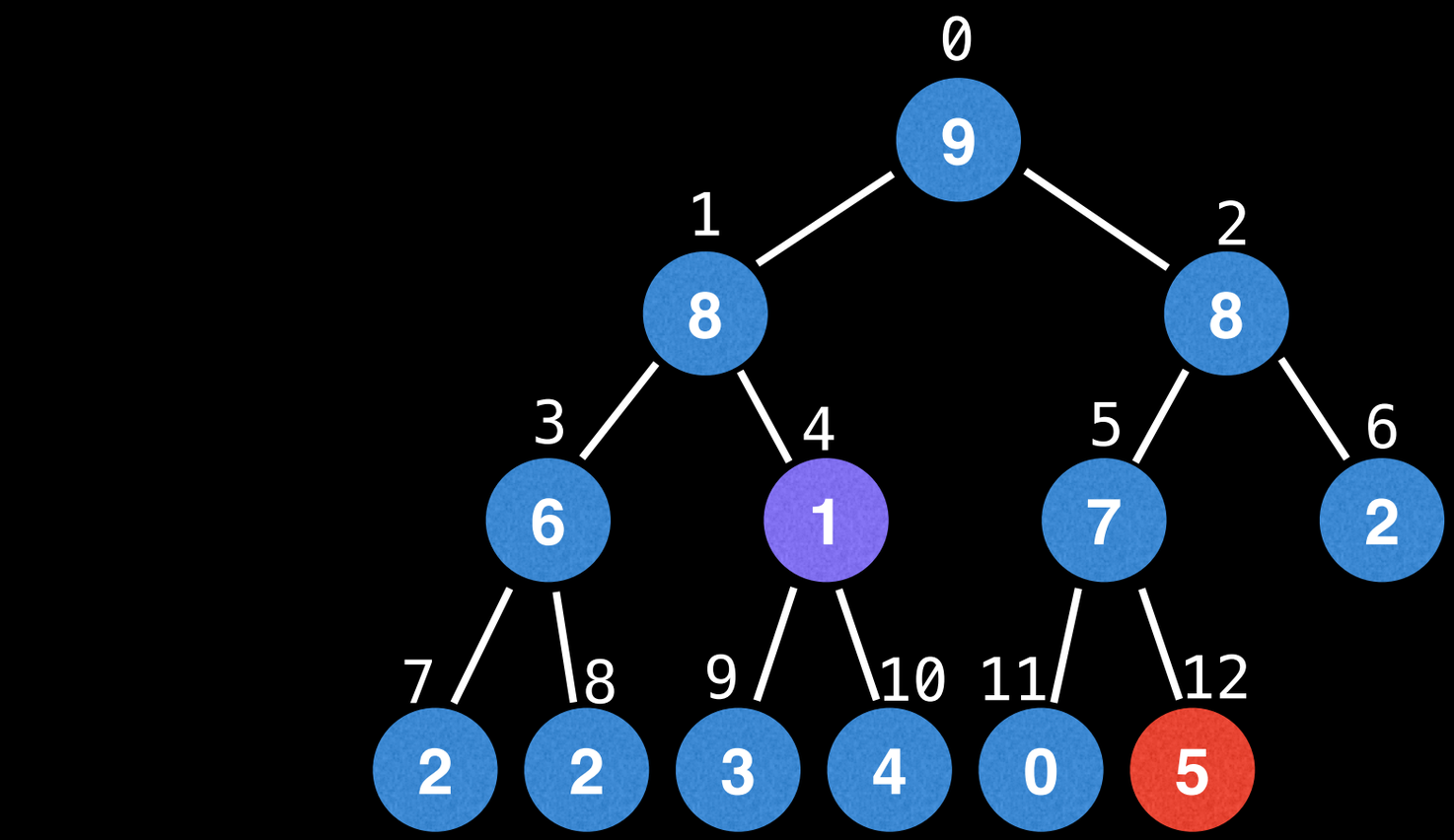
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	5	7	2	2	2	3	4	0	1	n/a	n/a

Swap node 5 with rightmost bottom node.

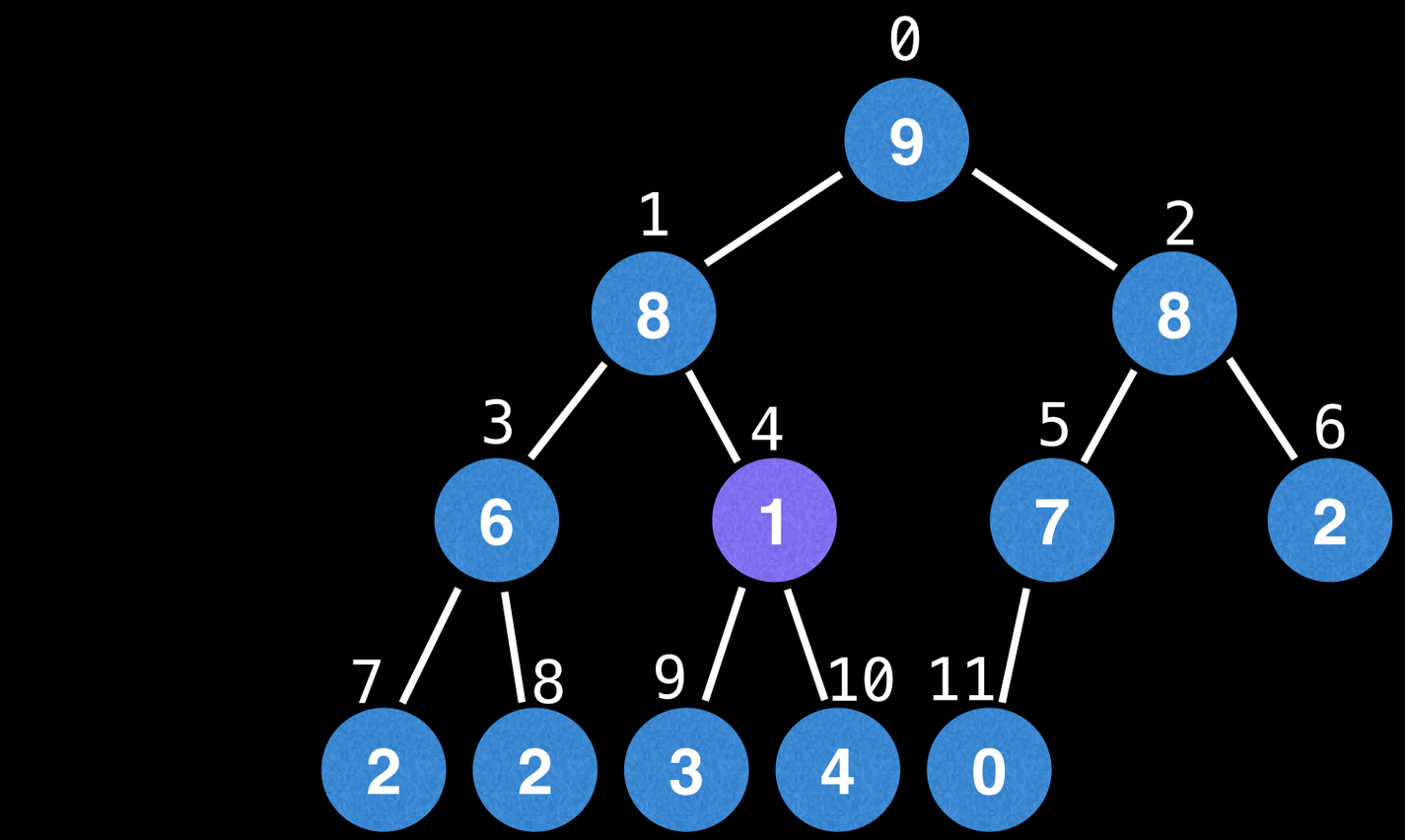
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	1	7	2	2	2	3	4	0	5	n/a	n/a

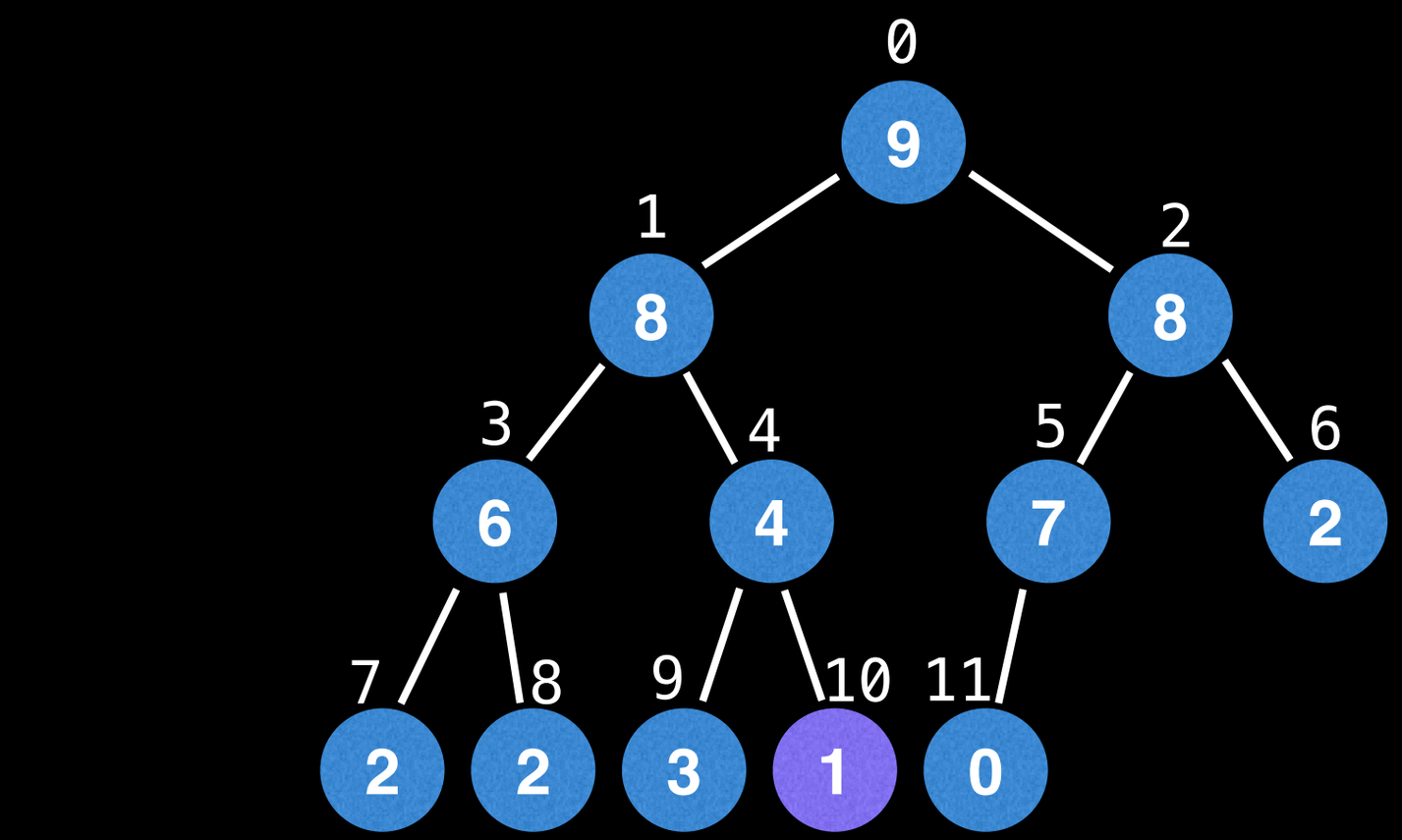
Remove node 5 from tree.

Refresher on the binary heap DS



Now the purple node we swapped may not satisfy the heap invariant so we need to either move it up or down the tree.

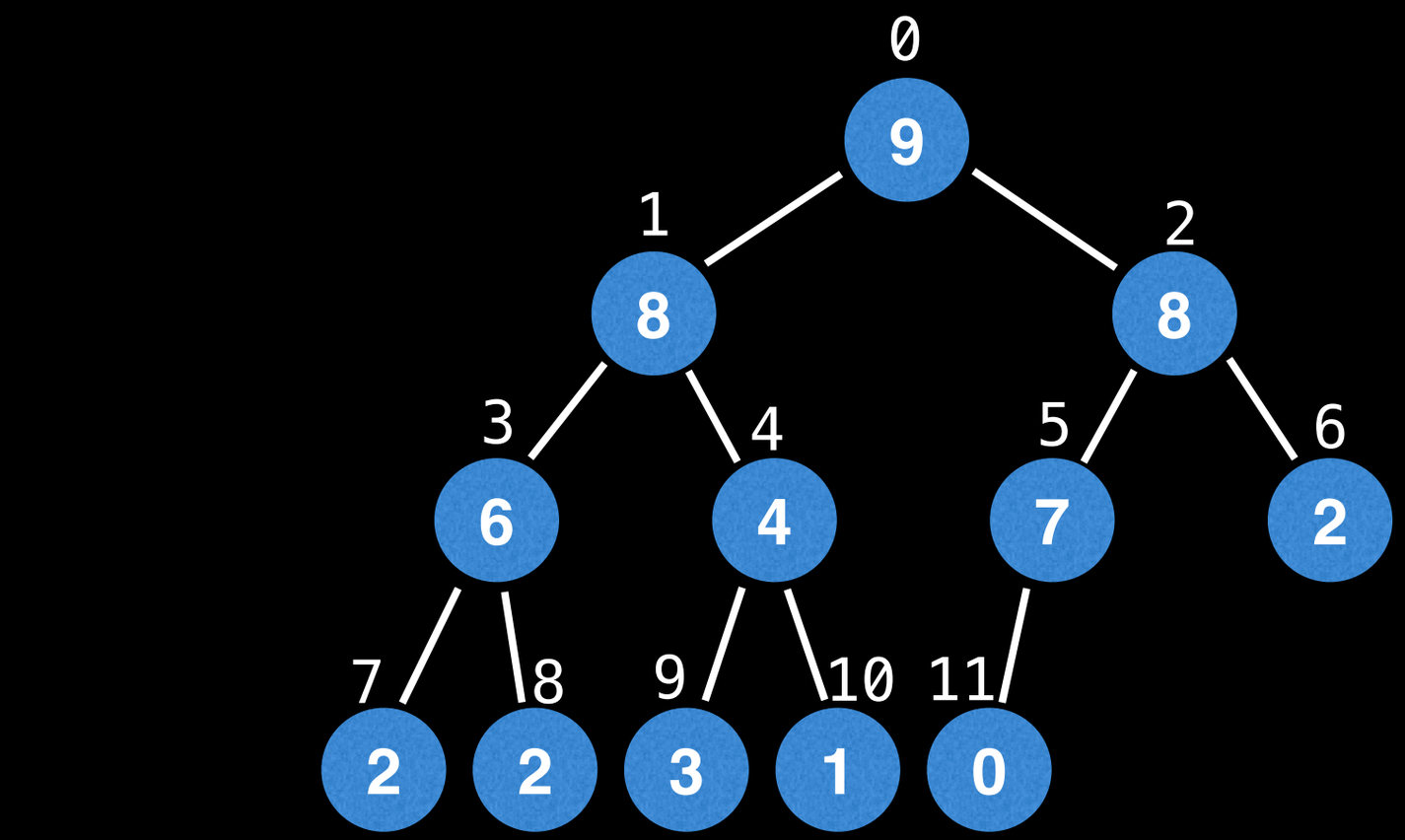
Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	4	7	2	2	2	3	1	0	n/a	n/a	n/a

Now the purple node we swapped may not satisfy the heap invariant so we need to either move it up or down the tree.

Refresher on the binary heap DS



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	8	6	4	7	2	2	2	3	1	0	n/a	n/a	n/a

Now the purple node we swapped may not satisfy the heap invariant so we need to either move it up or down the tree.

IPQ as binary heap

Suppose we have **N** people with different priorities we need to serve. Assume priorities can dynamically change and we always want to serve the person with the lowest priority.

Name	ki	Value
Anna		
Bella		
Carly		
Dylan		
Emily		
Fred		
George		
Henry		
Isaac		
James		
Kelly		
Laura		

IPQ as binary heap

To figure out who to serve next use a Min IPQ to sort by lowest value first.

Name	ki	Value
Anna		
Bella		
Carly		
Dylan		
Emily		
Fred		
George		
Henry		
Isaac		
James		
Kelly		
Laura		

IPQ as binary heap

To figure out who to serve next use a Min IPQ to sort by lowest value first.

Arbitrarily assign each person a unique index value between [0, N)

Name	ki	Value
Anna	0	
Bella	1	
Carly	2	
Dylan	3	
Emily	4	
Fred	5	
George	6	
Henry	7	
Isaac	8	
James	9	
Kelly	10	
Laura	11	

Key Index

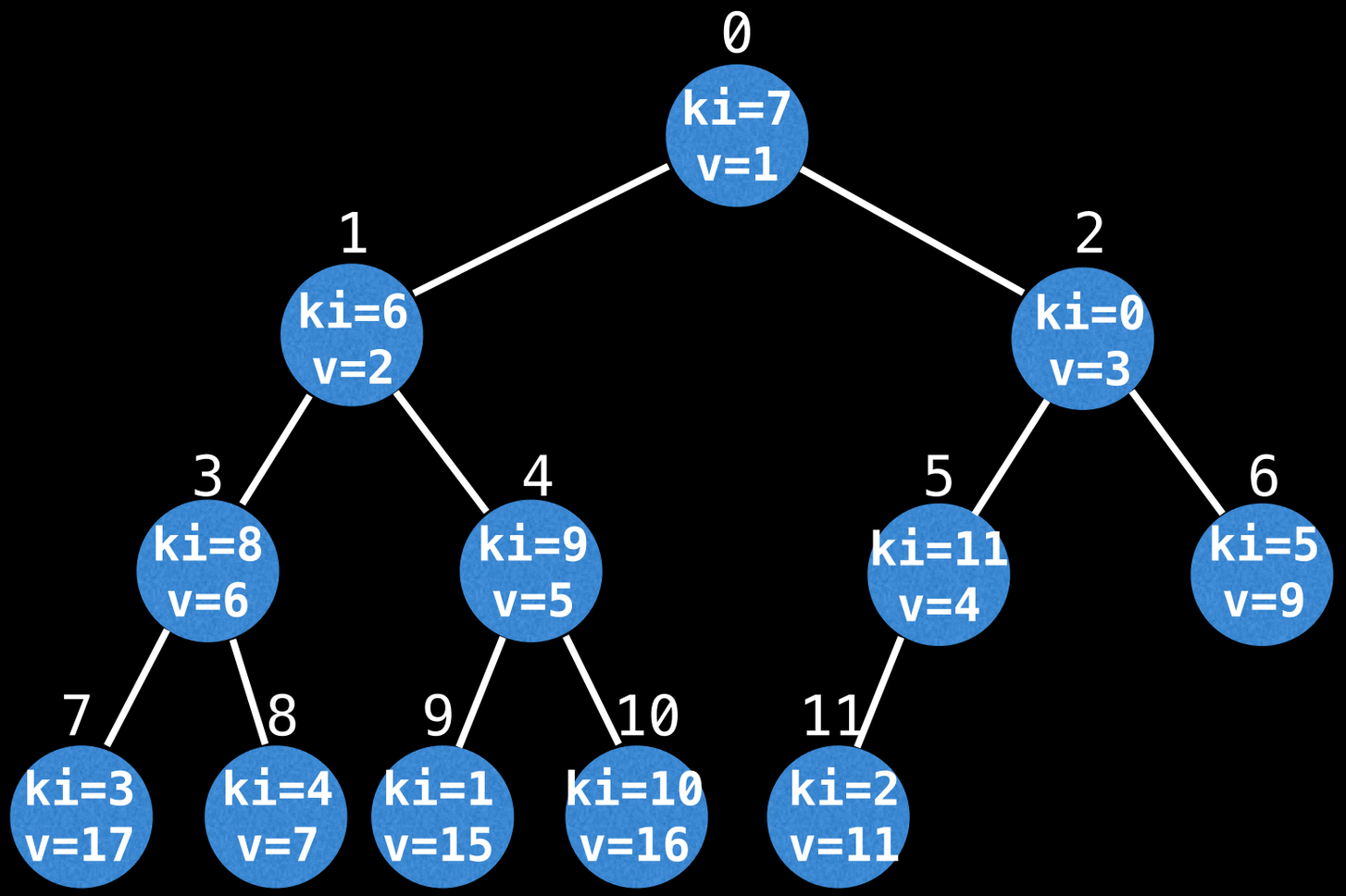
IPQ as binary heap

To figure out who to serve next use a Min IPQ to sort by lowest value first.

Name	ki	Value
Anna	0	3
Bella	1	15
Carly	2	11
Dylan	3	17
Emily	4	7
Fred	5	9
George	6	2
Henry	7	1
Isaac	8	6
James	9	5
Kelly	10	16
Laura	11	4

Initial values to place inside IPQ. These will be maintained by the IPQ once inserted. Note that values can be any comparable value not only integers.

IPQ as a binary heap

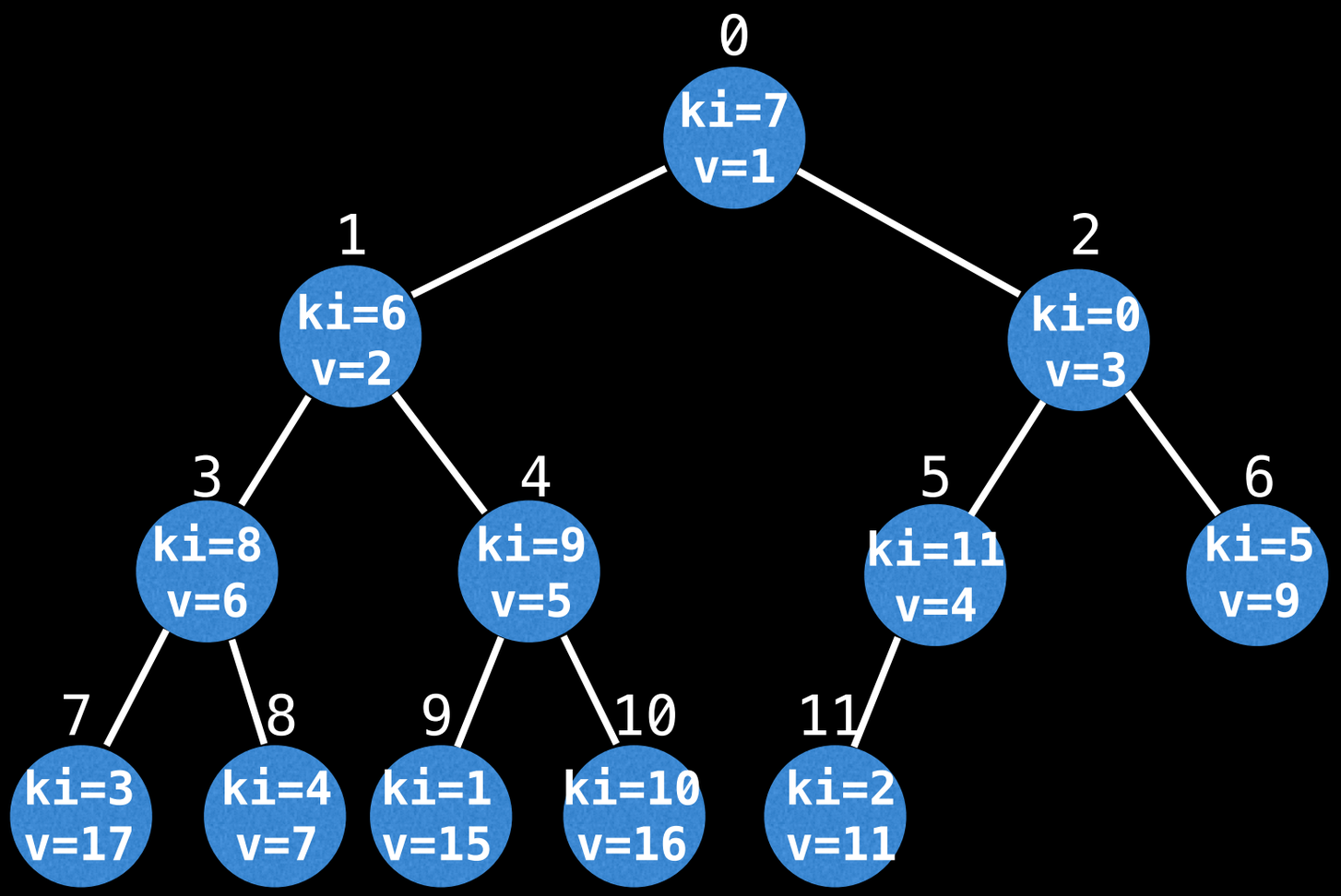


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

When we insert (ki, v) pairs into an IPQ we sort by the value associated with each key.

In the heap above we are sorting by smallest value since we're working with a min heap.

IPQ as a binary heap



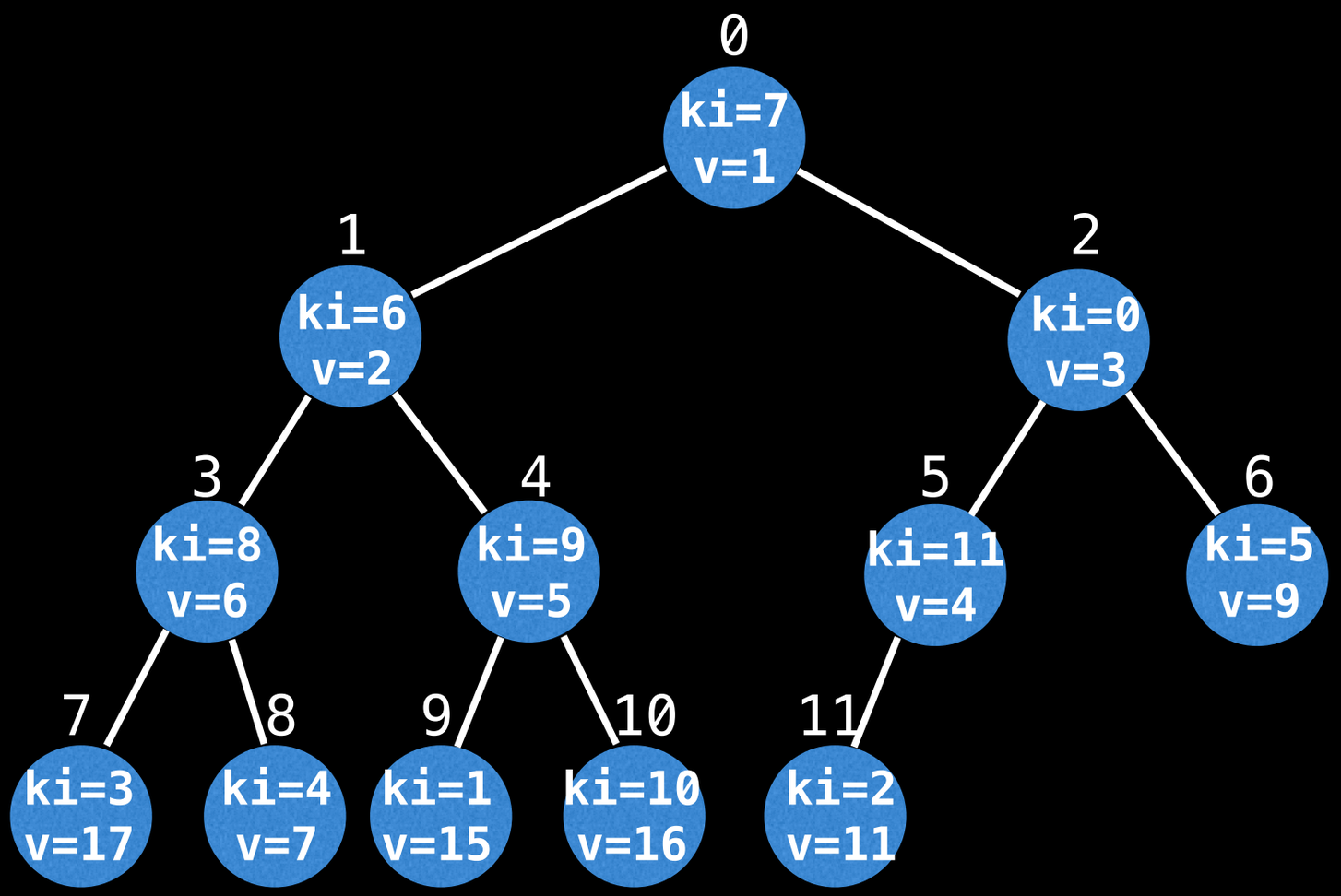
Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

vals

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1

To access the value for any given key k, find its key index (ki) and do a lookup in the **vals** array maintained by the IPQ.

IPQ as a binary heap

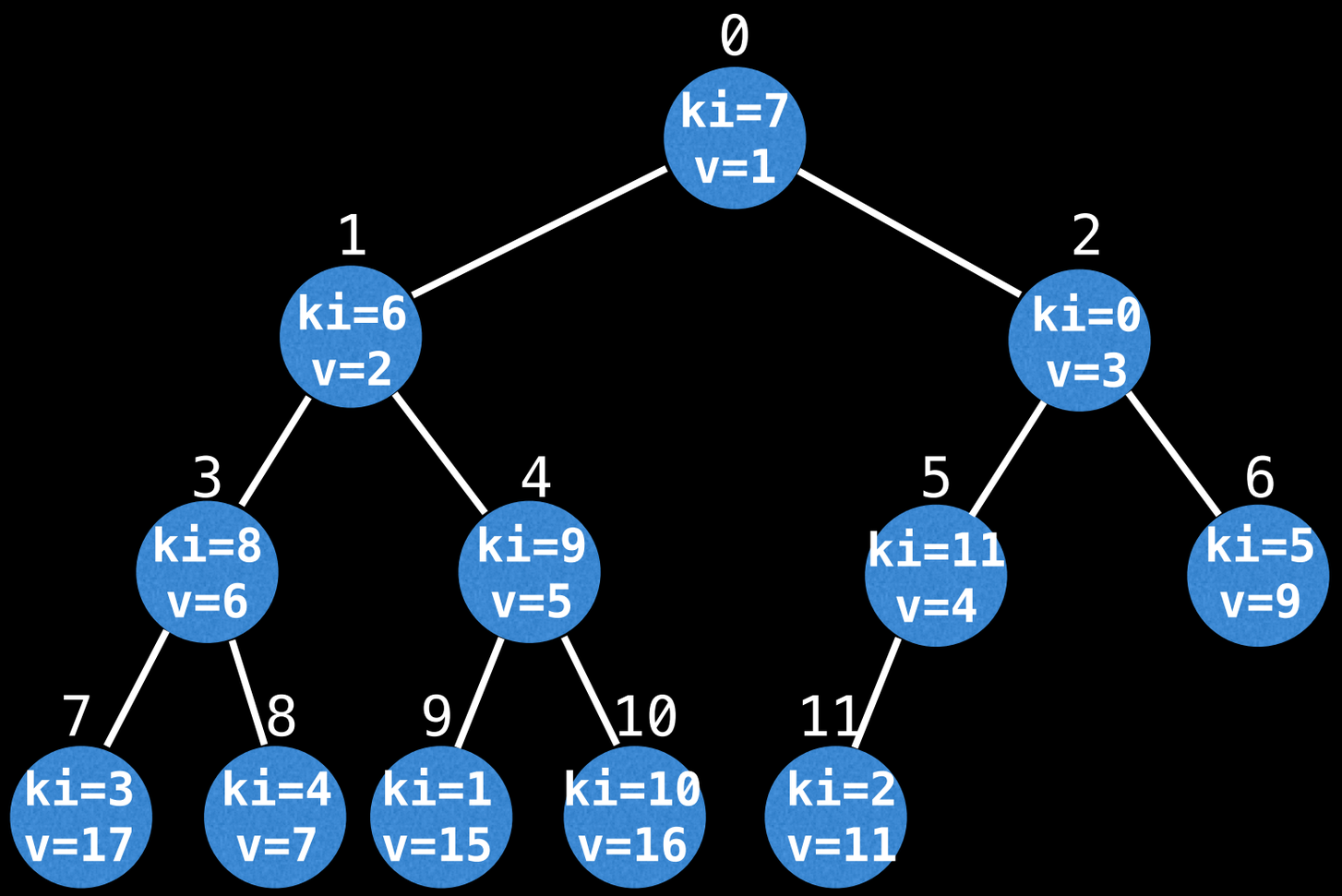


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1

Q: What value does “Bella” have in the IPQ?

IPQ as a binary heap

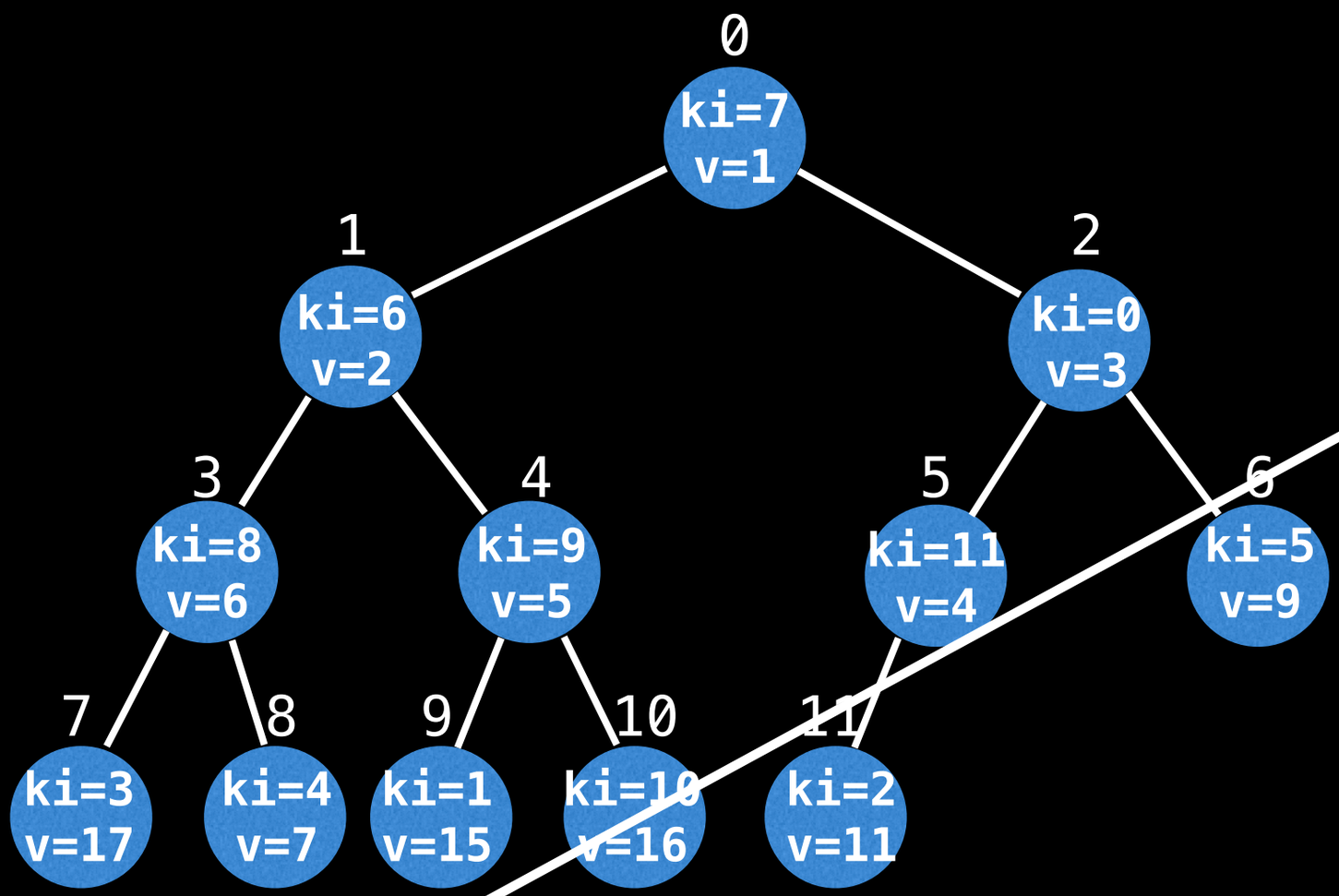


Name	ki
Anna	0
Bella →	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1

Q: What value does “Bella” have in the IPQ?

IPQ as a binary heap

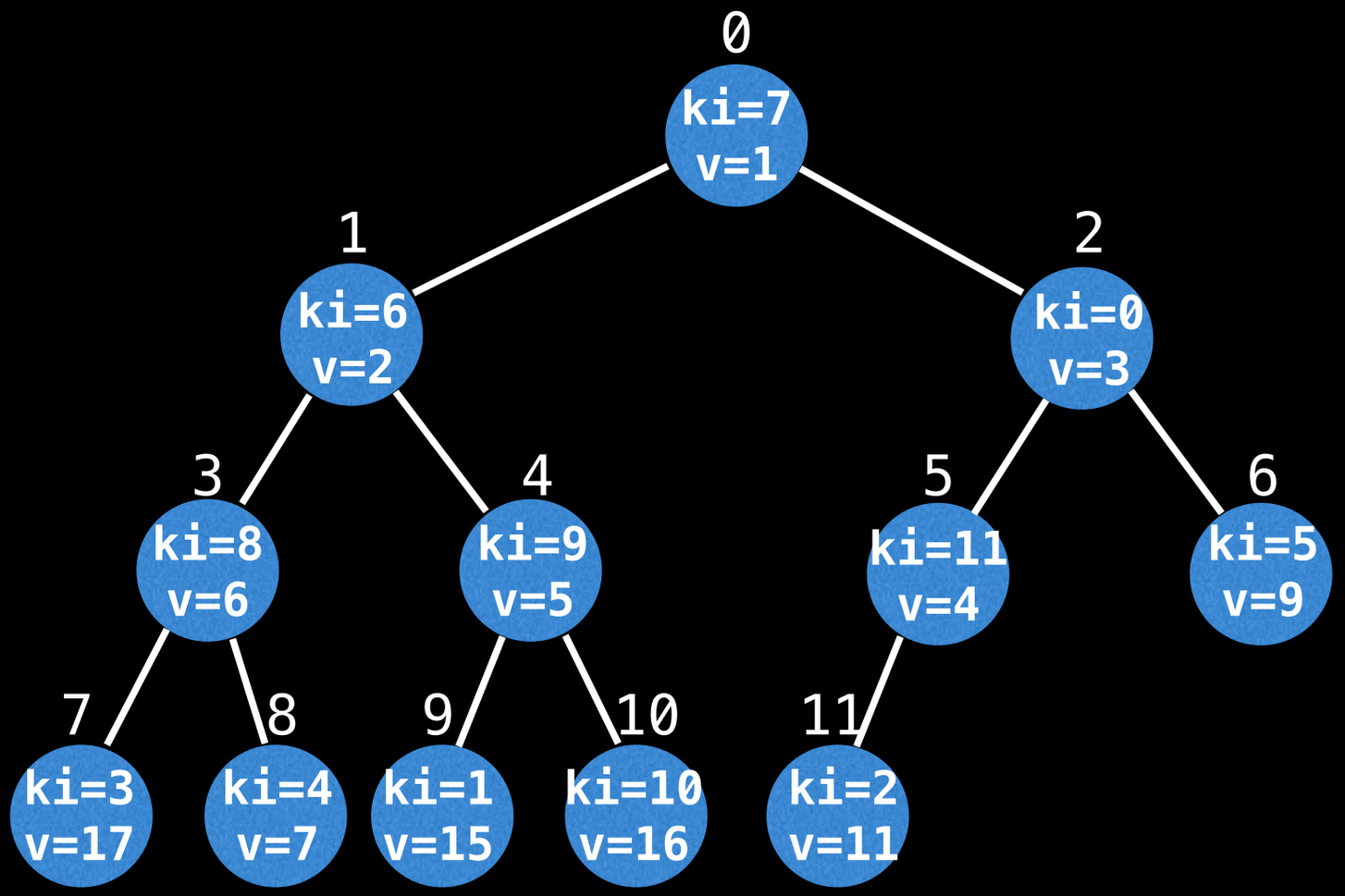


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1

ki = 1, so “Bella” has a value of vals[ki] = 15

IPQ as a binary heap

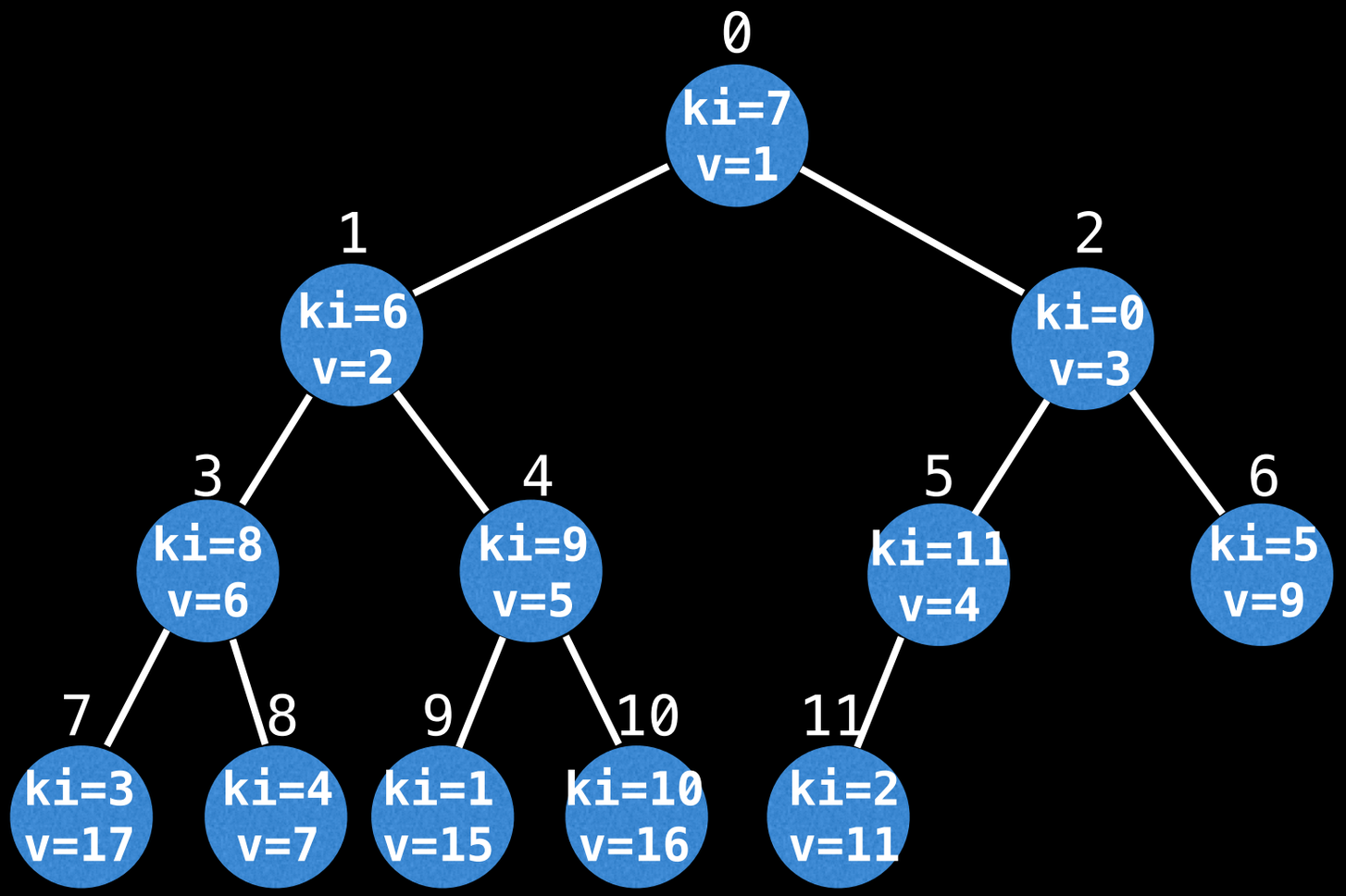


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1

ki = 1, so “Bella” has a value of vals[ki] = 15

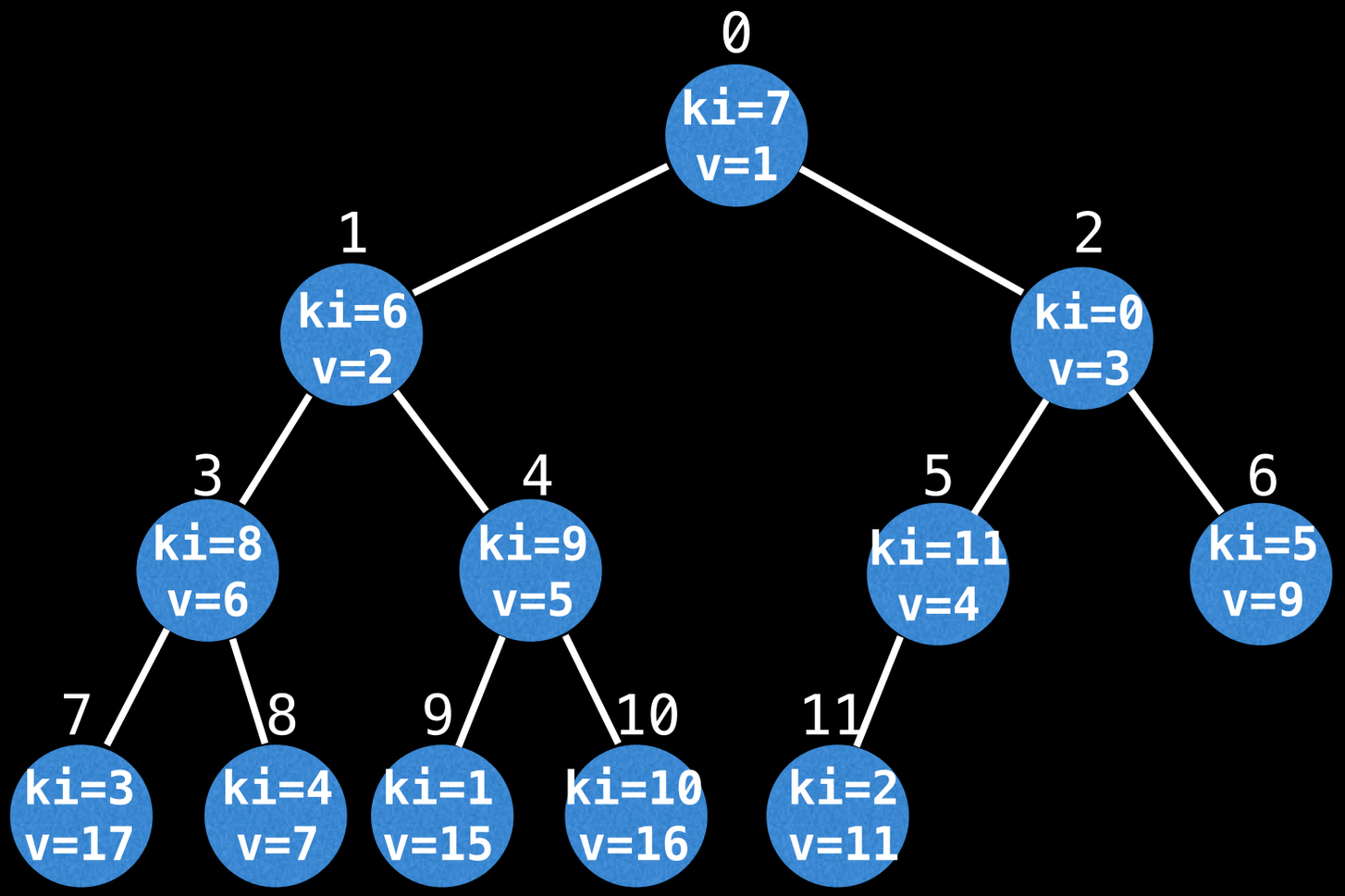
IPQ as a binary heap



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

Q: How do I find the index of the node for a particular key?

IPQ as a binary heap

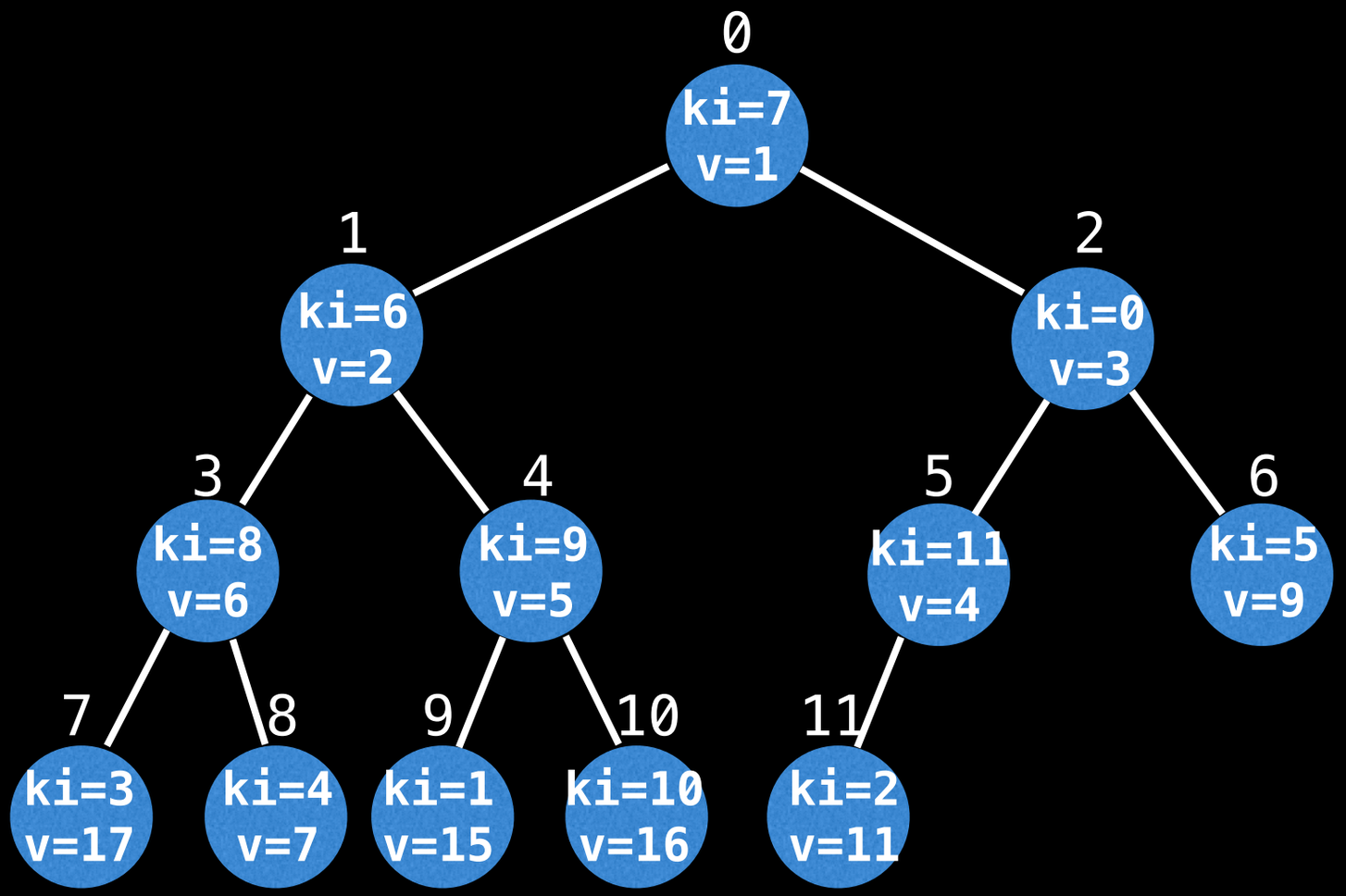


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

The array **pm** is a **Position Map** we maintain to tell us the index of the node in the heap for a given key index (*ki*).

IPQ as a binary heap

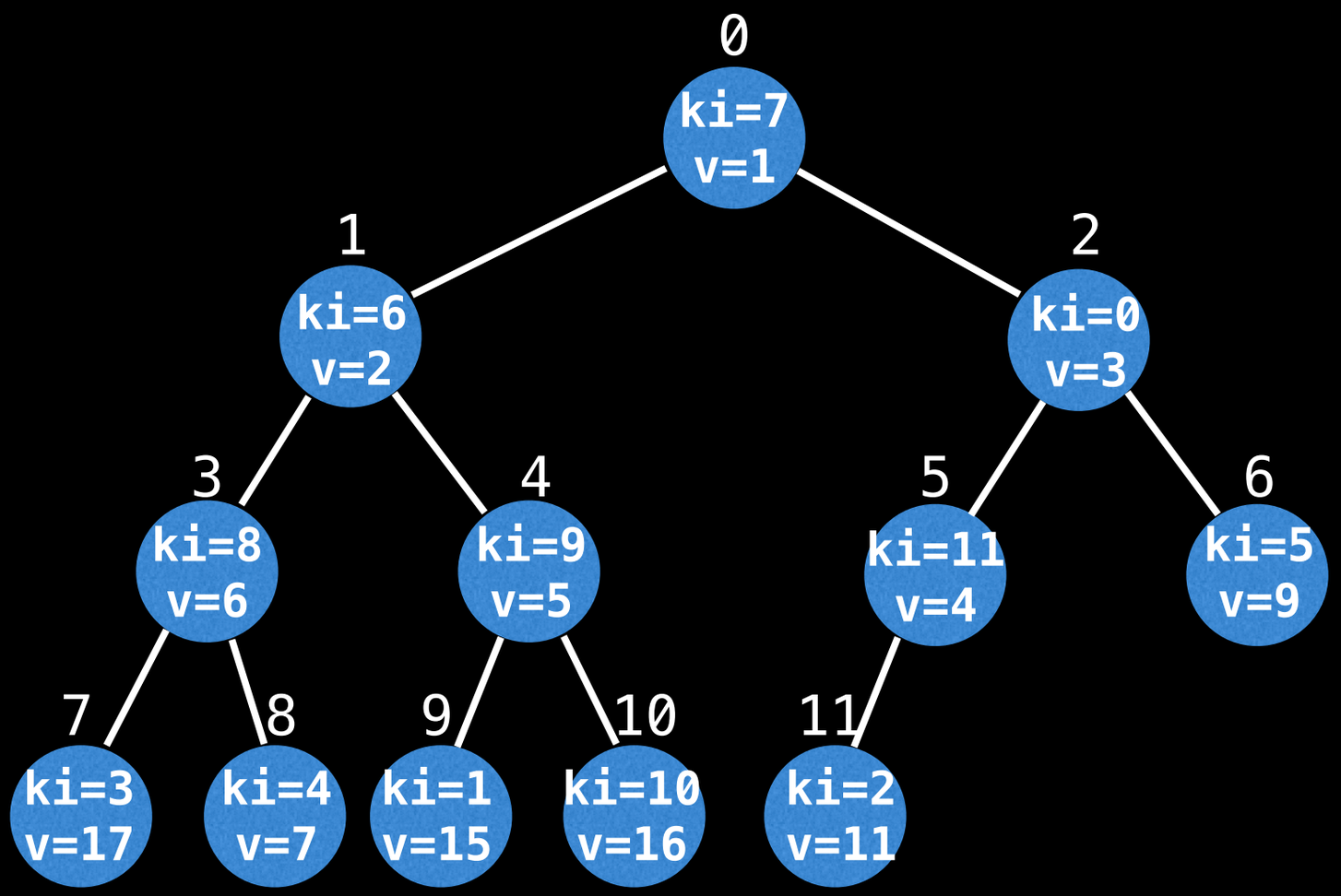


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

Q: Which node represents the key “Dylan”?

IPQ as a binary heap

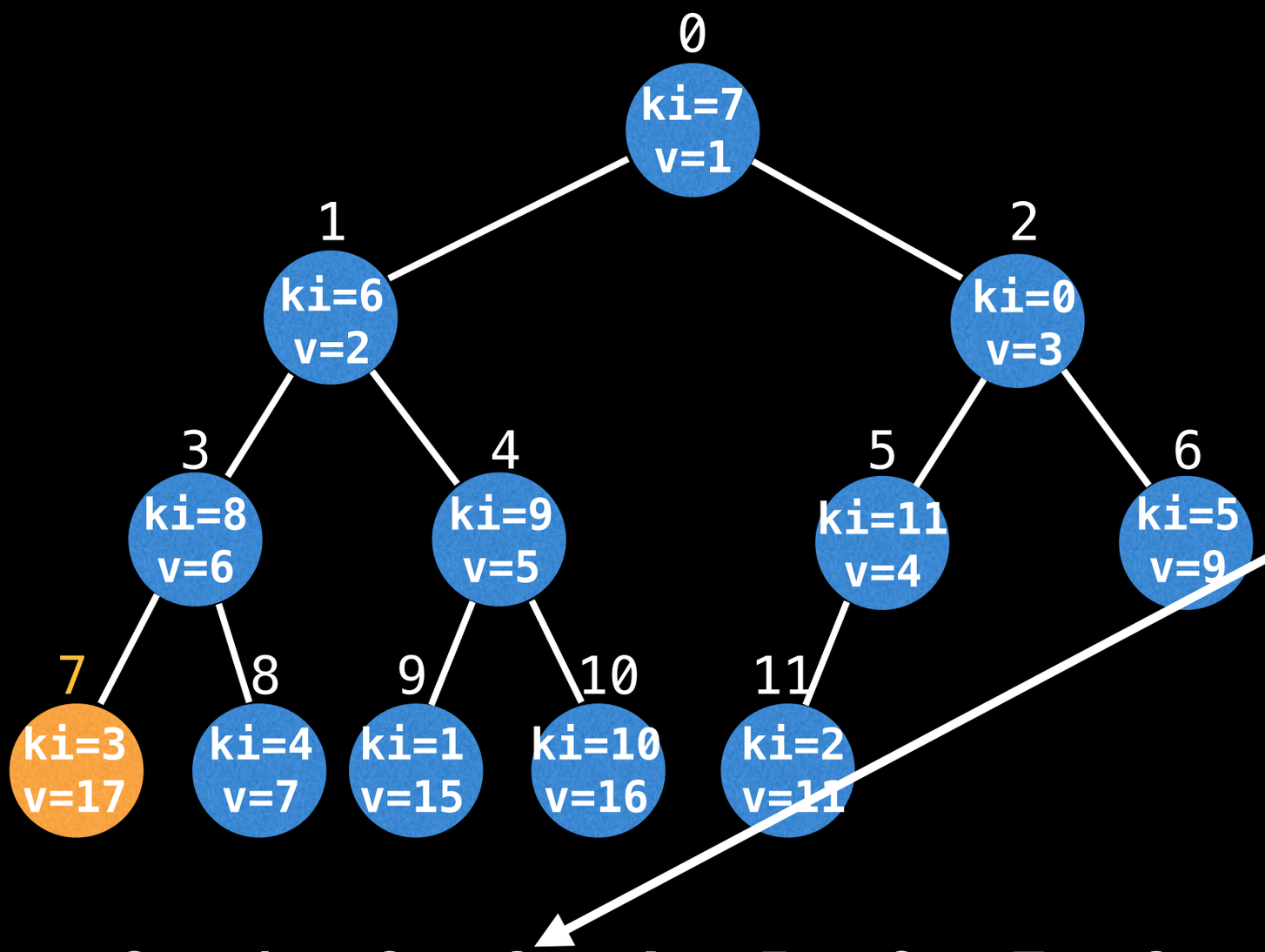


Name	ki
Anna	0
Bella	1
Carly	2
Dylan →	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

First find the key-index for “Dylan”

IPQ as a binary heap

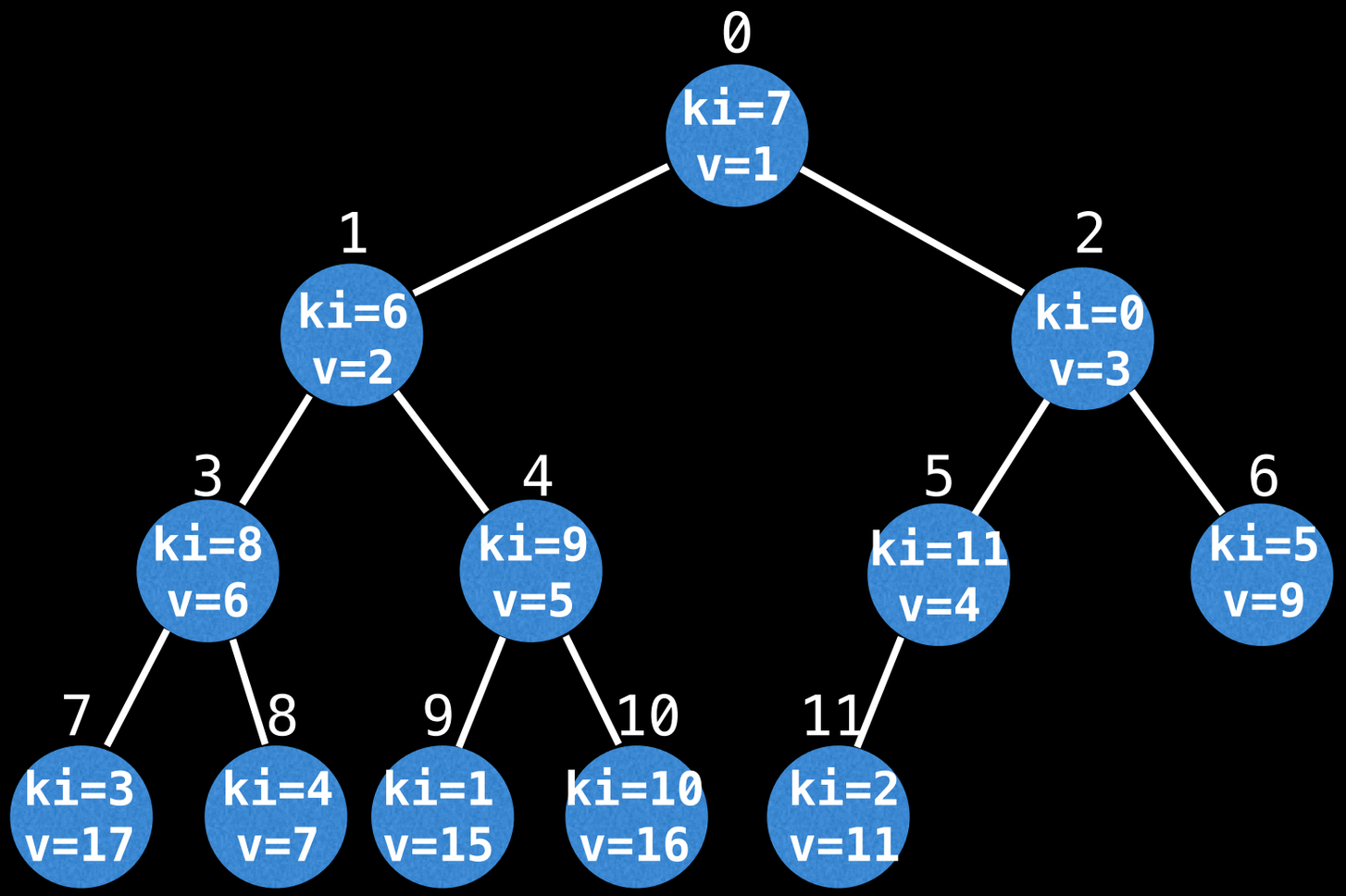


Name	ki
Anna	0
Bella	1
Carly	2
Dylan →	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

"Dylan" has a key index of 3, so pm[3] = 7 is where Dylan is in the heap.

IPQ as a binary heap

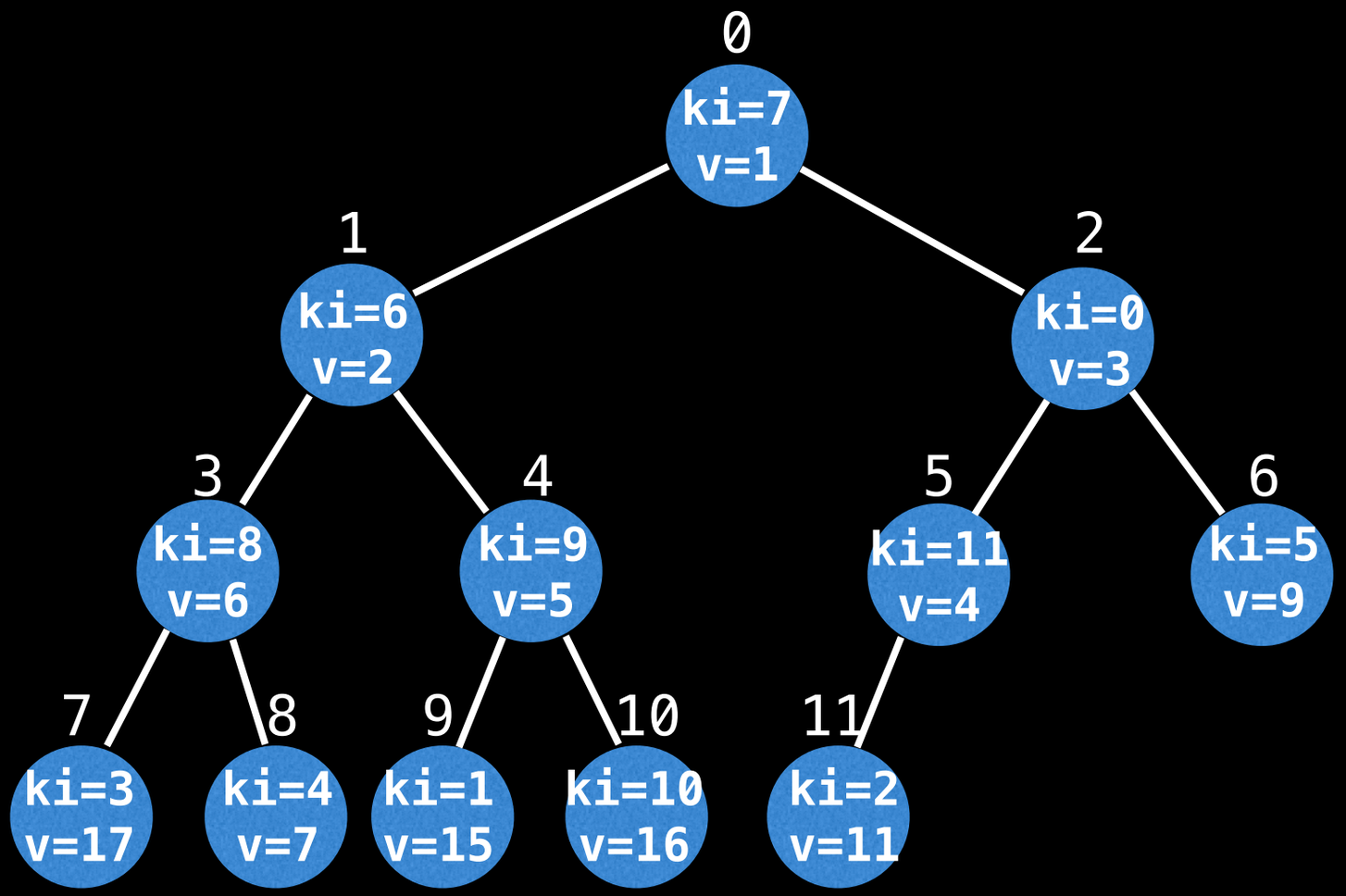


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

Q: Where is “George” in the heap?

IPQ as a binary heap

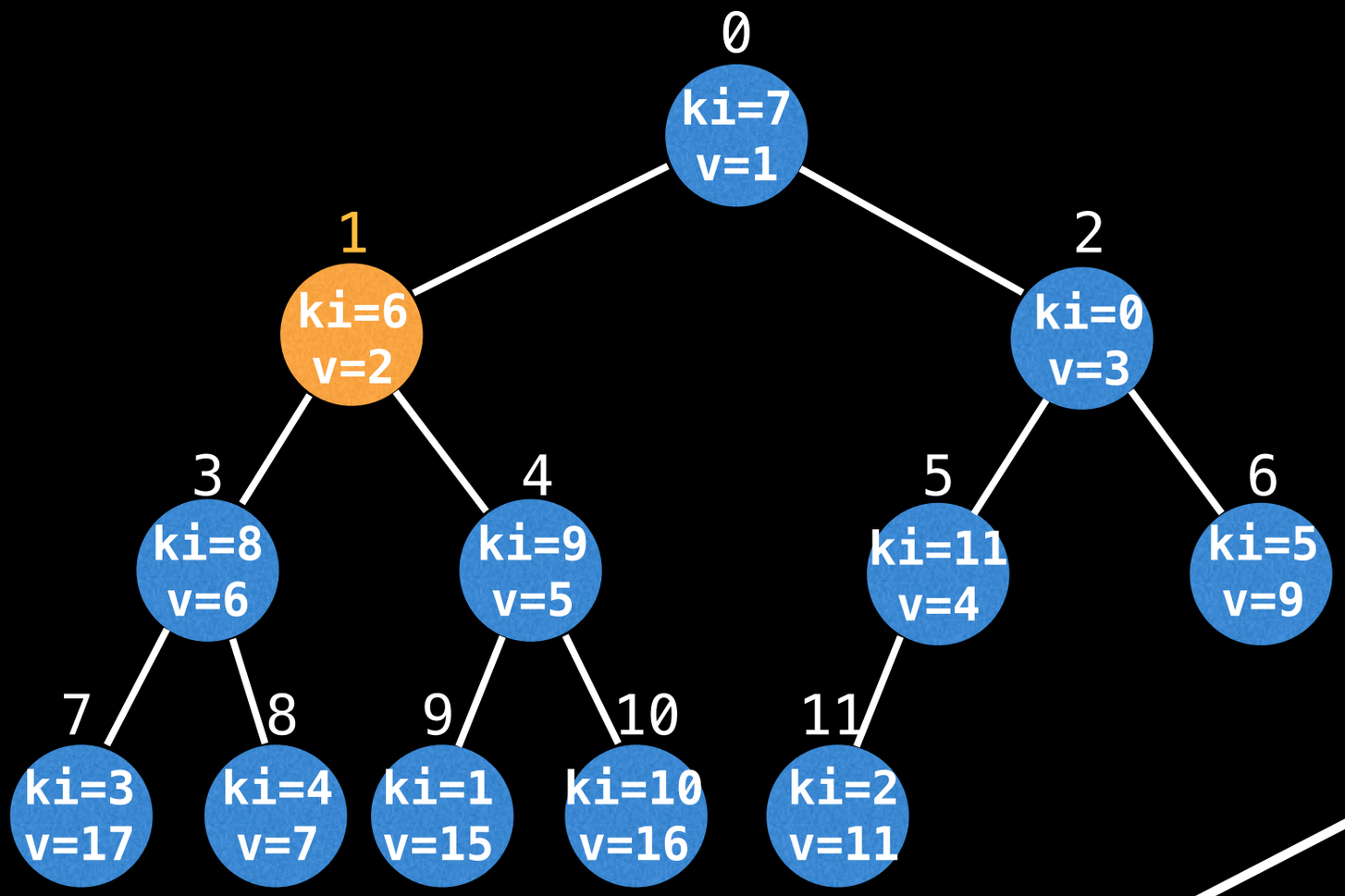


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George →	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

Q: Where is “George” in the heap?

IPQ as a binary heap

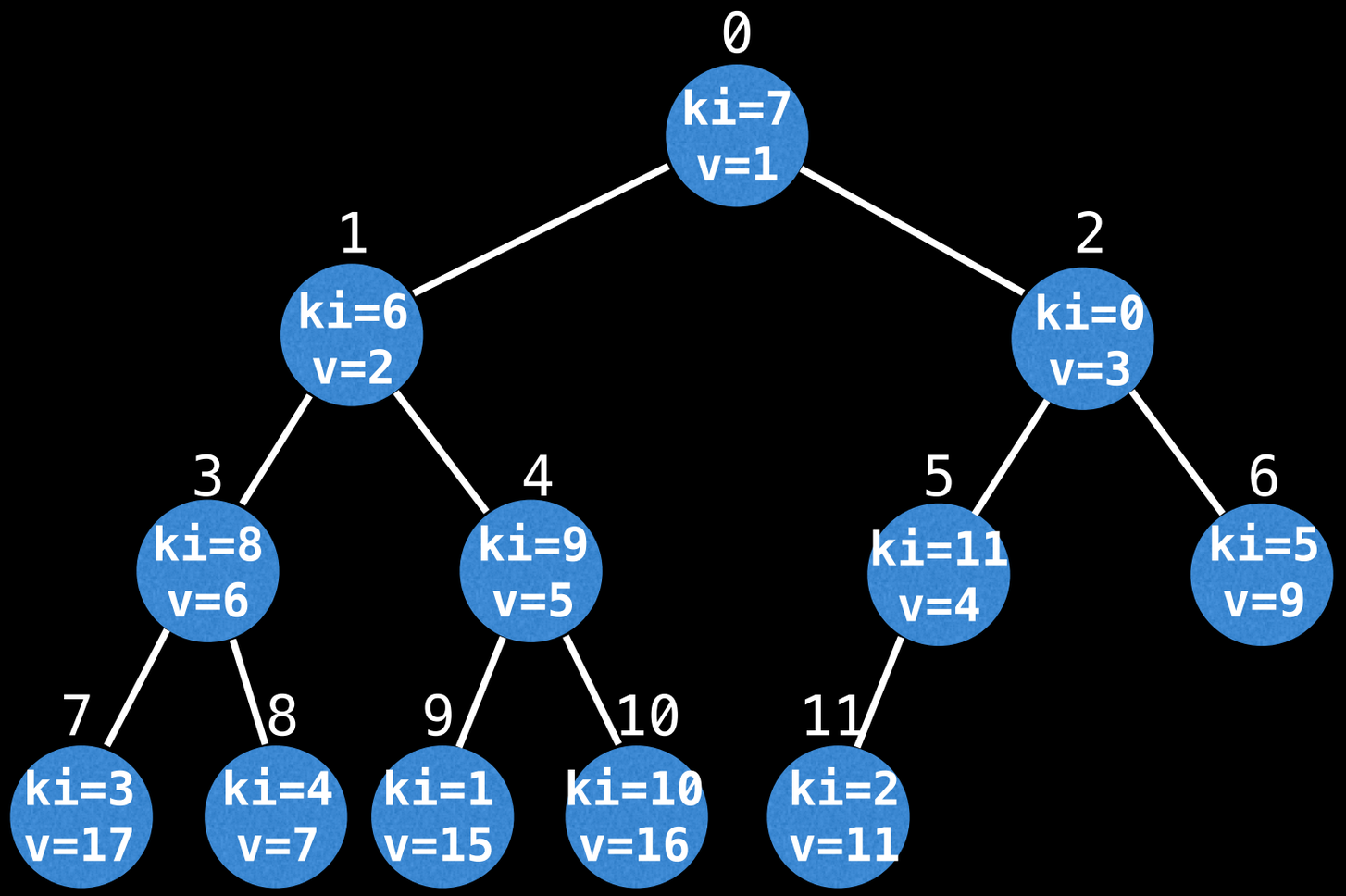


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

Q: Where is “George” in the heap?

IPQ as a binary heap

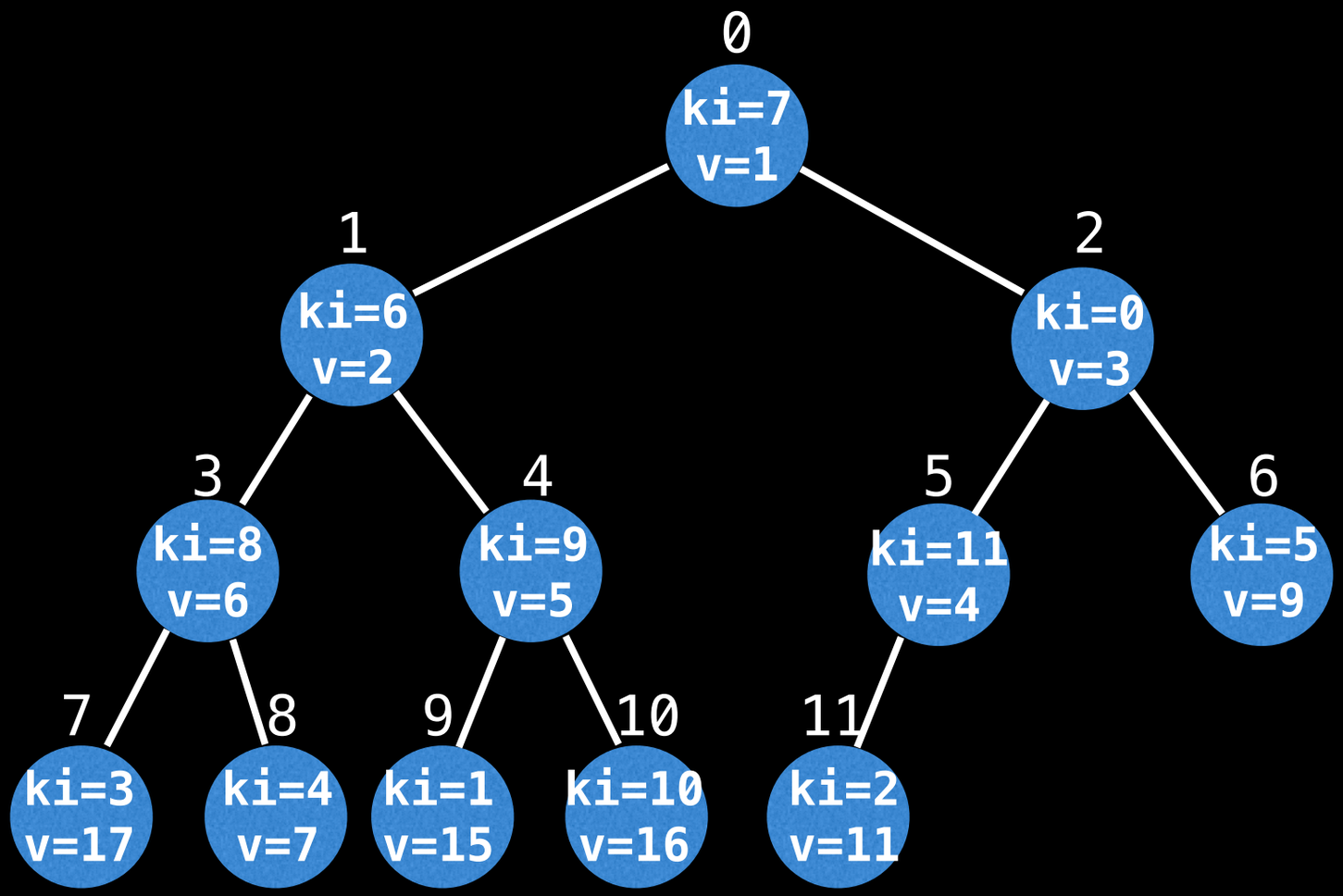


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1

Q: How do we go from knowing the position of a node to its key and ki value?

IPQ as a binary heap

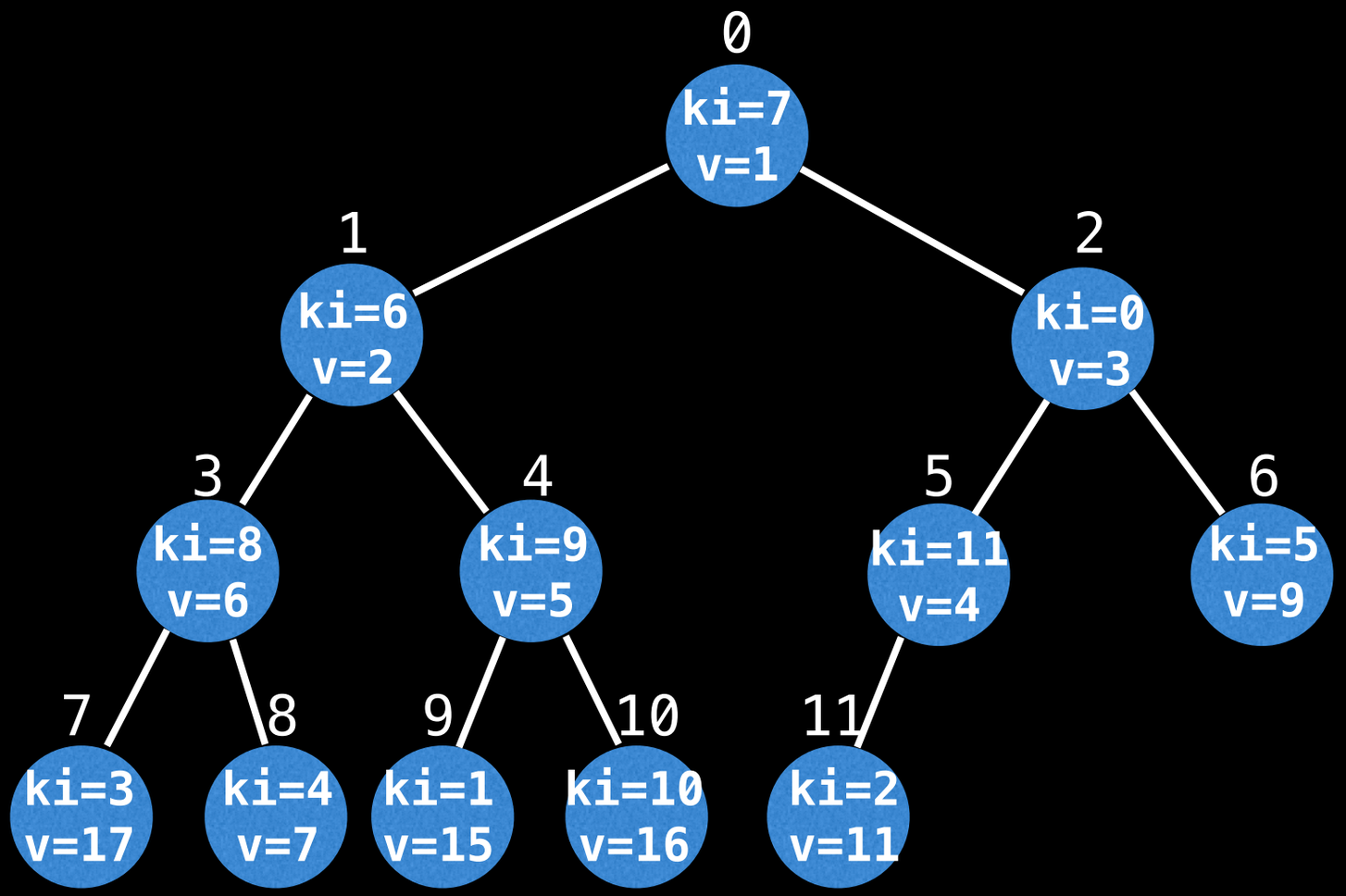


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

To do that we also need to maintain an inverse lookup table denoted: **im (Inverse Map)**

IPQ as a binary heap

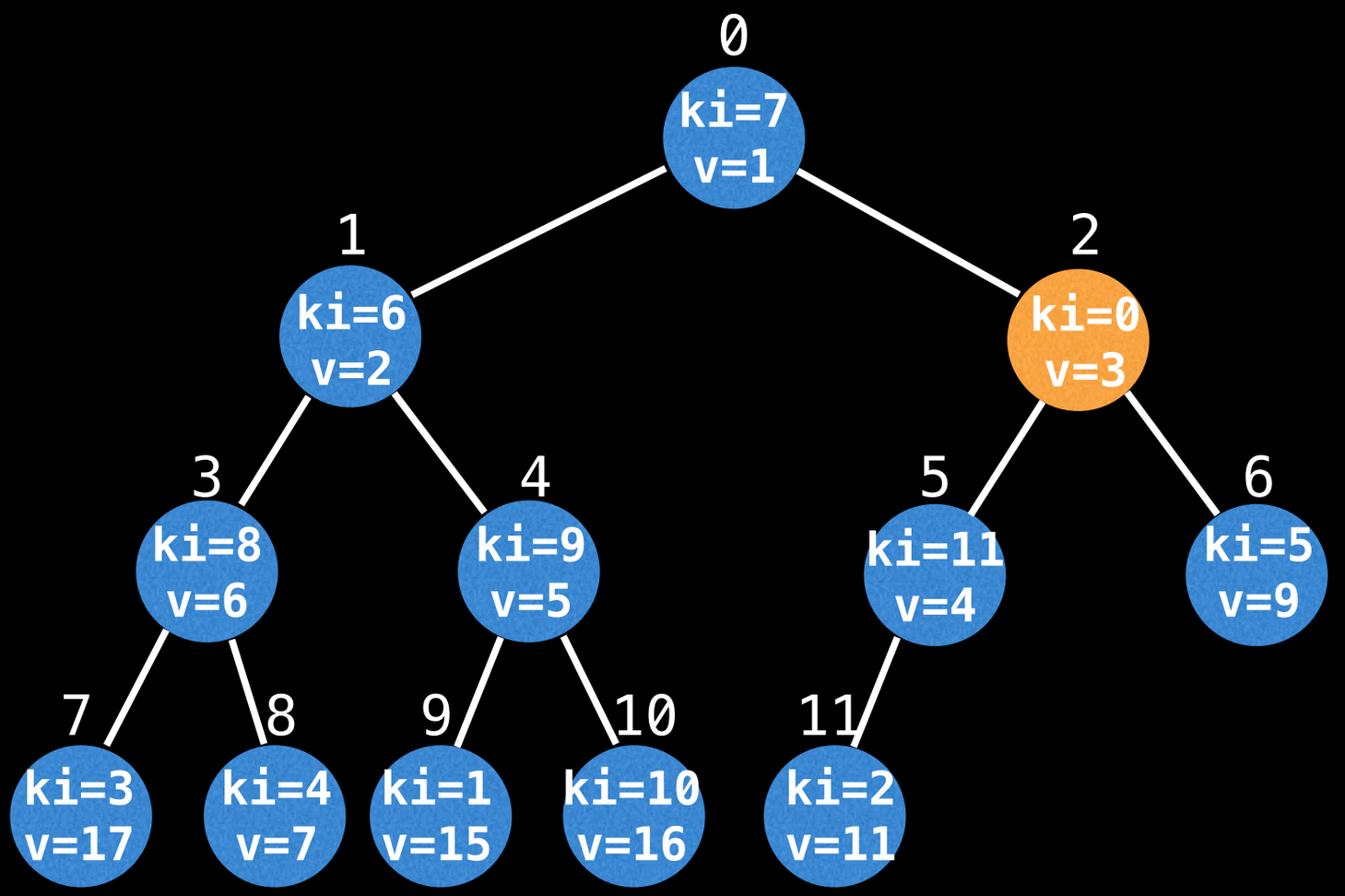


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Q: Which person (key) is represented in the node at index 2?

IPQ as a binary heap

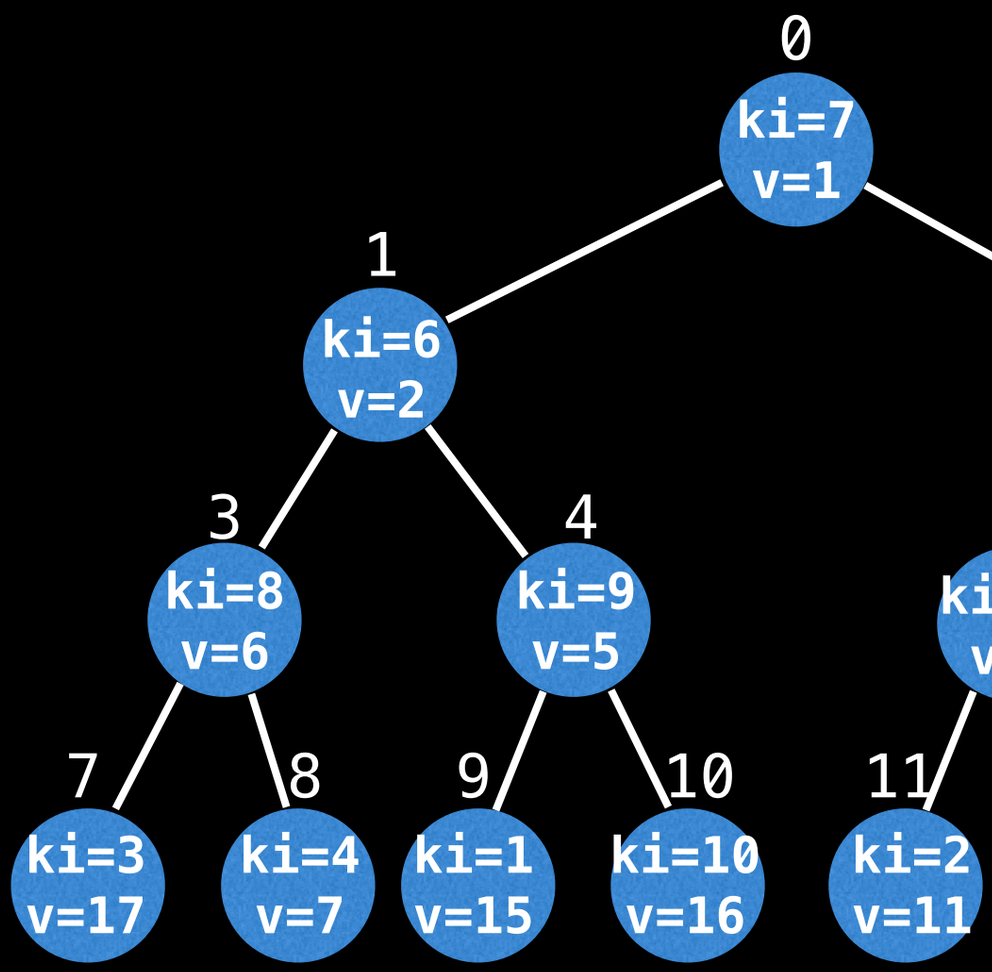


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

As an example, we can figure out the person represented in the node at index 2

IPQ as a binary heap

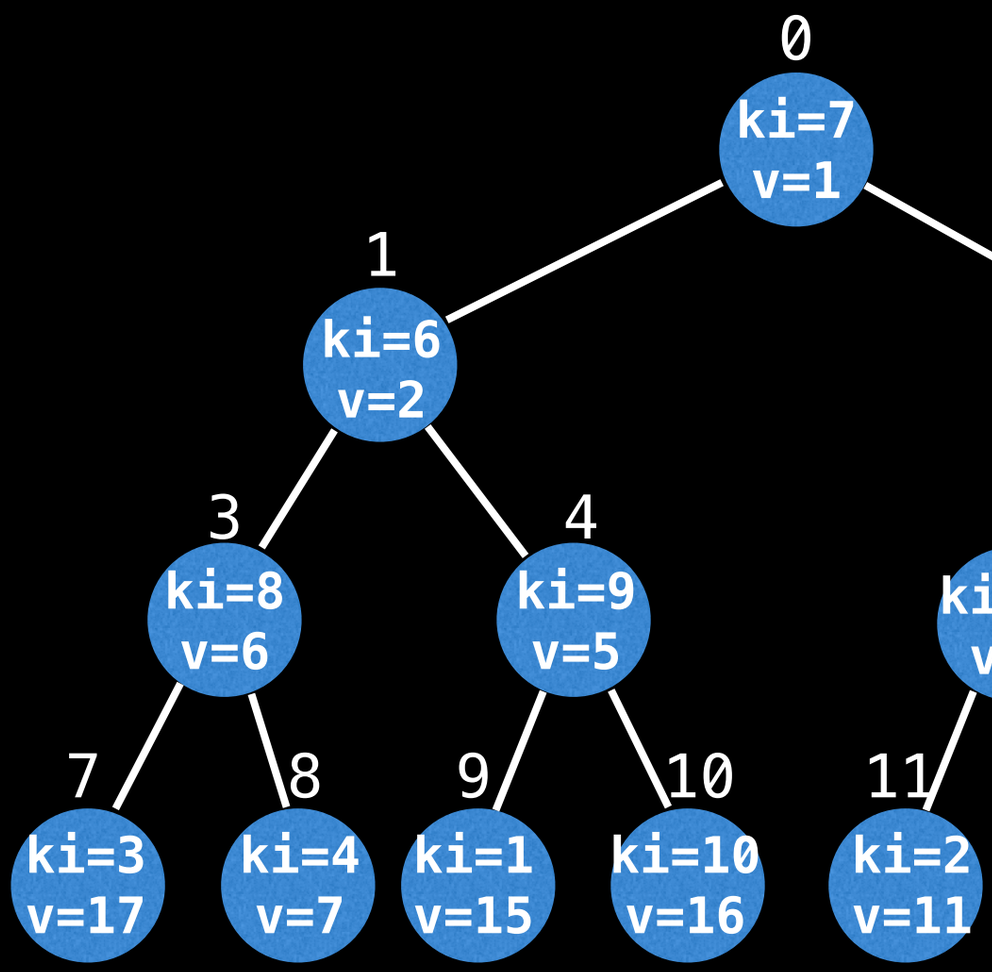


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

ki = im[2] = 0 tells you the key index (ki) for that node.

IPQ as a binary heap

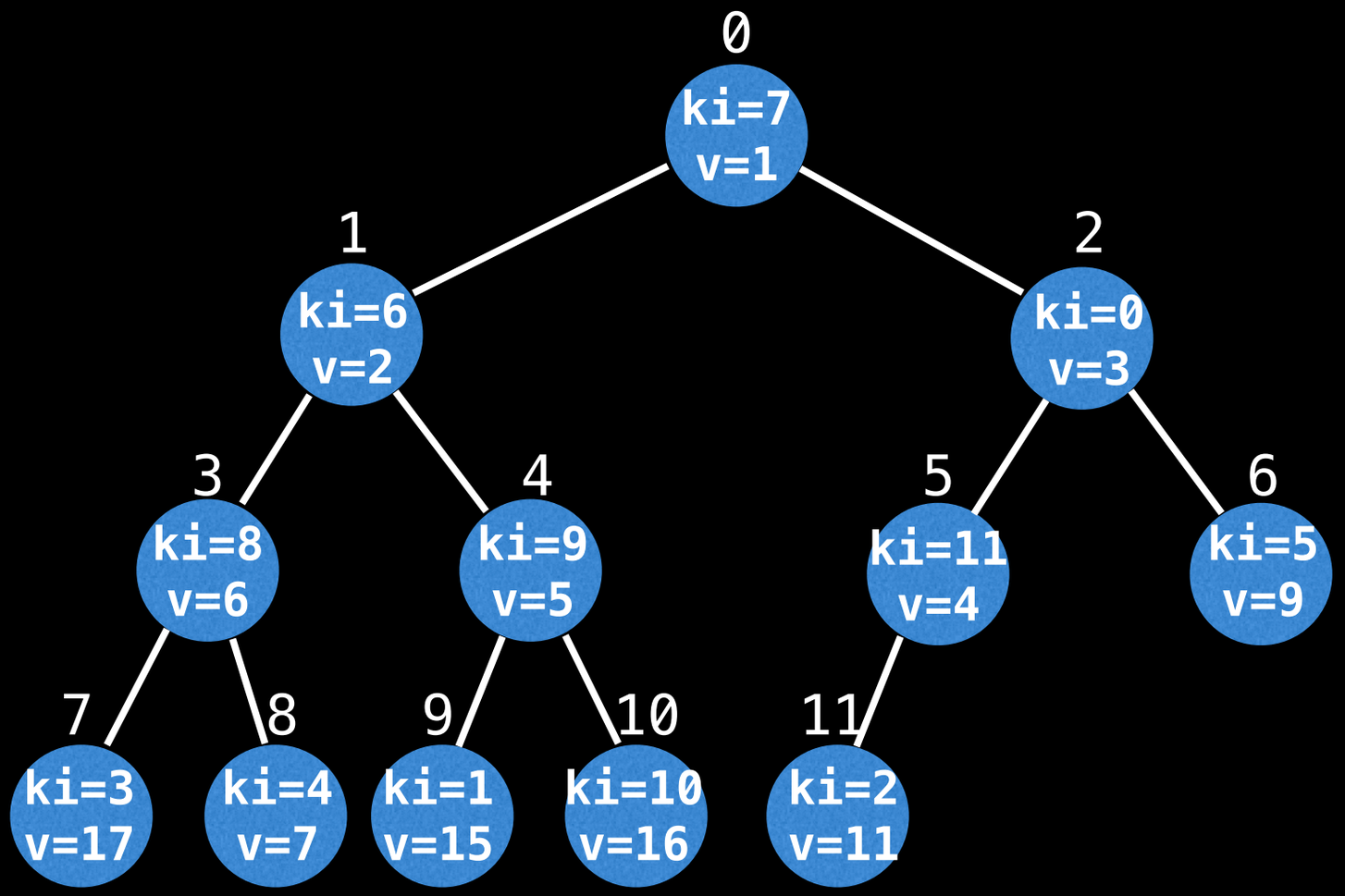


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

You can then use the ki value with the bidirectional hashtable to retrieve the actual key “Anna”

IPQ as a binary heap

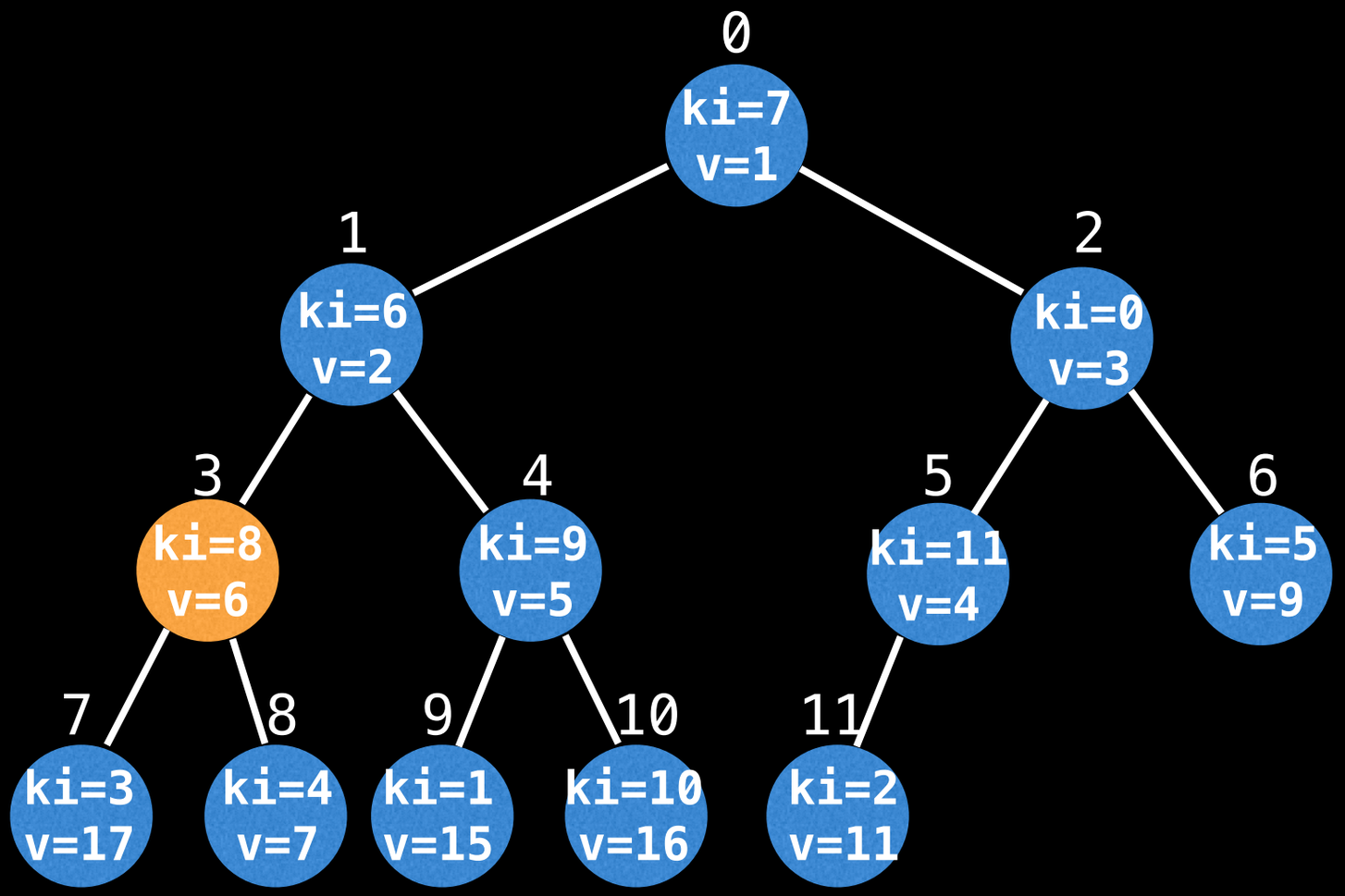


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Q: Which person (key) is being represented in the node at index position 3?

IPQ as a binary heap

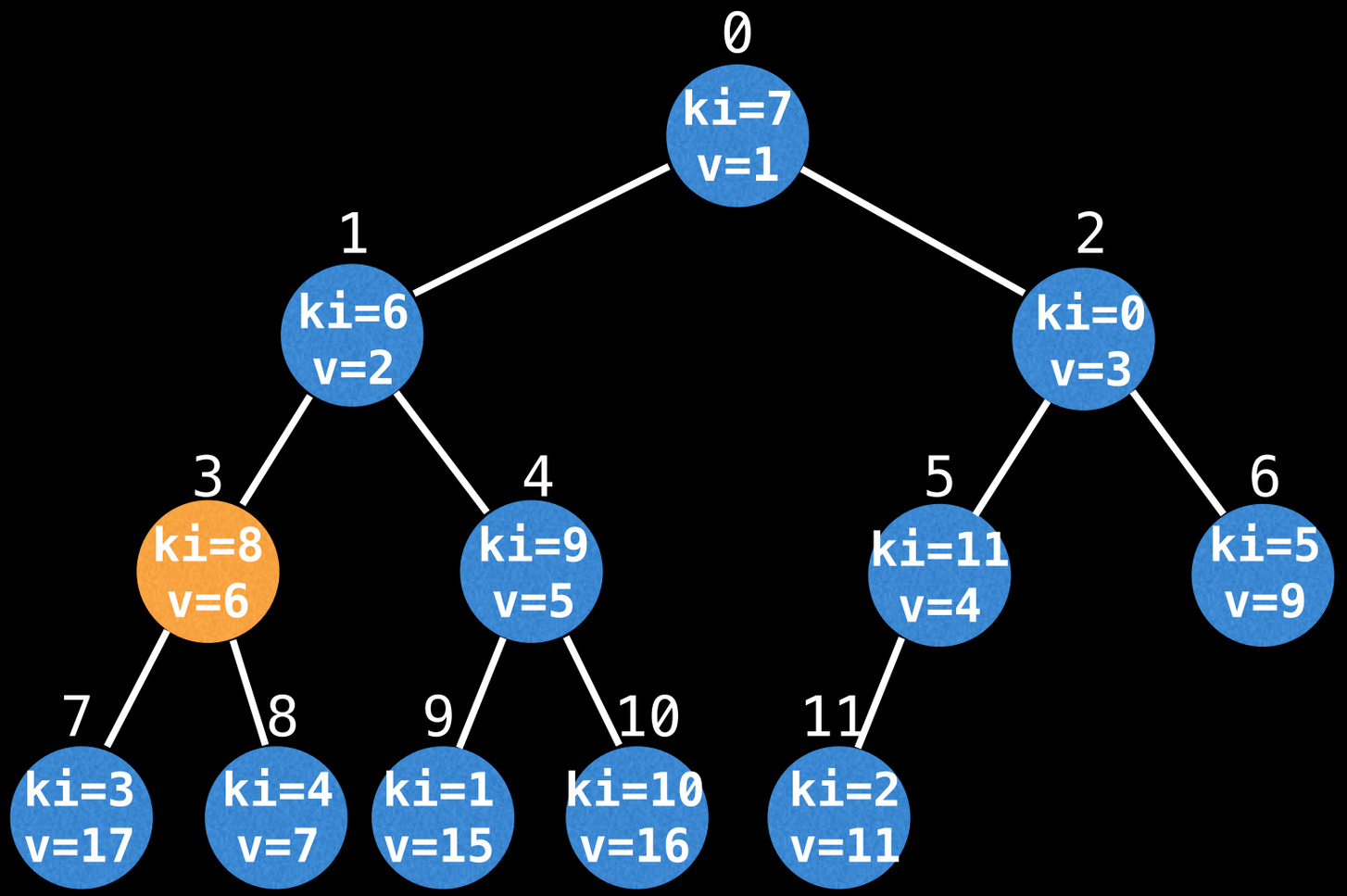


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Q: Which person (key) is being represented in the node at index position 3?

IPQ as a binary heap

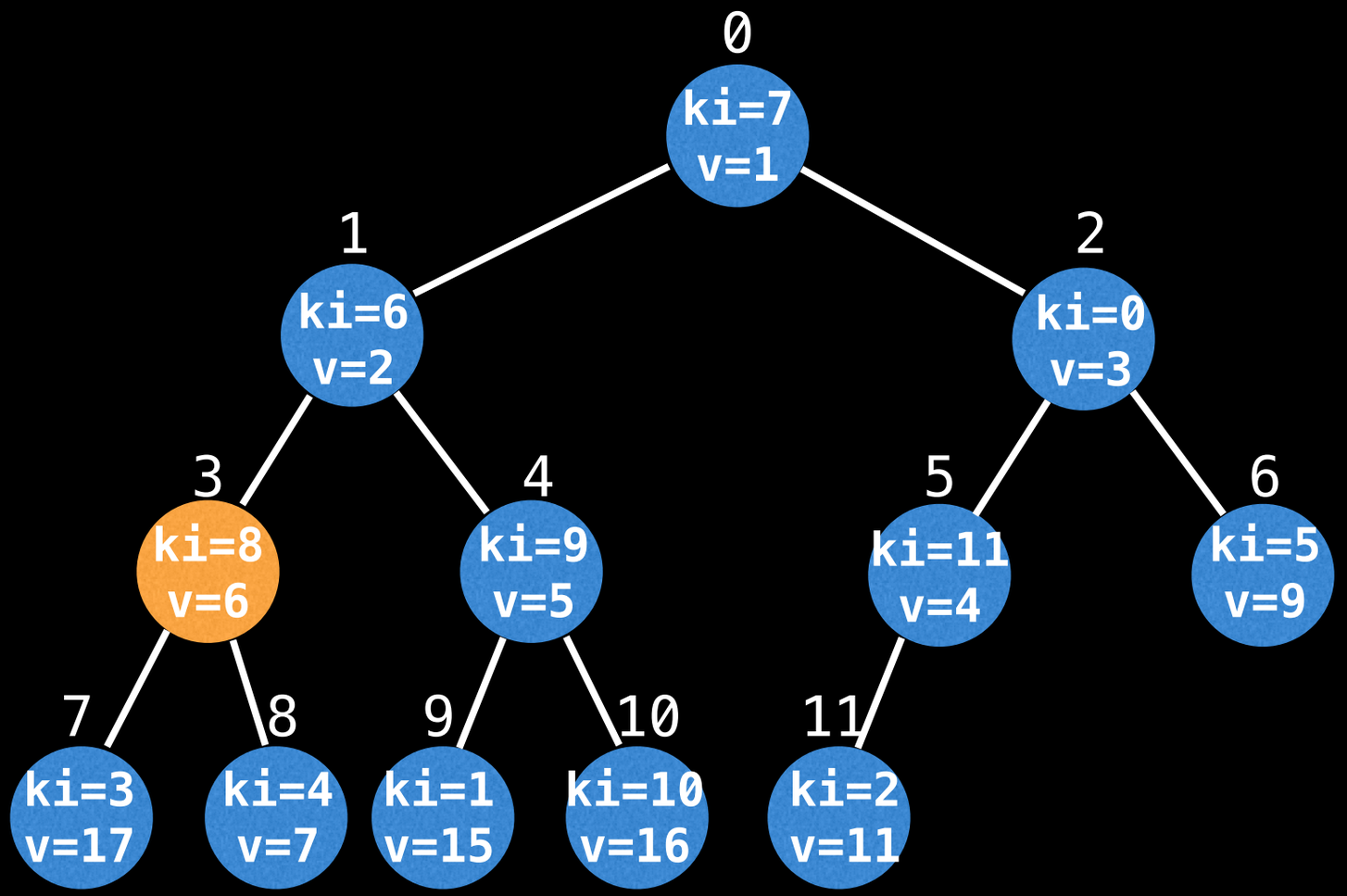


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Q: Which person (key) is being represented in the node at index position 3?

IPQ as a binary heap

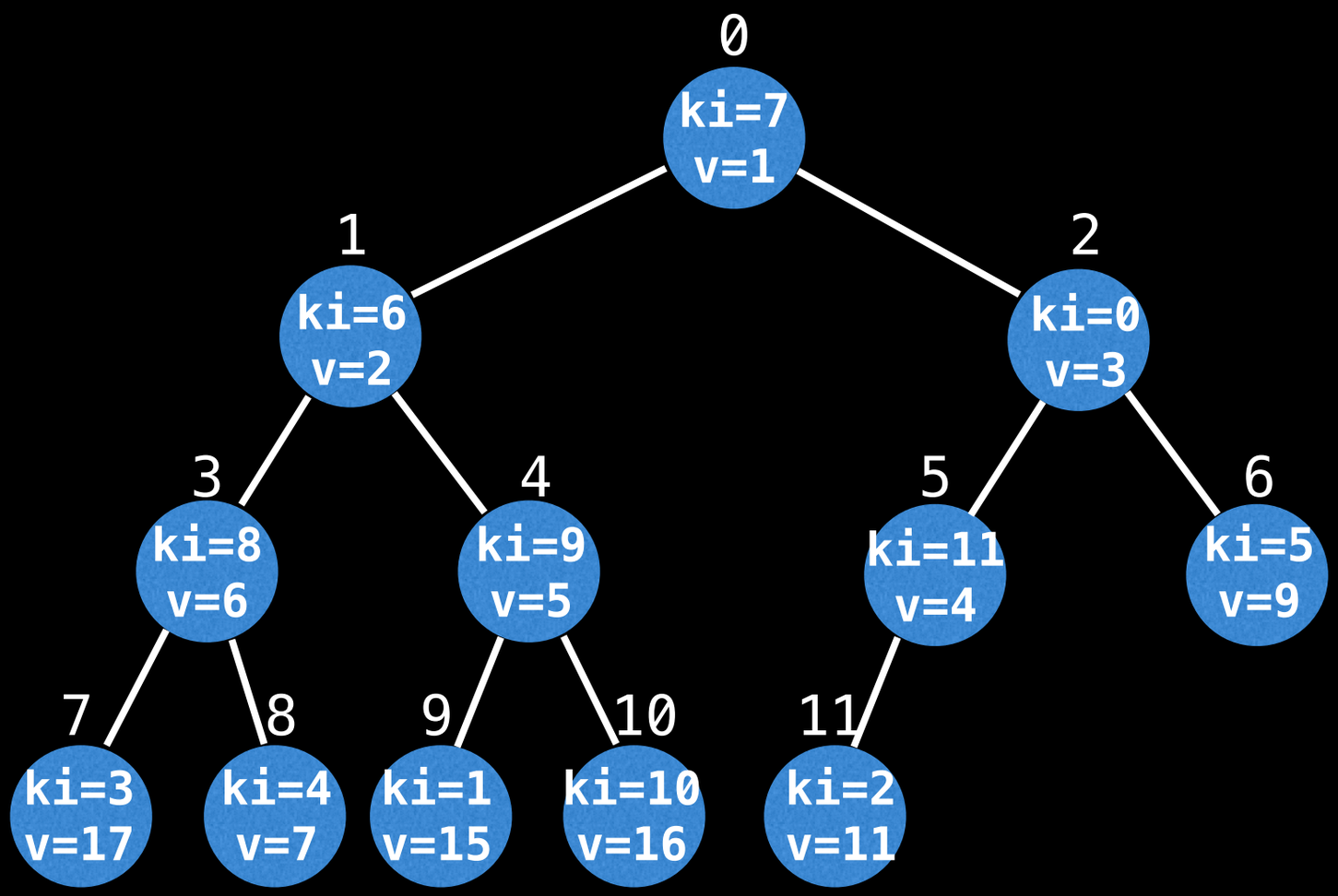


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

A: “Isaac”

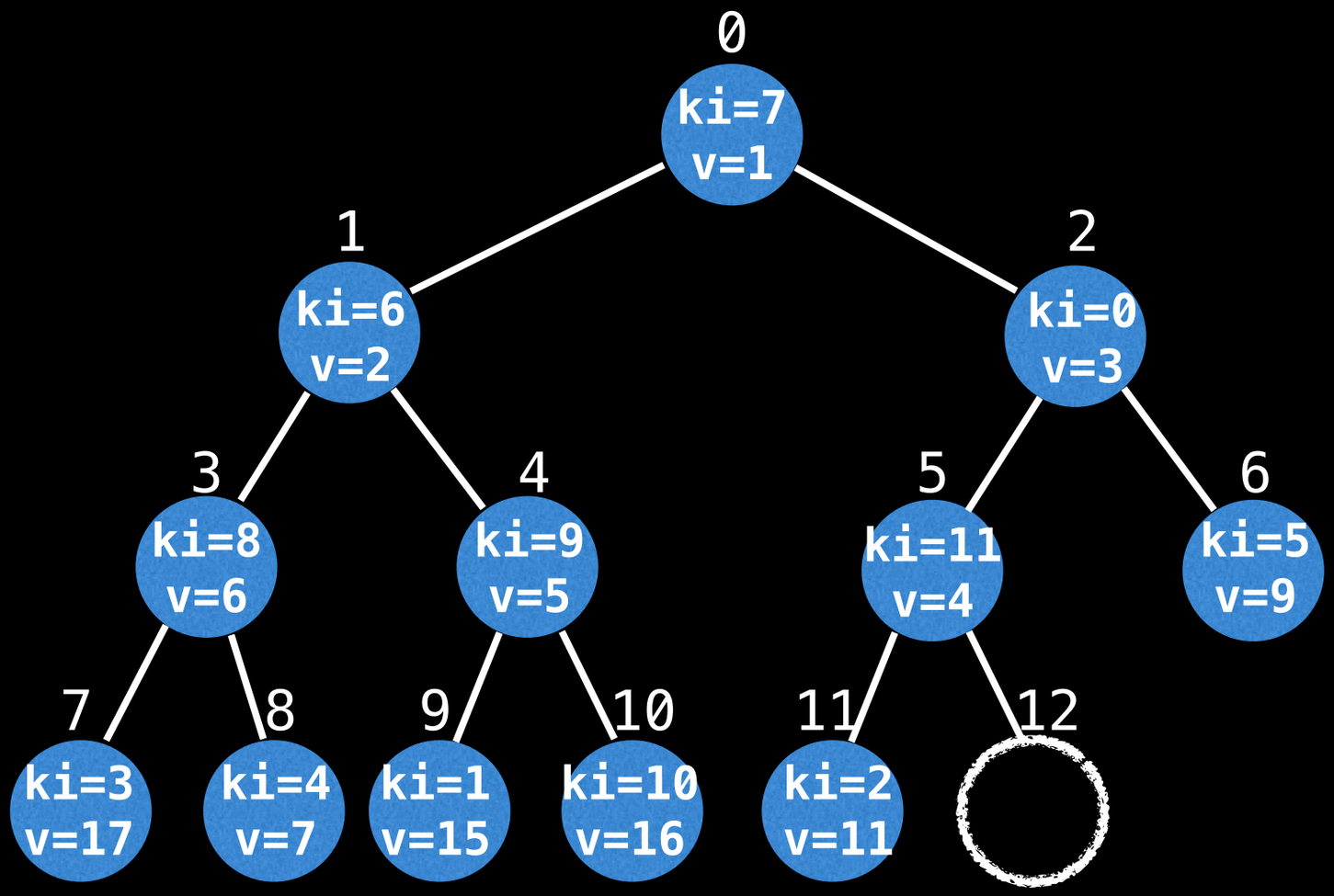
Insertion



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Insertion

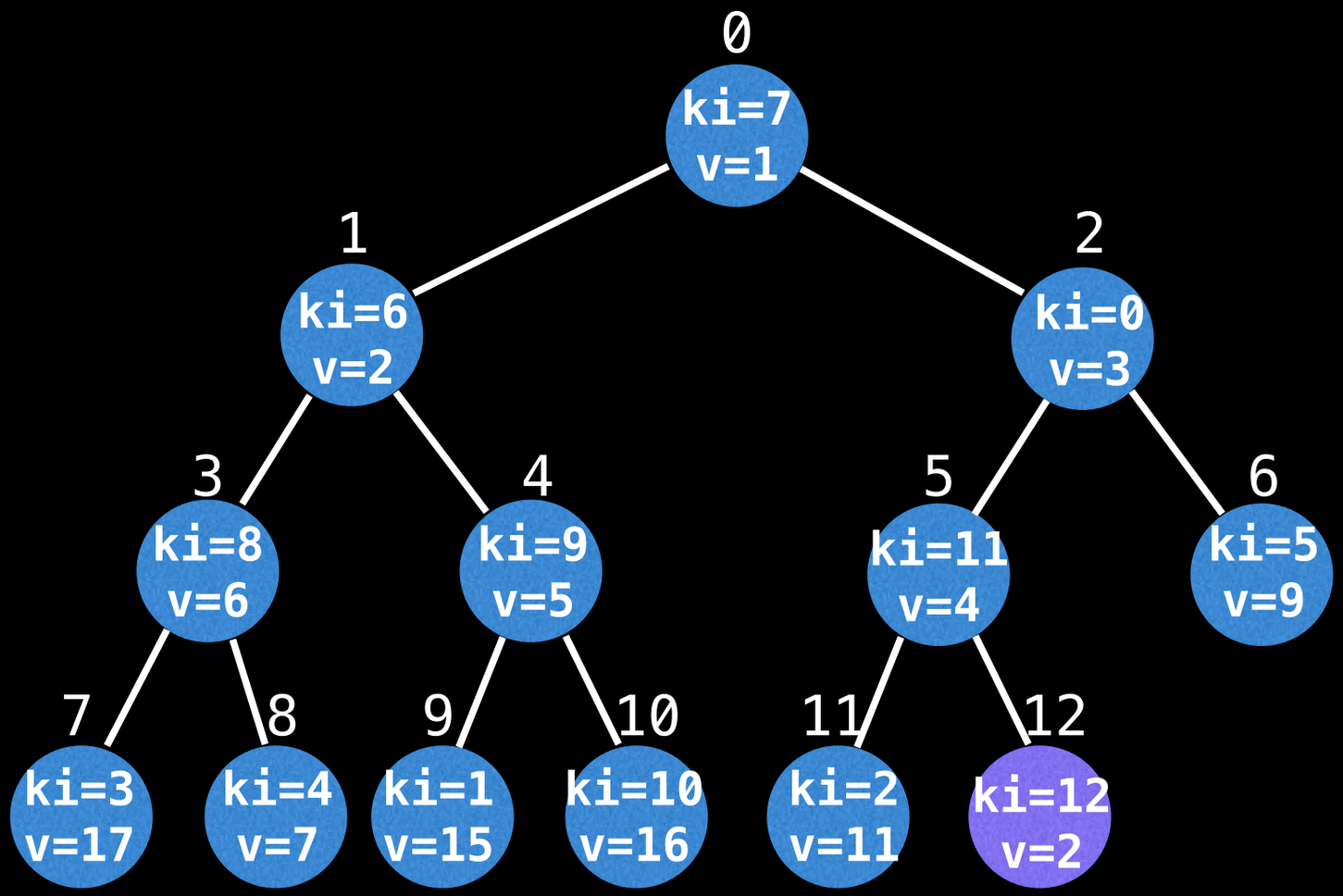


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	-1	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	-1	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	-1	-1	-1

Insert “Mary” with a value of 2

Insertion

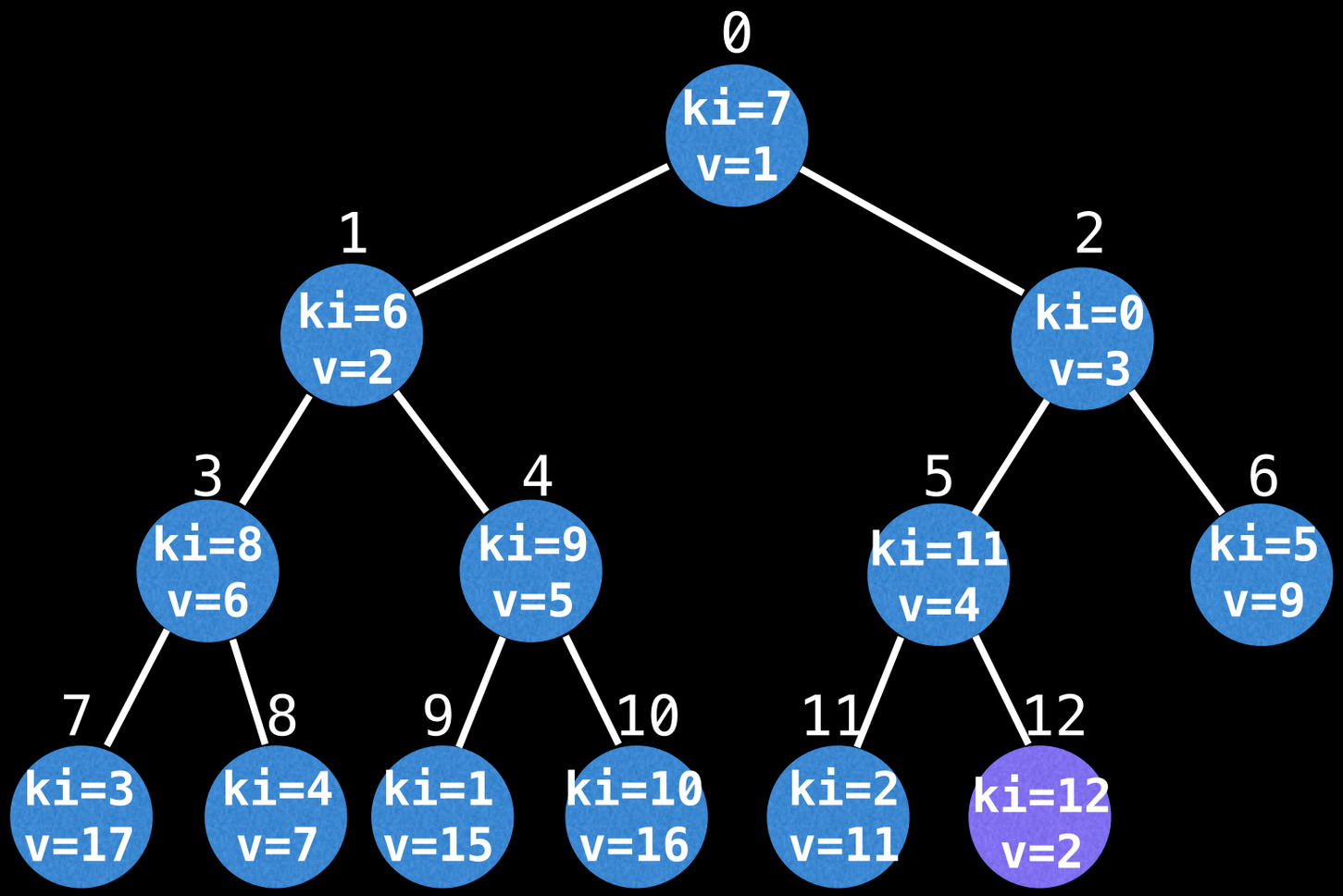


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	12	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	12	-1	-1

Insert “Mary” with a value of 2

Insertion

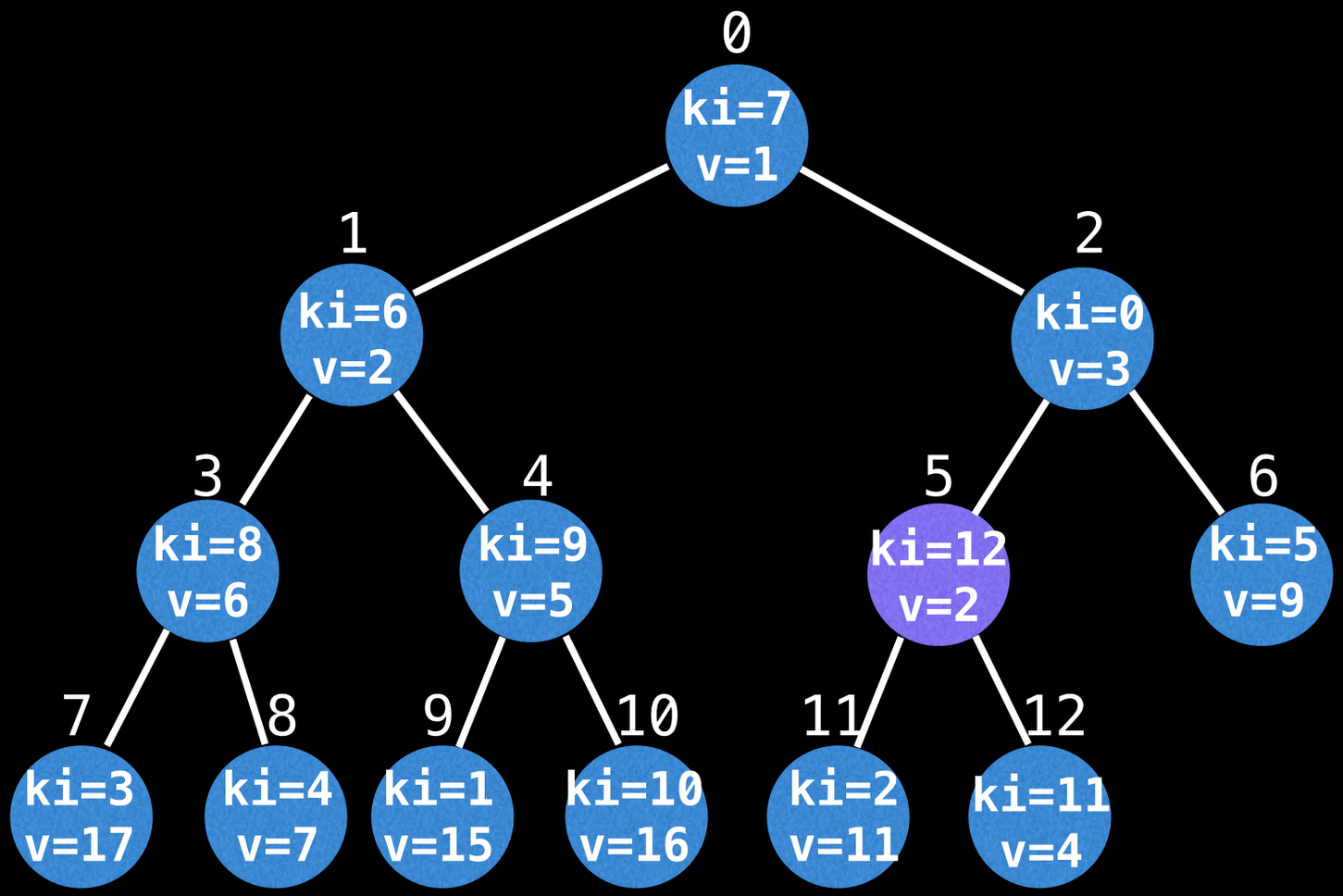


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	5	12	-1	-1
im	7	6	0	8	9	11	5	3	4	1	10	2	12	-1	-1

The heap invariant is not satisfied ATM since node at index 12 has a value less then node 5.

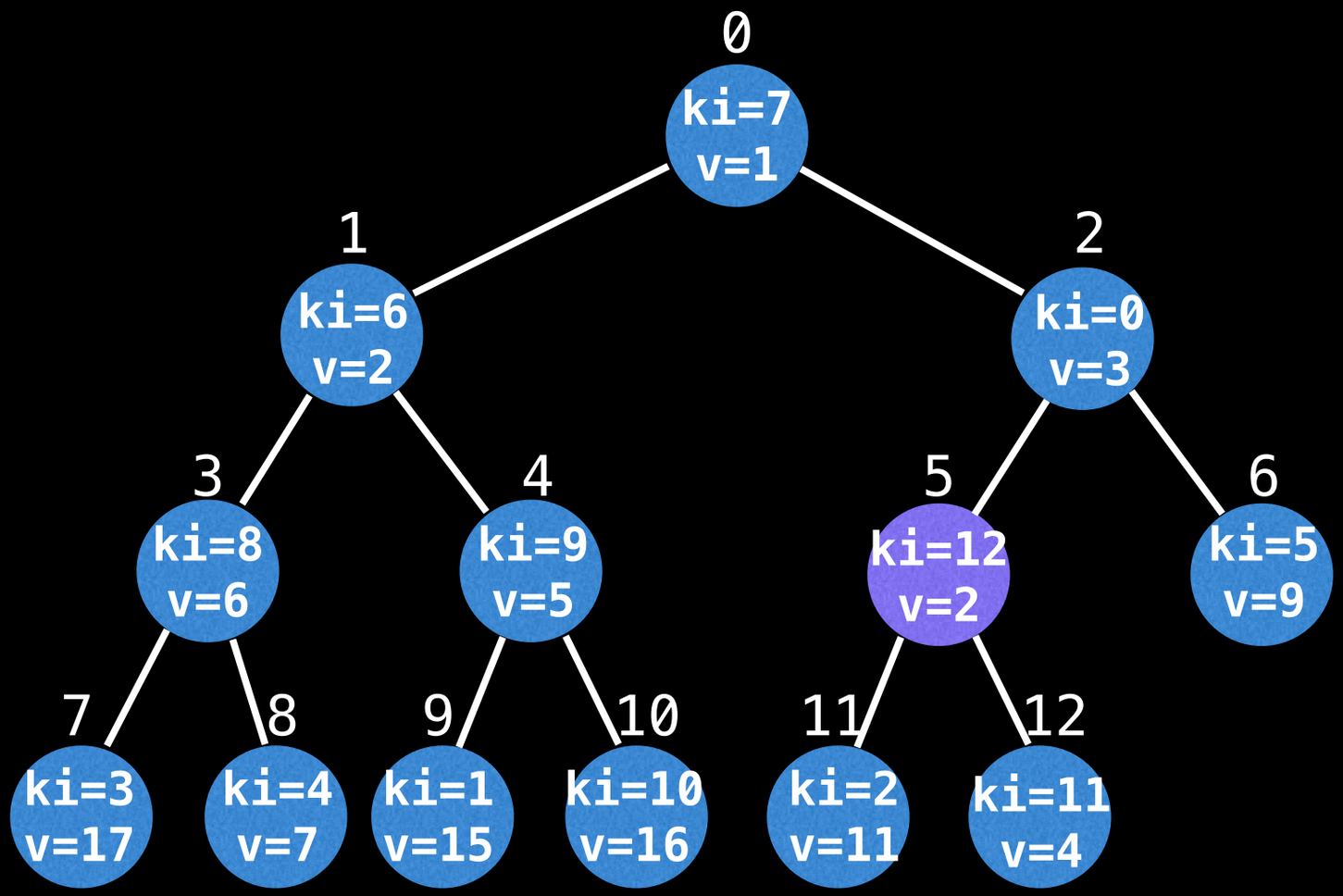
Insertion



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	12	5	-1	-1
im	7	6	0	8	9	12	5	3	4	1	10	2	11	-1	-1

Insertion

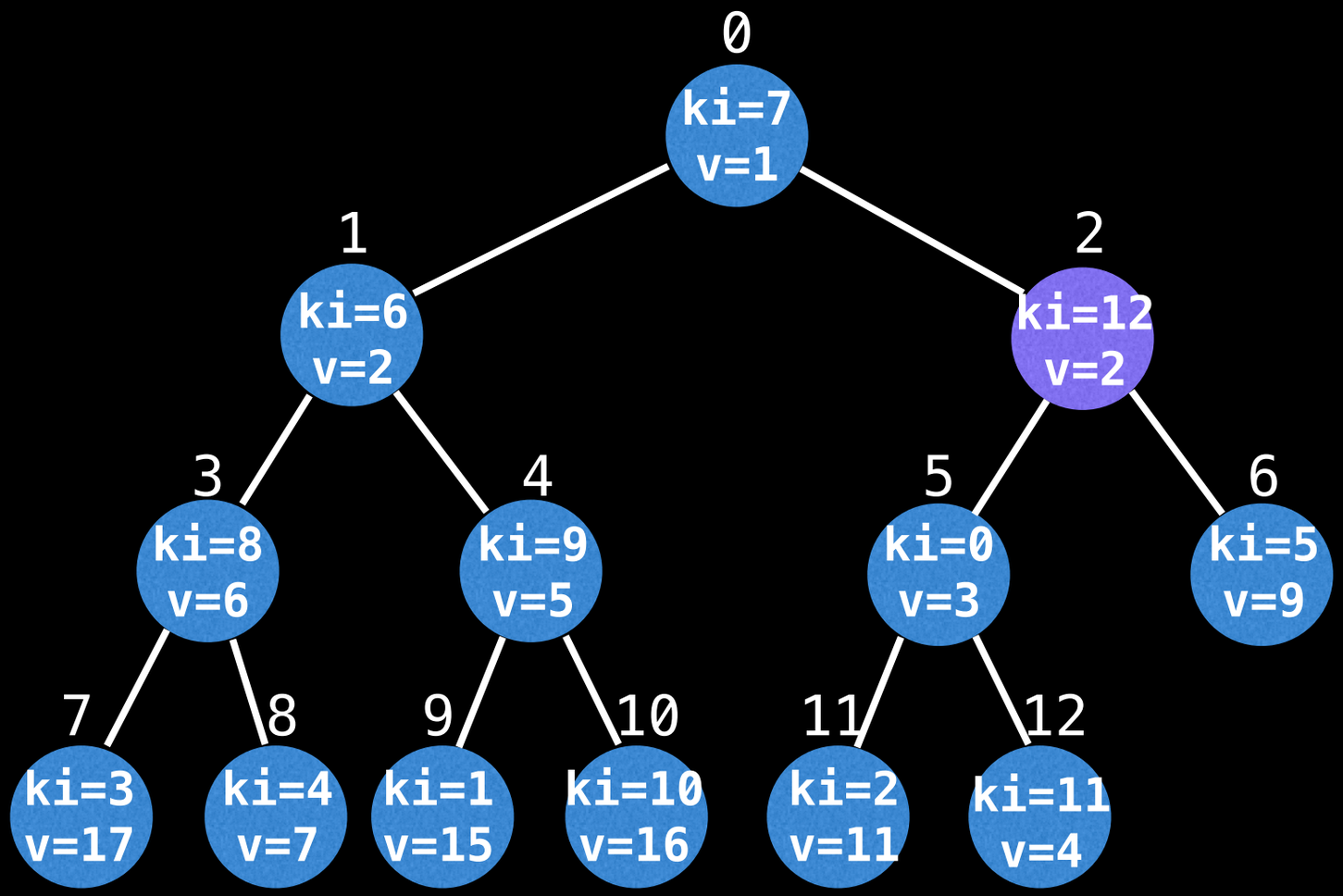


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	2	9	11	7	8	6	1	0	3	4	10	12	5	-1	-1
im	7	6	0	8	9	12	5	3	4	1	10	2	11	-1	-1

The heap invariant is still not satisfied so keep swapping upwards.

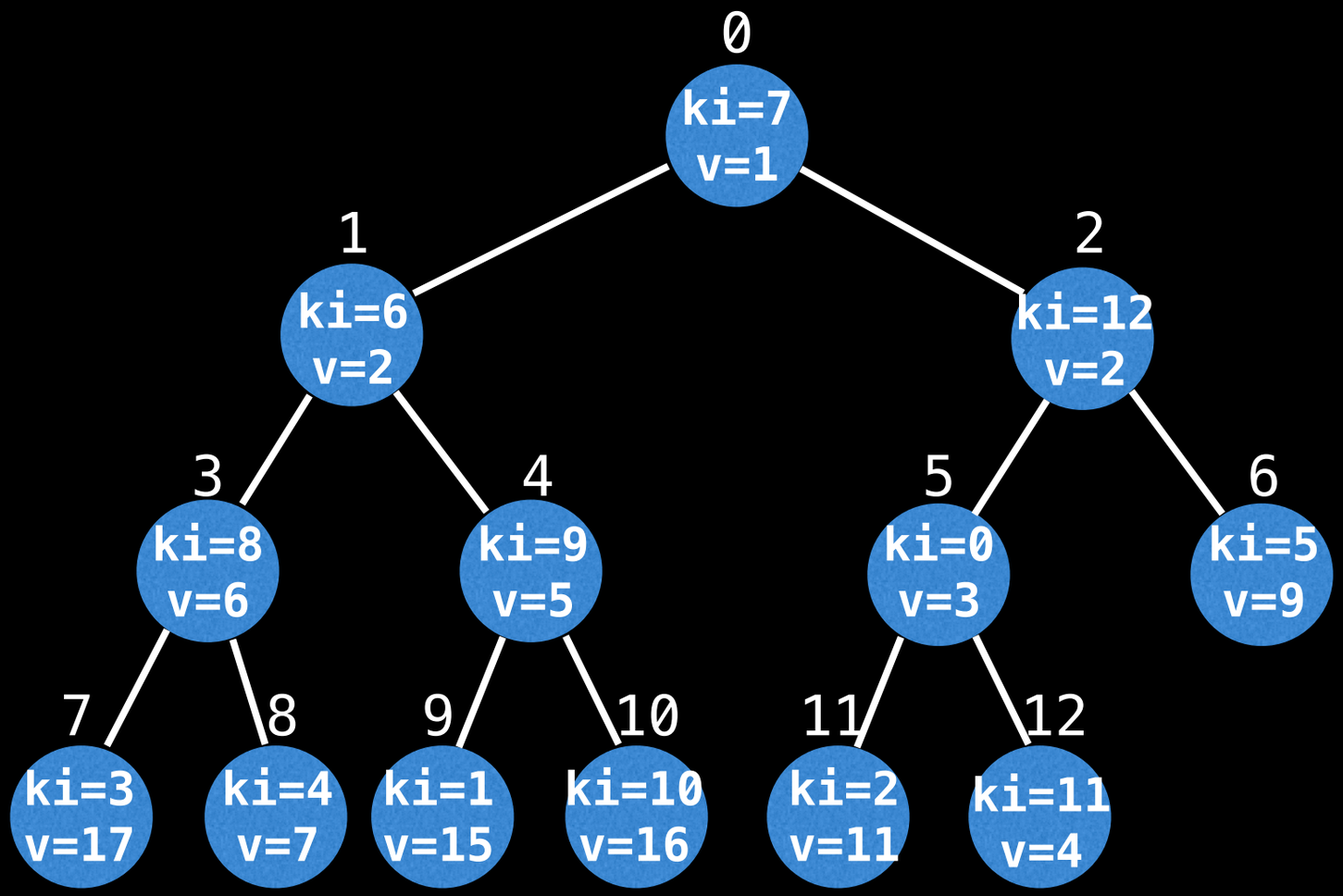
Insertion



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	0	3	4	10	12	2	-1	-1
im	7	6	12	8	9	0	5	3	4	1	10	2	11	-1	-1

Insertion



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	0	3	4	10	12	2	-1	-1
im	7	6	12	8	9	0	5	3	4	1	10	2	11	-1	-1

Insertion Pseudo Code

```
# Inserts a value into the min indexed binary
# heap. The key index must not already be in
# the heap and the value must not be null.
function insert(ki, value):
  values[ki] = value
  # ‘sz’ is the current size of the heap
  pm[ki] = sz
  im[sz] = ki
  swim(sz)
  sz = sz + 1
```

Insertion Pseudo Code

```
# Inserts a value into the min indexed binary
# heap. The key index must not already be in
# the heap and the value must not be null.
function insert(ki, value):
  values[ki] = value
  # ‘sz’ is the current size of the heap
  pm[ki] = sz
  im[sz] = ki
  swim(sz)
  sz = sz + 1
```

Insertion Pseudo Code

```
# Inserts a value into the min indexed binary
# heap. The key index must not already be in
# the heap and the value must not be null.
function insert(ki, value):
  values[ki] = value
# ‘sz’ is the current size of the heap
pm[ki] = sz
im[sz] = ki
swim(sz)
sz = sz + 1
```

Insertion Pseudo Code

```
# Inserts a value into the min indexed binary
# heap. The key index must not already be in
# the heap and the value must not be null.
function insert(ki, value):
  values[ki] = value
  # ‘sz’ is the current size of the heap
  pm[ki] = sz
  im[sz] = ki
  swim(sz)
  sz = sz + 1
```

Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

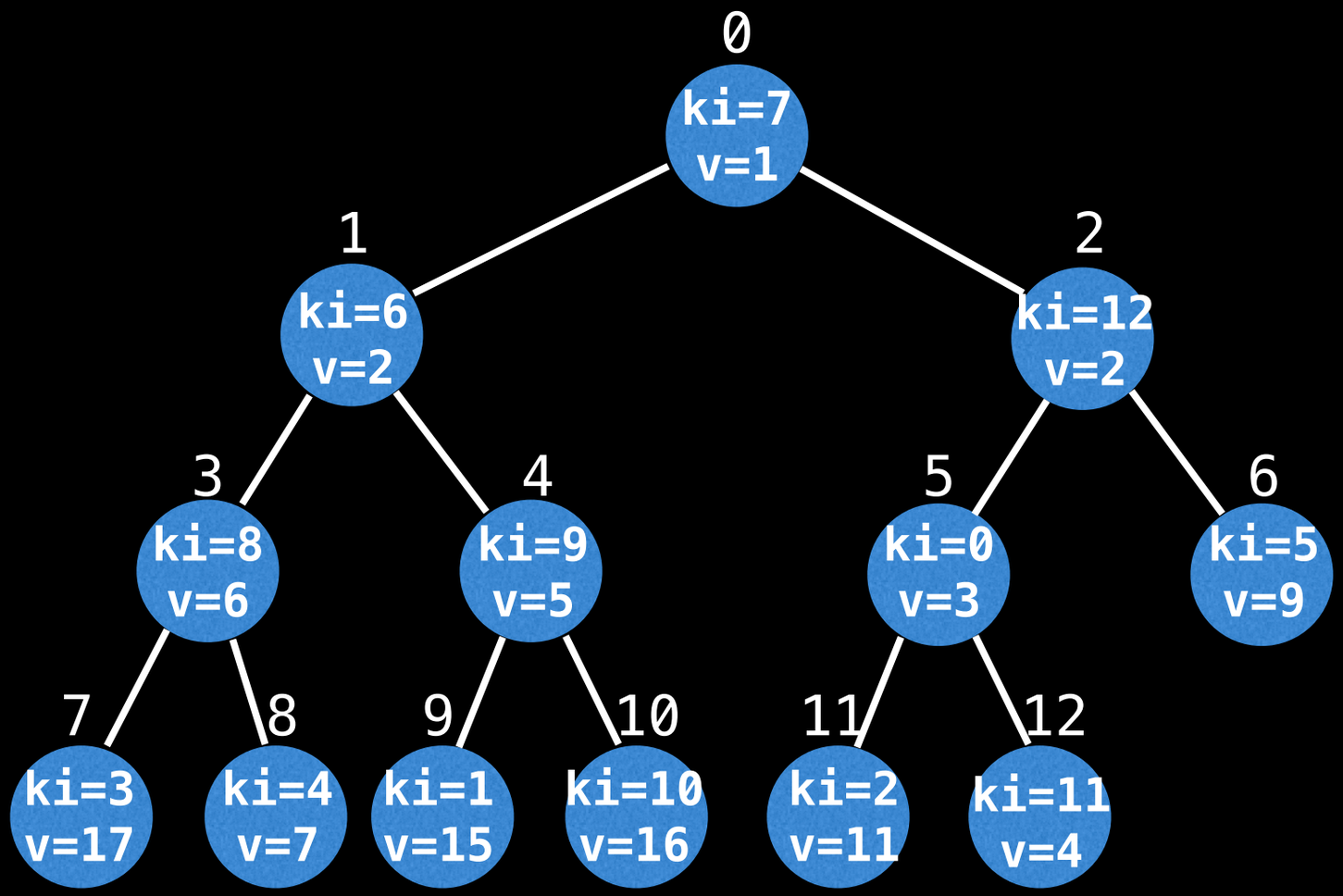
Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

Swim Pseudo Code

```
# Swims up node i (zero based) until heap
# invariant is satisfied.
function swim(i):
  for (p = (i-1)/2; i > 0 and less(i, p)):
    swap(i, p)
    i = p
    p = (i-1)/2

function swap(i, j):
  pm[im[j]] = i
  pm[im[i]] = j
  tmp = im[i]
  im[i] = im[j]
  im[j] = tmp

function less(i, j):
  return values[im[i]] < values[im[j]]
```

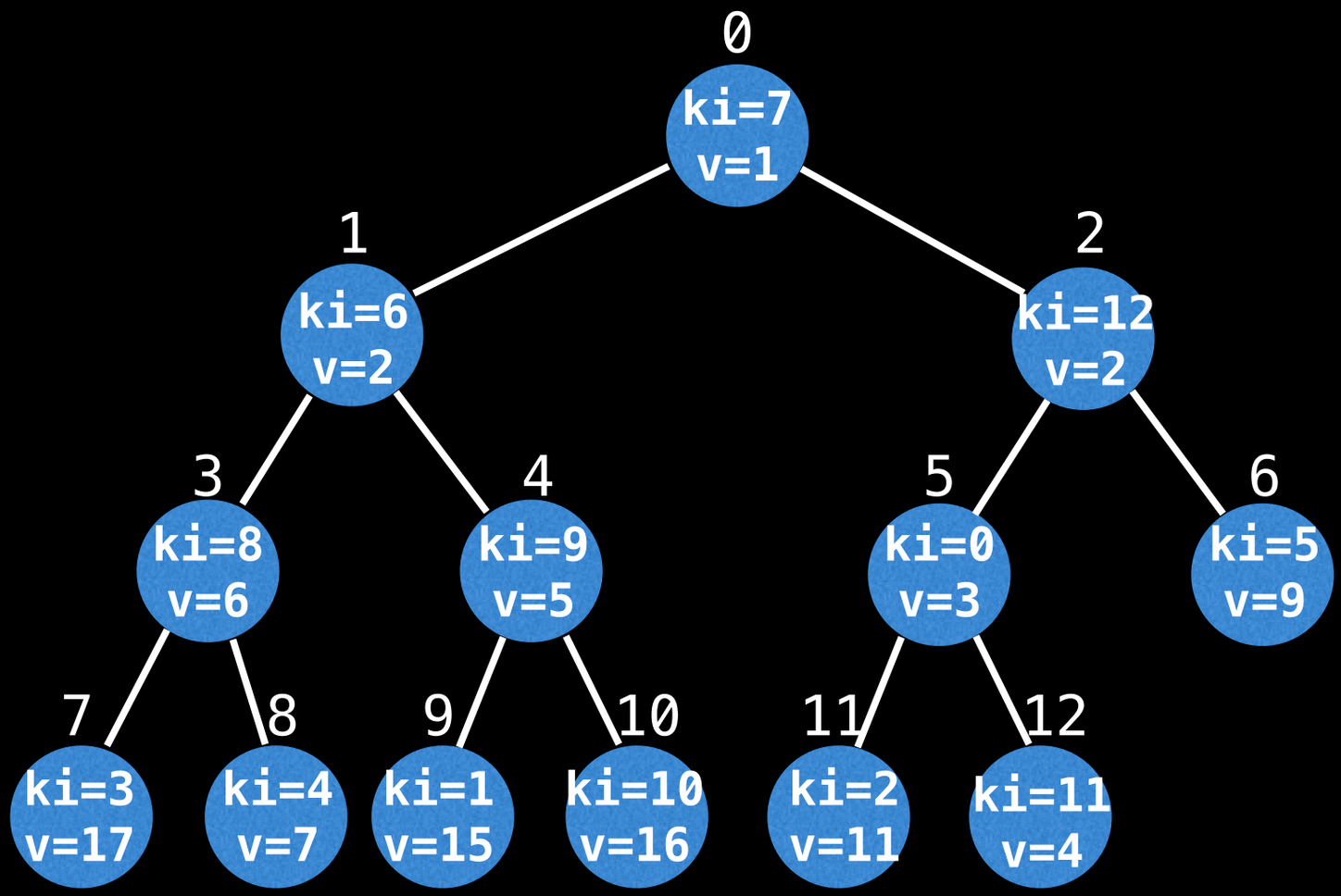
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

Polling is still **O(log(n))** in an IPQ, but removing is improved from **O(n)** in a traditional PQ to **O(log(n))** since node position lookups are **O(1)** but repositioning is still **O(log(n))**

Polling & Removals

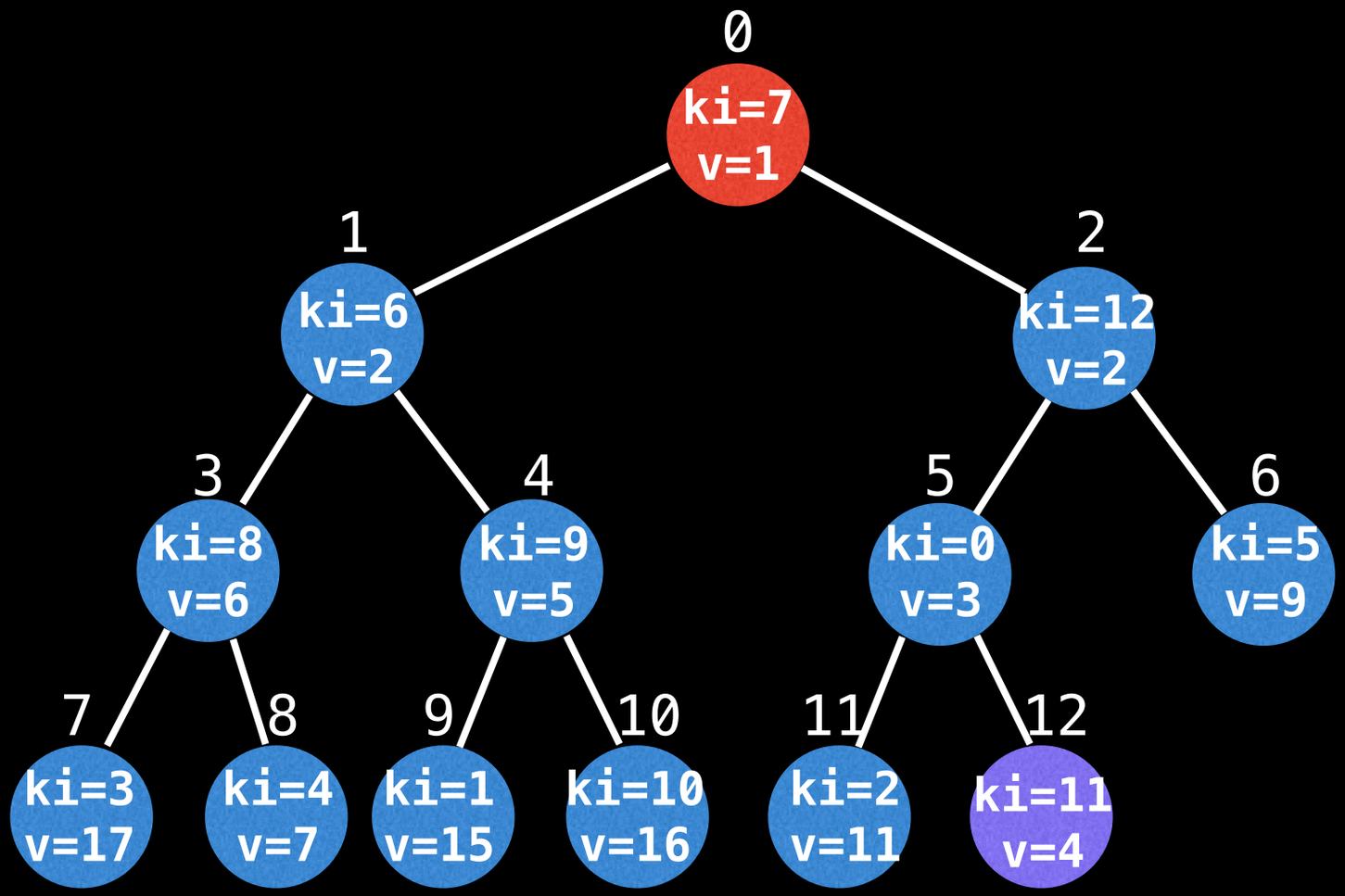


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	0	3	4	10	12	2	-1	-1
im	7	6	12	8	9	0	5	3	4	1	10	2	11	-1	-1

Let's poll the root node. The required steps are almost exactly like a regular binary heap.

Polling & Removals

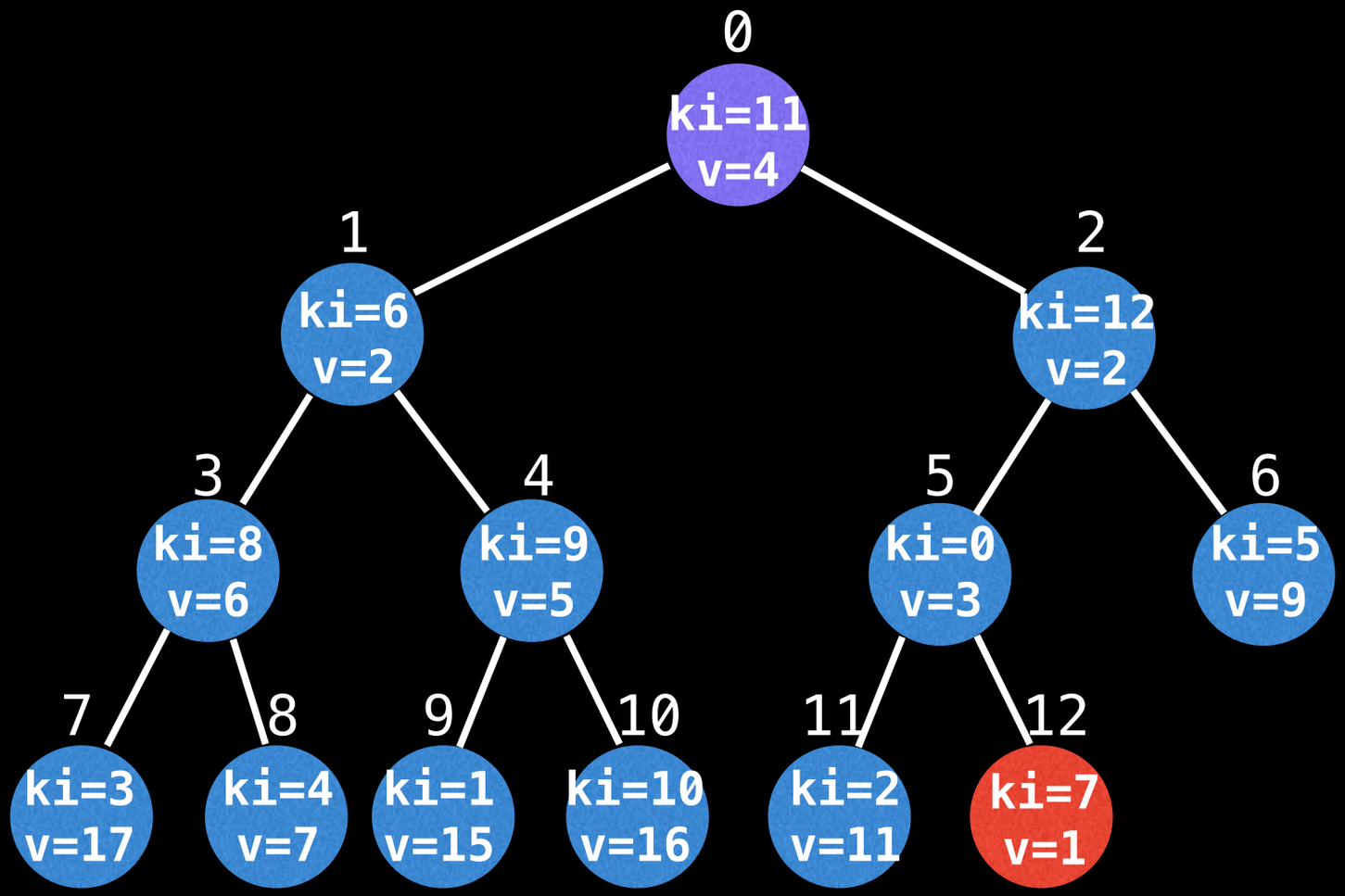


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	0	3	4	10	12	2	-1	-1
im	7	6	12	8	9	0	5	3	4	1	10	2	11	-1	-1

Exchange the root node and the bottom right node.

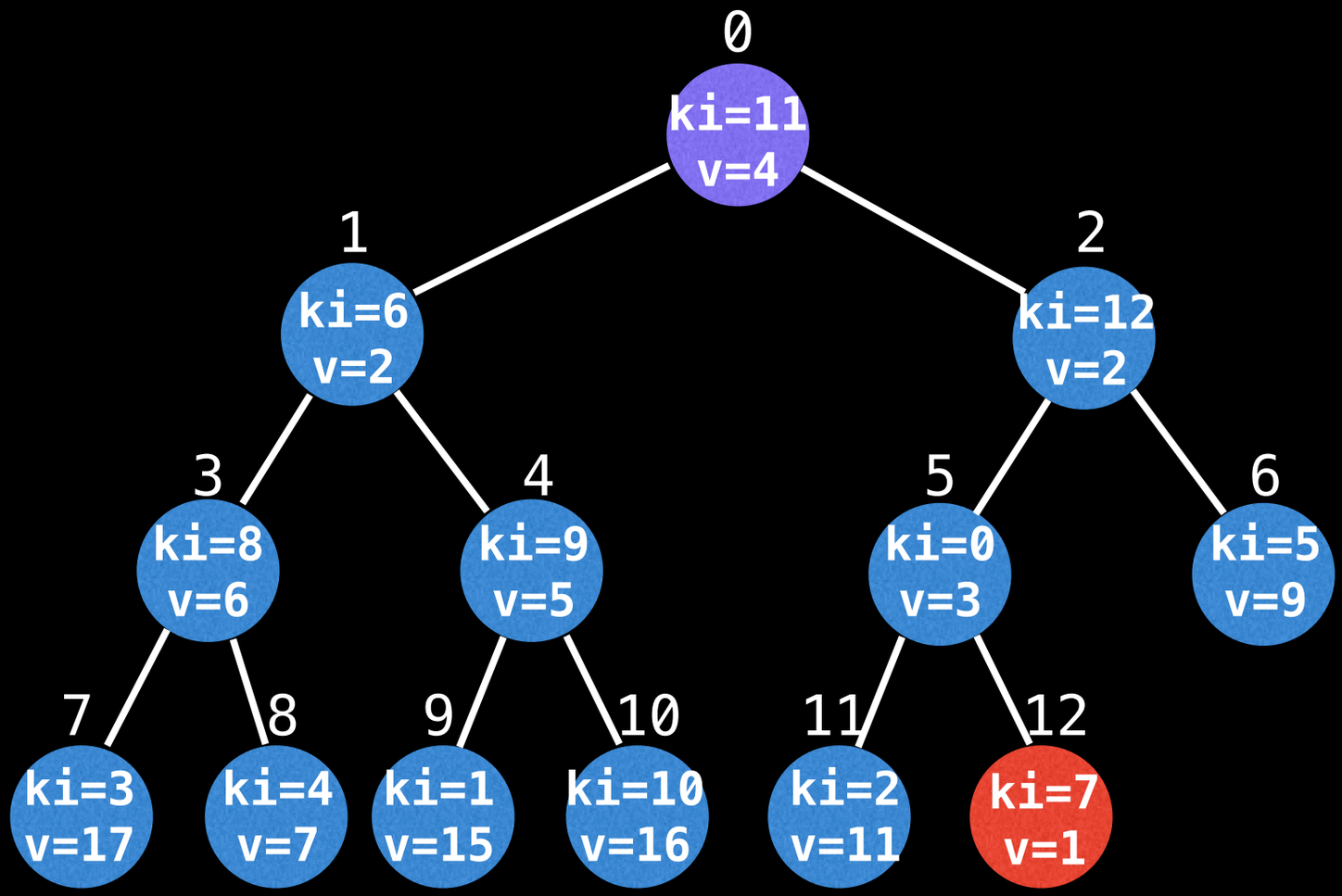
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	12	3	4	10	0	2	-1	-1
im	11	6	12	8	9	0	5	3	4	1	10	2	7	-1	-1

Polling & Removals

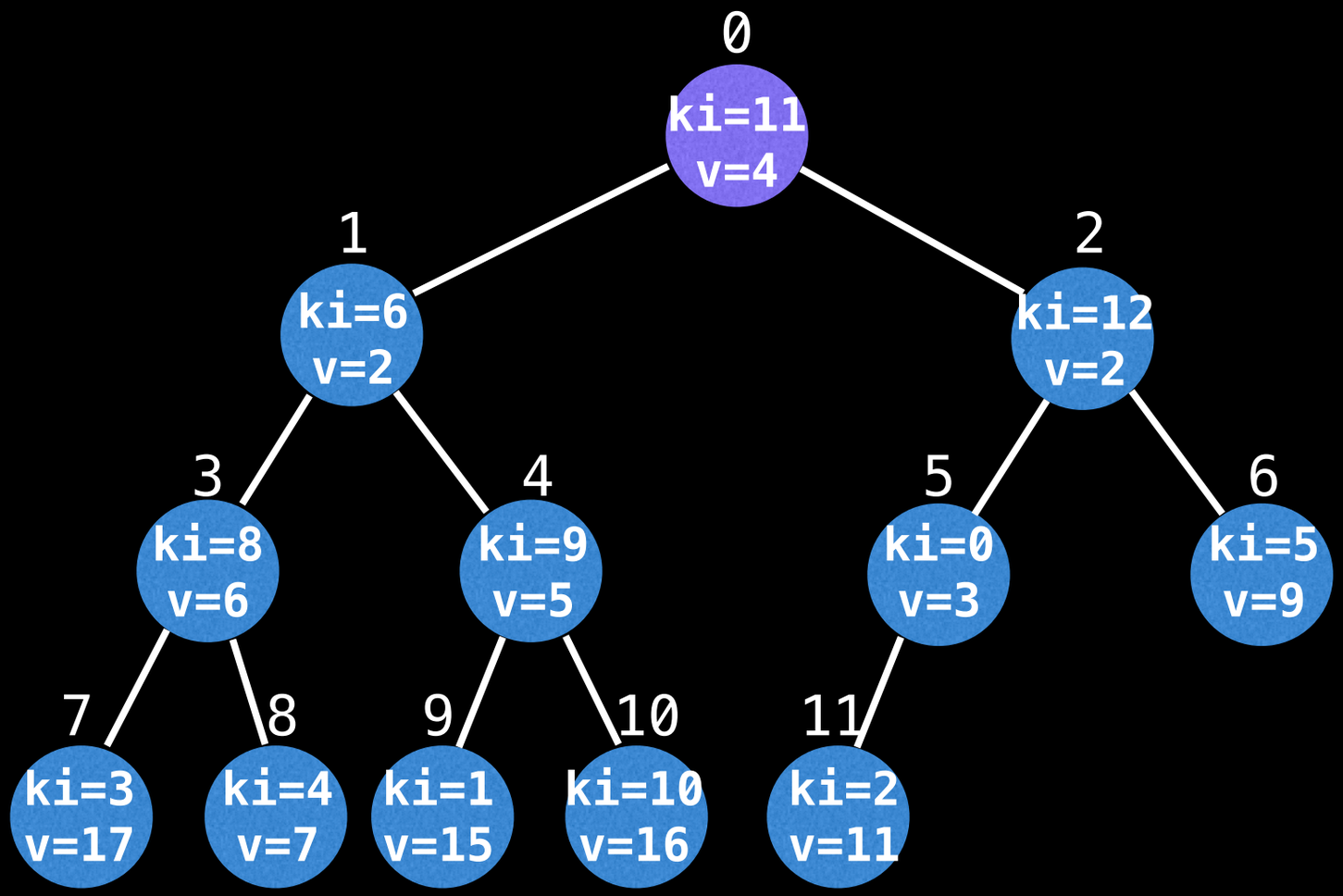


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	1	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	12	3	4	10	0	2	-1	-1
im	11	6	12	8	9	0	5	3	4	1	10	2	7	-1	-1

Remove and return the key-value pair (7, 1)

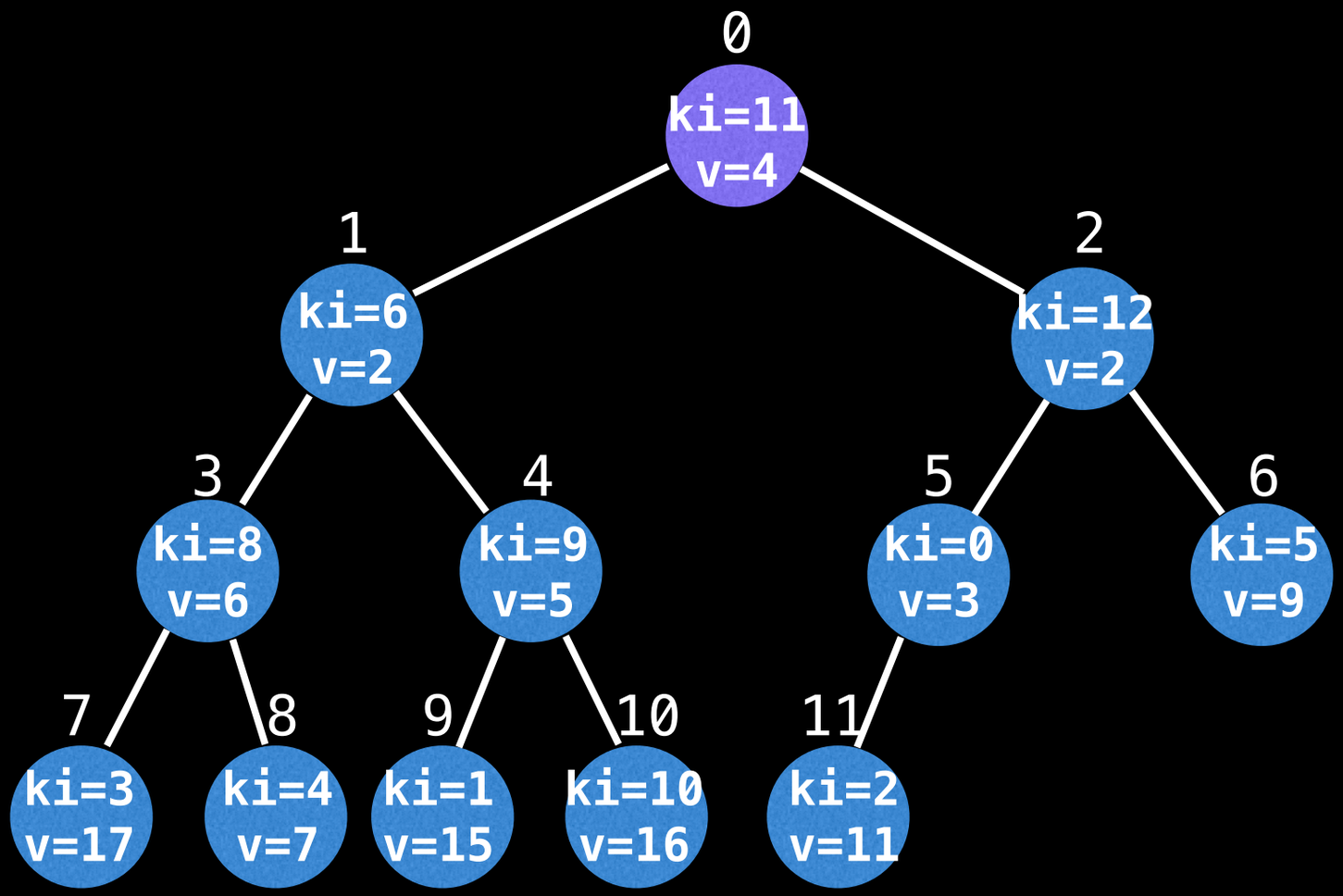
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	-1	3	4	10	0	2	-1	-1
im	11	6	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

Polling & Removals

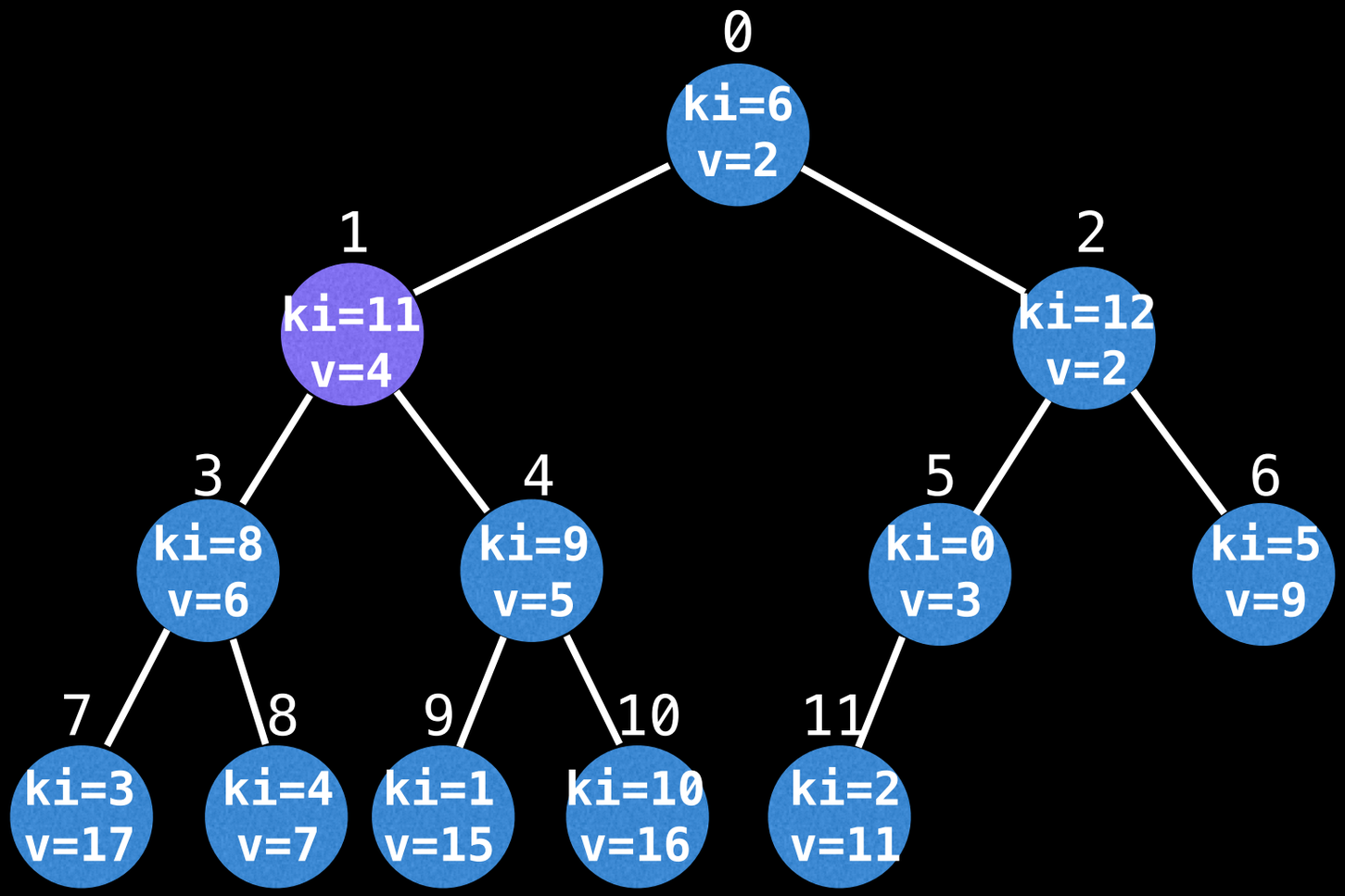


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	1	-1	3	4	10	0	2	-1	-1
im	11	6	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

Finally restore heap invariant by moving swapped purple node up or down.

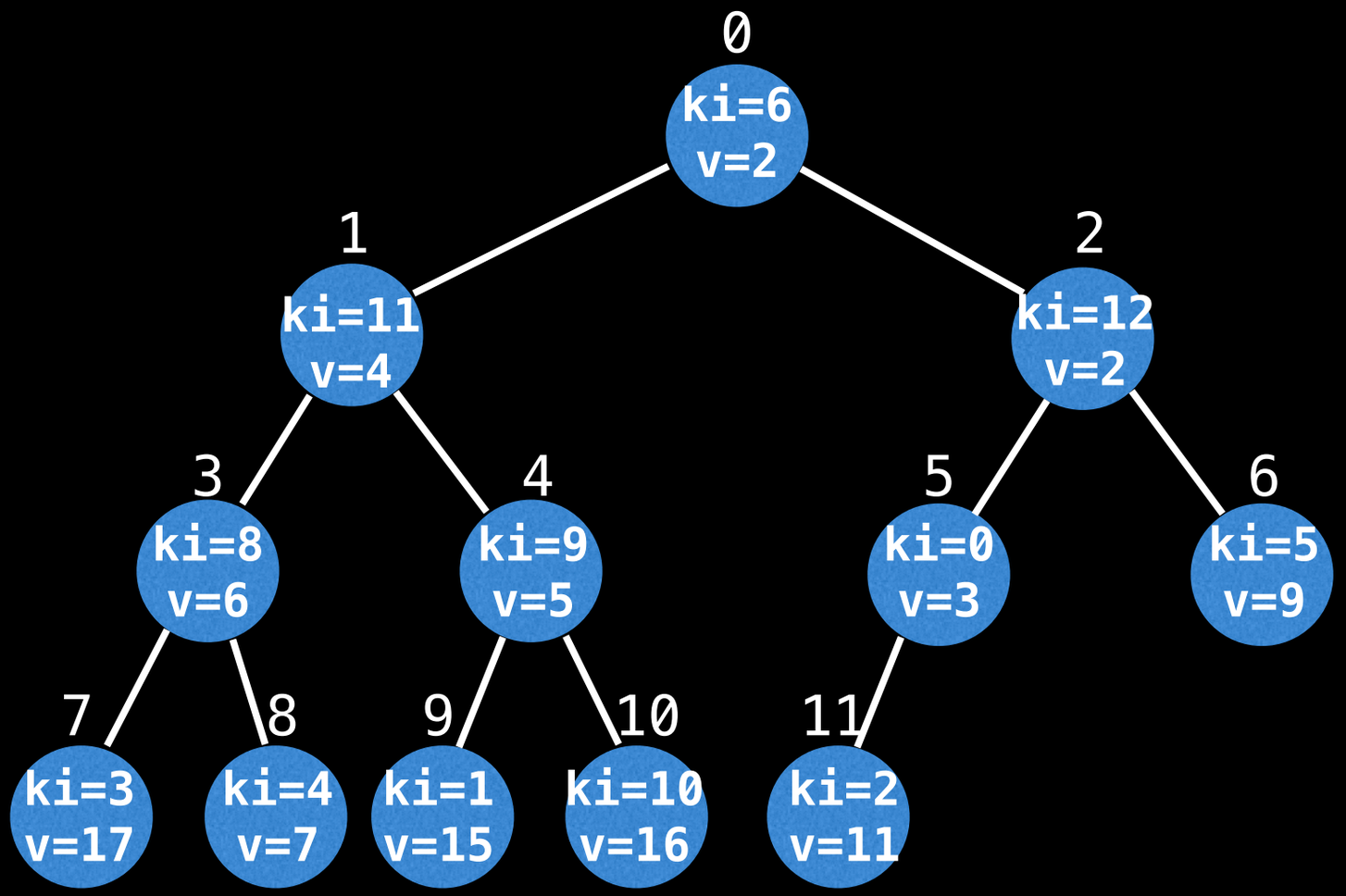
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	0	-1	3	4	10	1	2	-1	-1
im	6	11	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

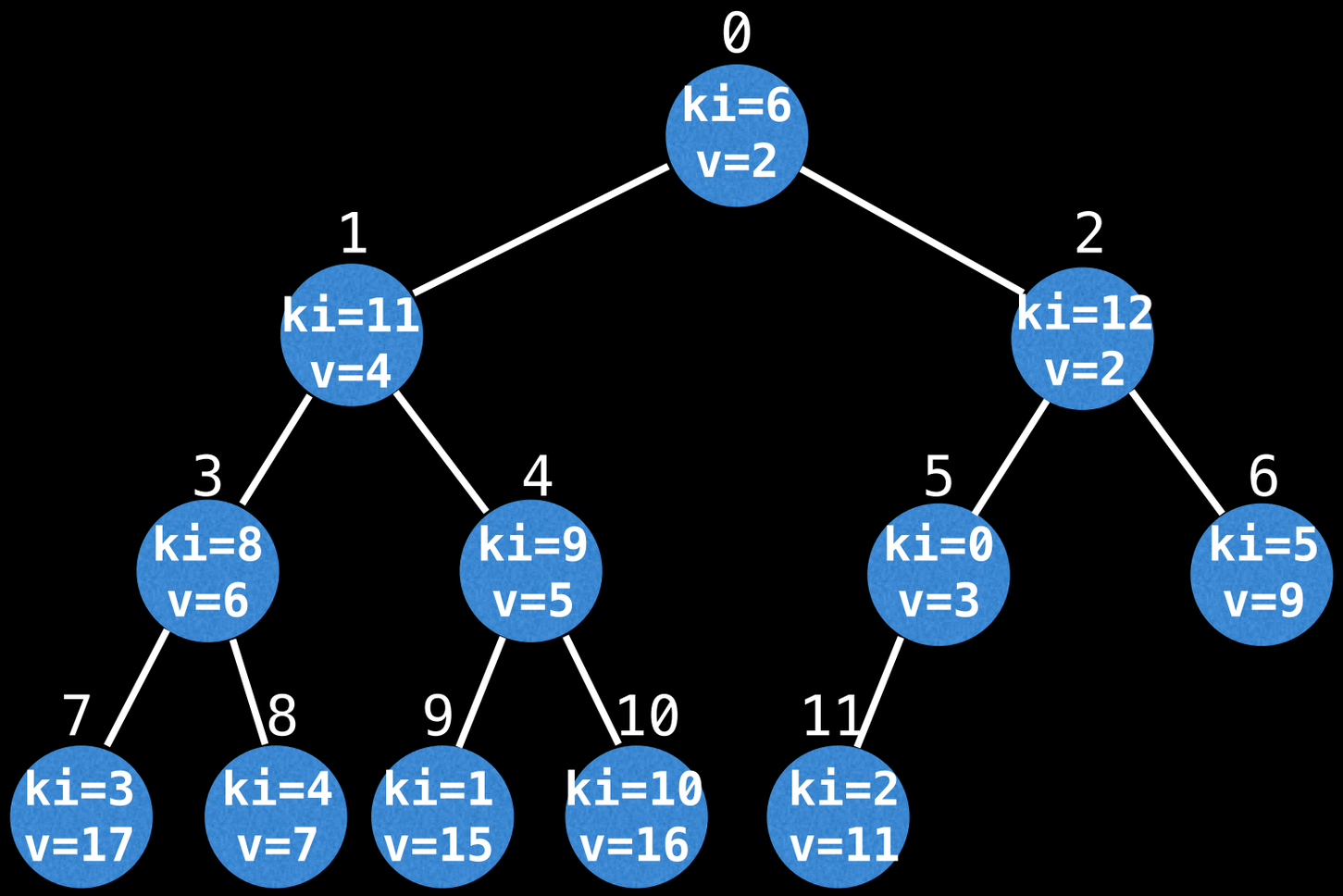
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	0	-1	3	4	10	1	2	-1	-1
im	6	11	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

Polling & Removals

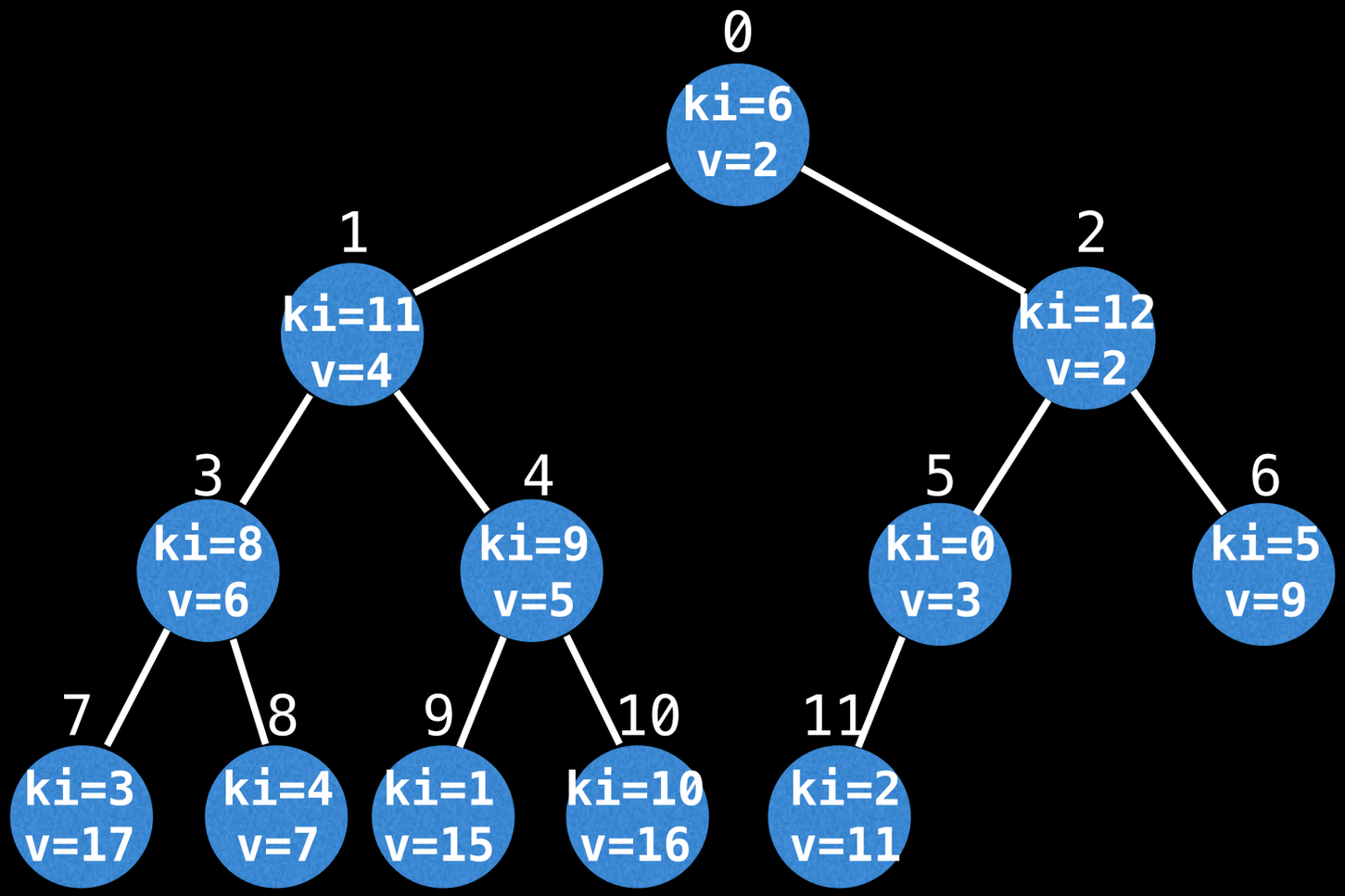


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	0	-1	3	4	10	1	2	-1	-1
im	6	11	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

Now let’s remove “Laura” from IPQ

Polling & Removals

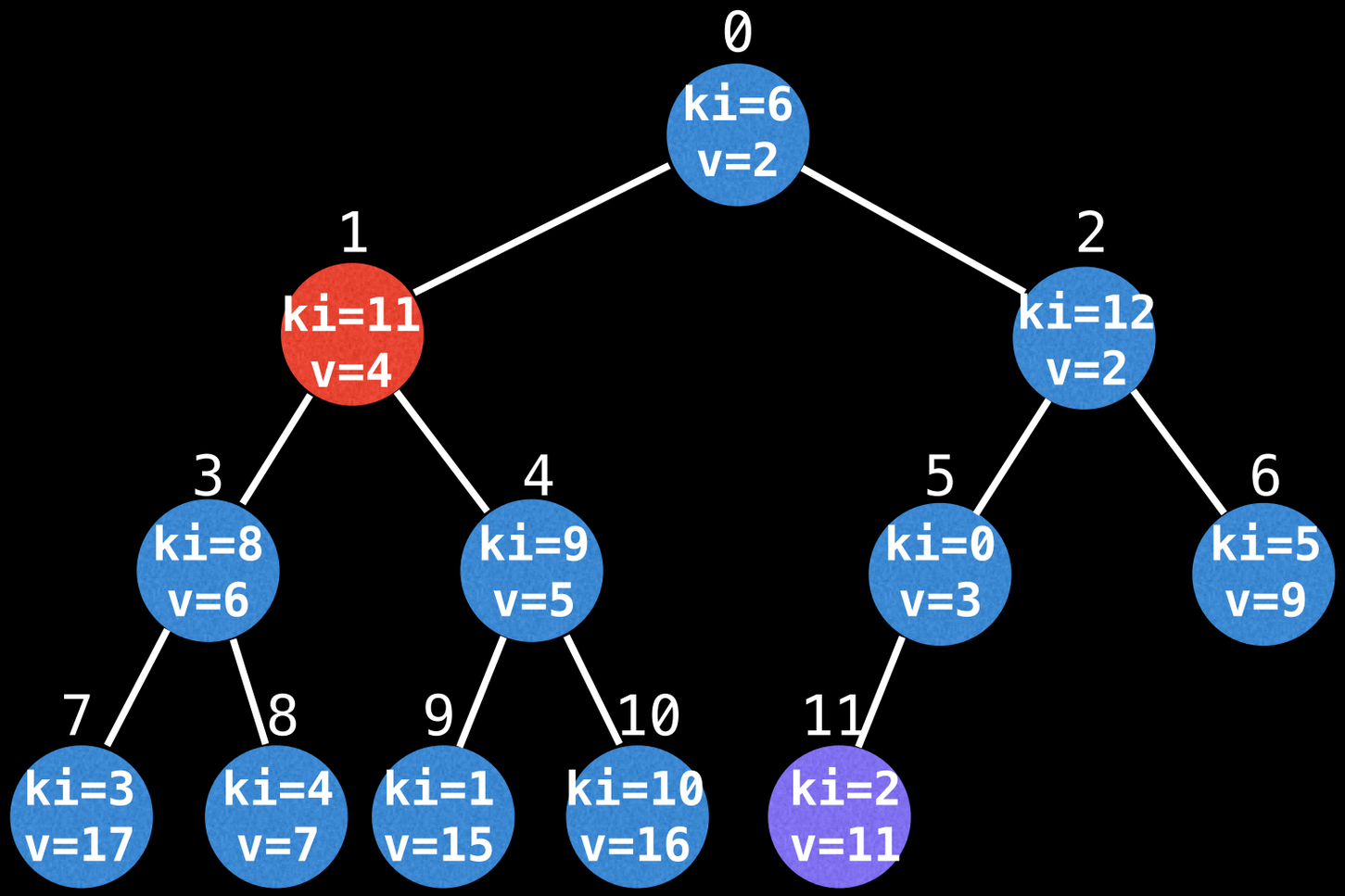


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura →	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	0	-1	3	4	10	1	2	-1	-1
im	6	11	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

The ki for “Laura” is 11

Polling & Removals

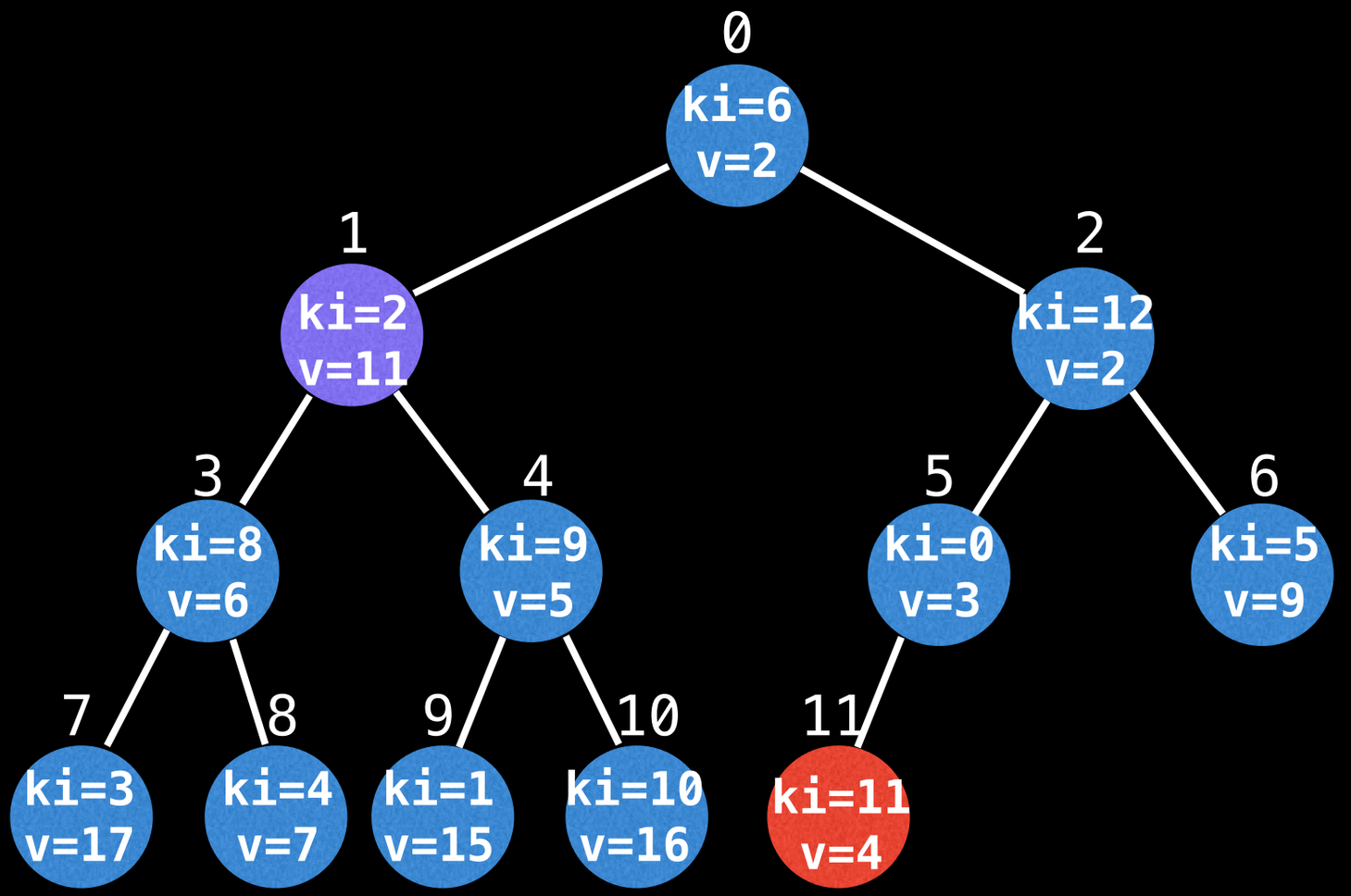


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura →	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	11	7	8	6	0	-1	3	4	10	1	2	-1	-1
im	6	11	12	8	9	0	5	3	4	1	10	2	-1	-1	-1

Look inside the position map to find what node is associated with ki=11 and perform swap.

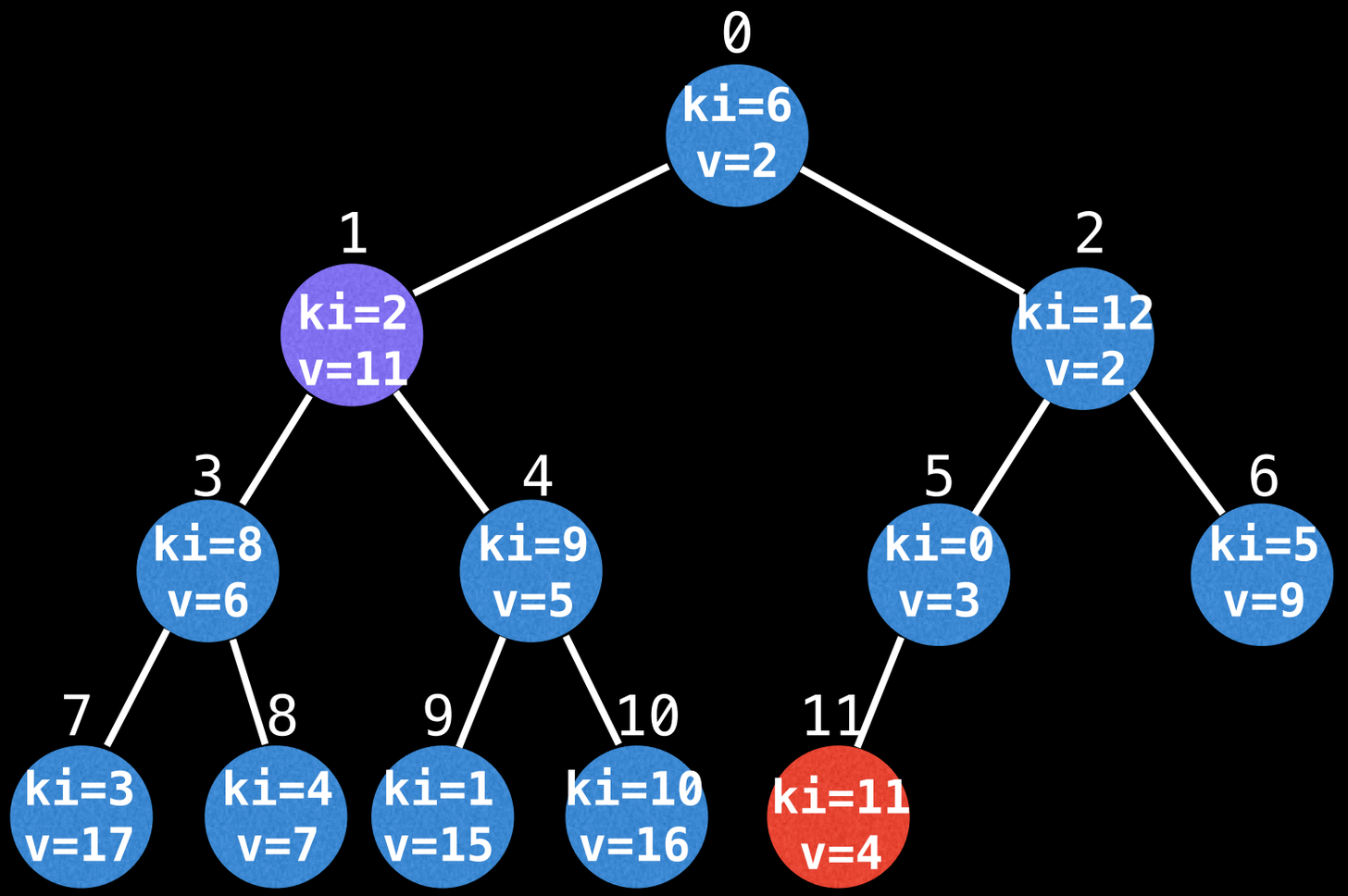
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	1	7	8	6	0	-1	3	4	10	11	2	-1	-1
im	6	2	12	8	9	0	5	3	4	1	10	11	-1	-1	-1

Polling & Removals

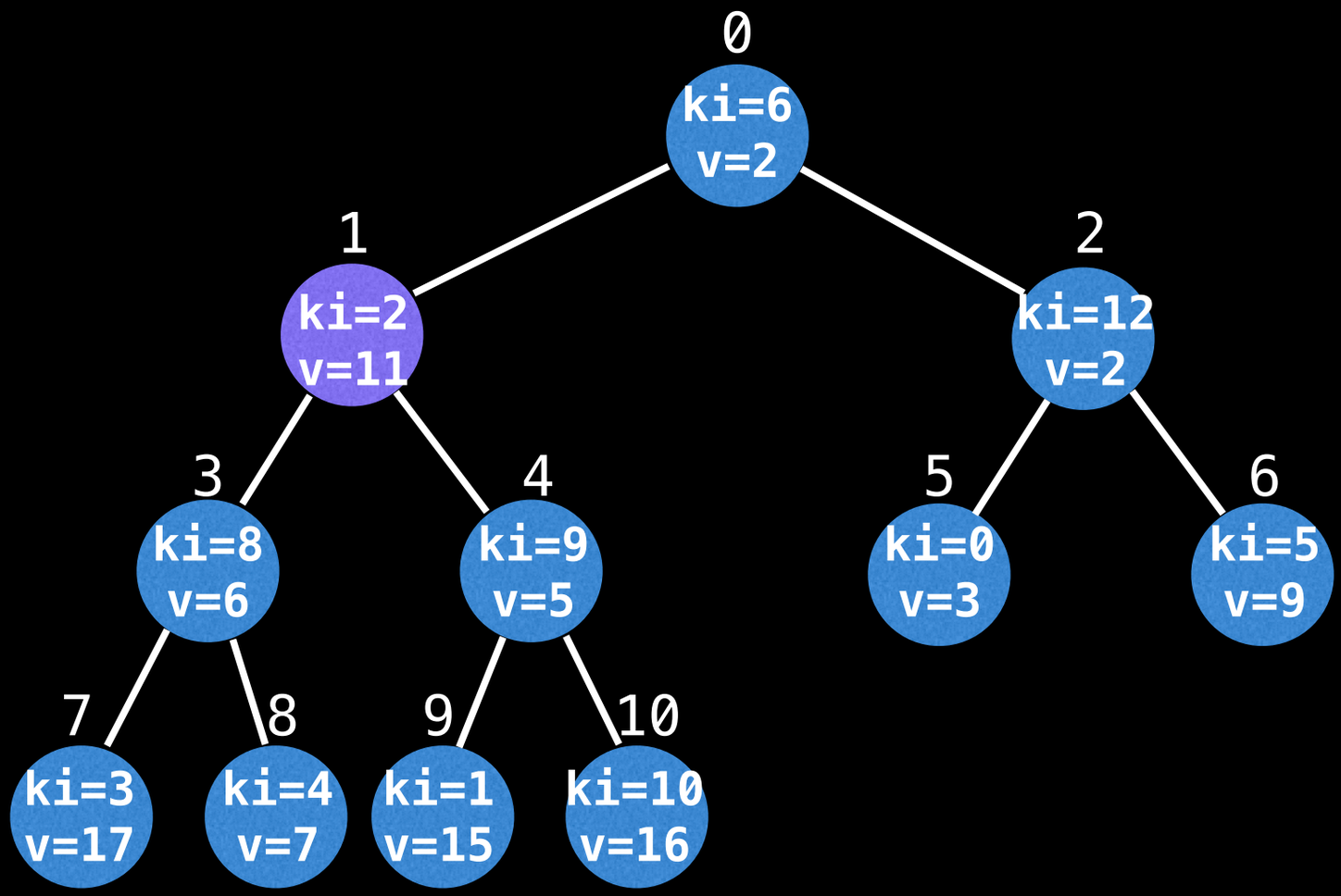


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	4	2	-1	-1
pm	5	9	1	7	8	6	0	-1	3	4	10	11	2	-1	-1
im	6	2	12	8	9	0	5	3	4	1	10	11	-1	-1	-1

Remove and return key-value pair (11, 4)

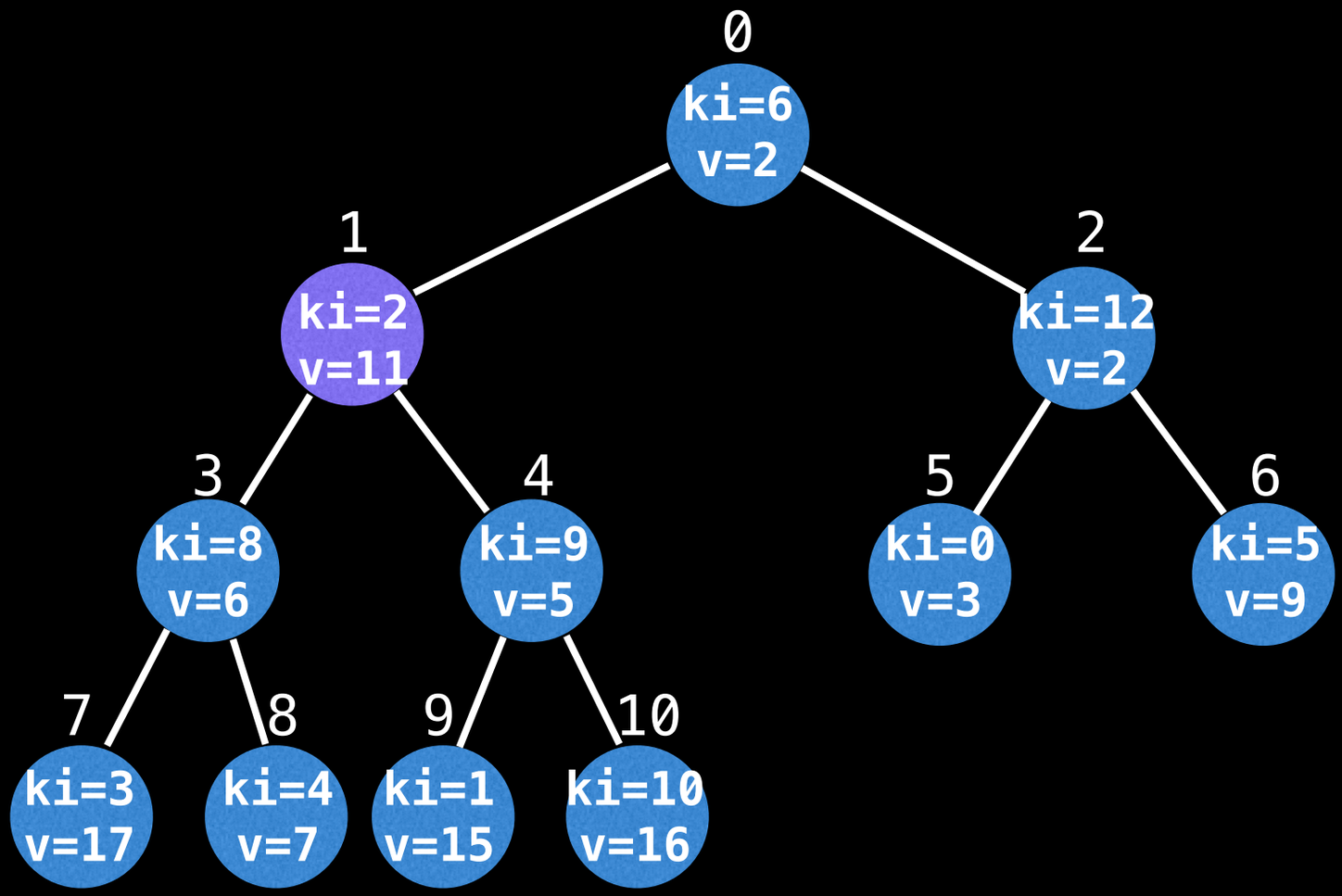
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	1	7	8	6	0	-1	3	4	10	-1	2	-1	-1
im	6	2	12	8	9	0	5	3	4	1	10	-1	-1	-1	-1

Polling & Removals

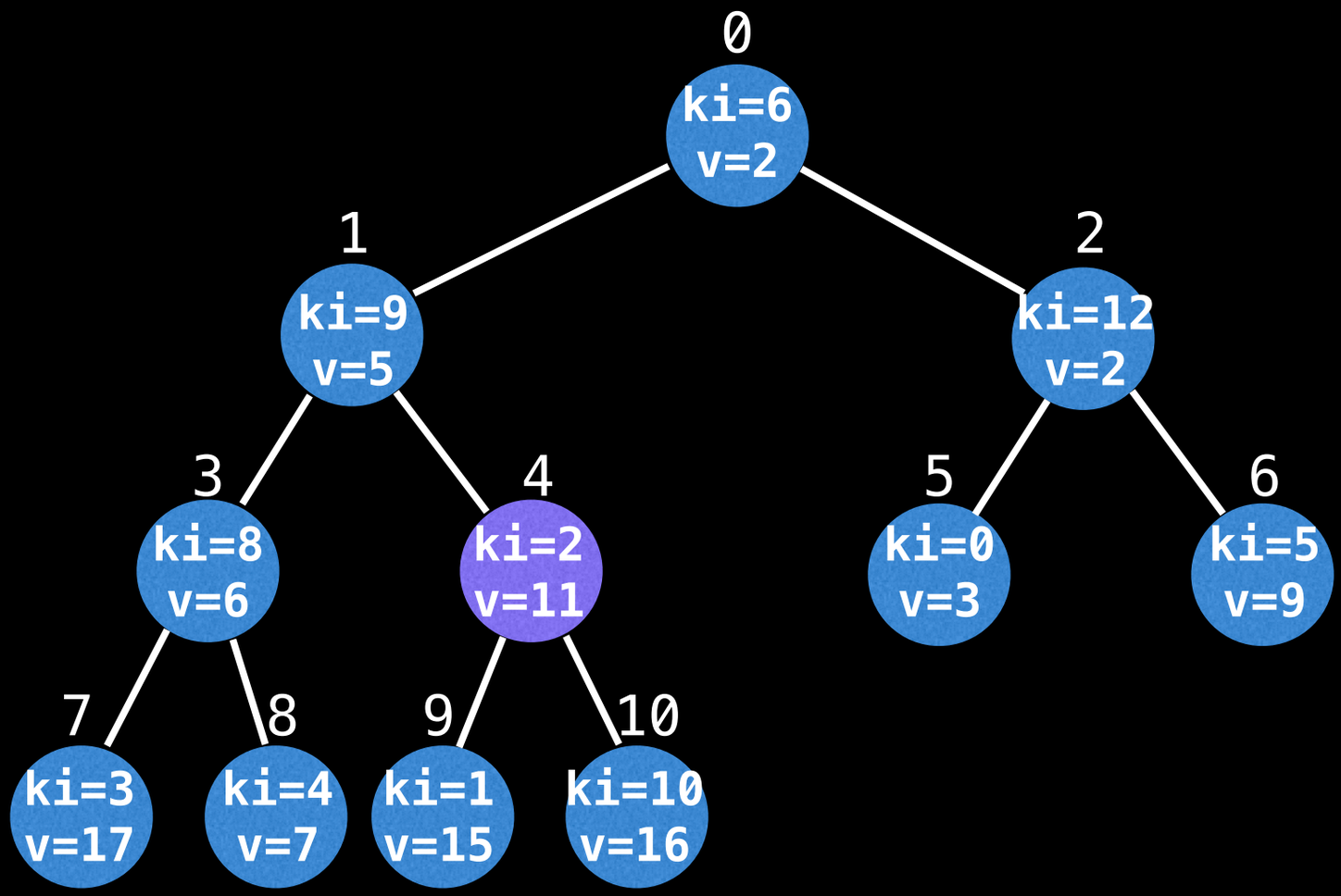


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	1	7	8	6	0	-1	3	4	10	-1	2	-1	-1
im	6	2	12	8	9	0	5	3	4	1	10	-1	-1	-1	-1

Finally restore heap invariant by moving swapped purple node up or down.

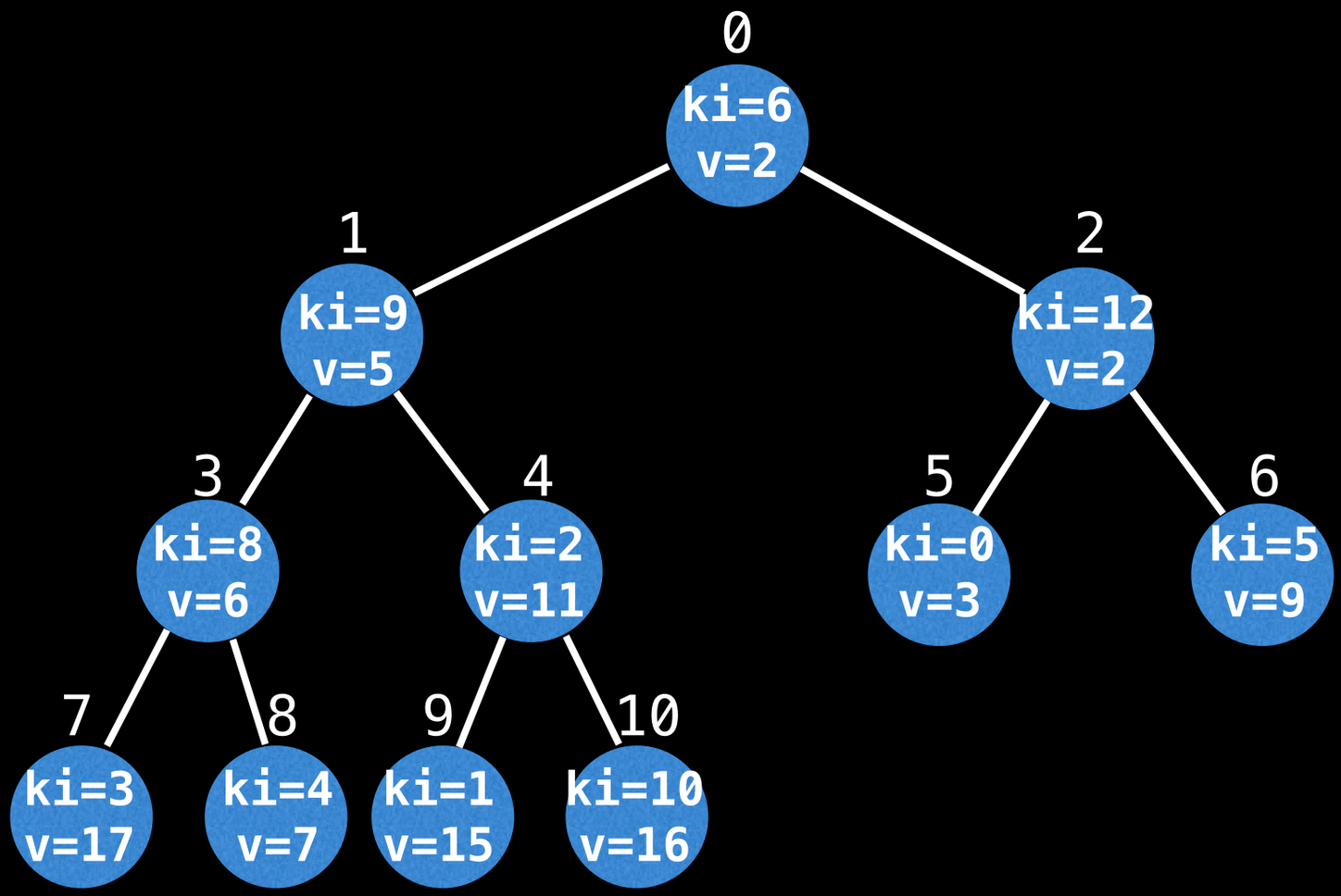
Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	4	7	8	6	0	-1	3	1	10	-1	2	-1	-1
im	6	9	12	8	2	0	5	3	4	1	10	-1	-1	-1	-1

Polling & Removals



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	4	7	8	6	0	-1	3	1	10	-1	2	-1	-1
im	6	9	12	8	2	0	5	3	4	1	10	-1	-1	-1	-1

Removal Pseudo Code

```
# Deletes the node with the key index ki
# in the heap. The key index ki must exist
# and be present in the heap.
function remove(ki):
  i = pm[ki]
  sz = sz - 1
  swap(i, sz)
  sink(i)
  swim(i)
  values[ki] = null
  pm[ki] = -1
  im[sz] = -1
```

Removal Pseudo Code

```
# Deletes the node with the key index ki
# in the heap. The key index ki must exist
# and be present in the heap.
function remove(ki):
    i = pm[ki]
    sz = sz - 1
    swap(i, sz)
    sink(i)
    swim(i)
    values[ki] = null
    pm[ki] = -1
    im[sz] = -1
```

Removal Pseudo Code

```
# Deletes the node with the key index ki
# in the heap. The key index ki must exist
# and be present in the heap.
function remove(ki):
  i = pm[ki]
  sz = sz - 1
  swap(i, sz)
  sink(i)
  swim(i)
  values[ki] = null
  pm[ki] = -1
  im[sz] = -1
```

Removal Pseudo Code

```
# Deletes the node with the key index ki
# in the heap. The key index ki must exist
# and be present in the heap.
function remove(ki):
  i = pm[ki]
  sz = sz - 1
  swap(i, sz)
  sink(i)
  swim(i)
  values[ki] = null
  pm[ki] = -1
  im[sz] = -1
```

Sink Pseudo Code

```
# Sinks the node at index i by swapping
# itself with the smallest of the left
# or the right child node.
function sink(i):
  while true:
    left  = 2*i + 1
    right = 2*i + 2
    smallest = left

    if right < sz and less(right, left):
      smallest = right

    if left >= sz or less(i, smallest):
      break

    swap(smallest, i)
    i = smallest
```

Sink Pseudo Code

```
# Sinks the node at index i by swapping
# itself with the smallest of the left
# or the right child node.
function sink(i):
  while true:
    left  = 2*i + 1
    right = 2*i + 2
    smallest = left

    if right < sz and less(right, left):
      smallest = right

    if left >= sz or less(i, smallest):
      break

    swap(smallest, i)
    i = smallest
```

Sink Pseudo Code

```
# Sinks the node at index i by swapping
# itself with the smallest of the left
# or the right child node.
function sink(i):
  while true:
    left  = 2*i + 1
    right = 2*i + 2
    smallest = left

    if right < sz and less(right, left):
      smallest = right

    if left >= sz or less(i, smallest):
      break

    swap(smallest, i)
    i = smallest
```

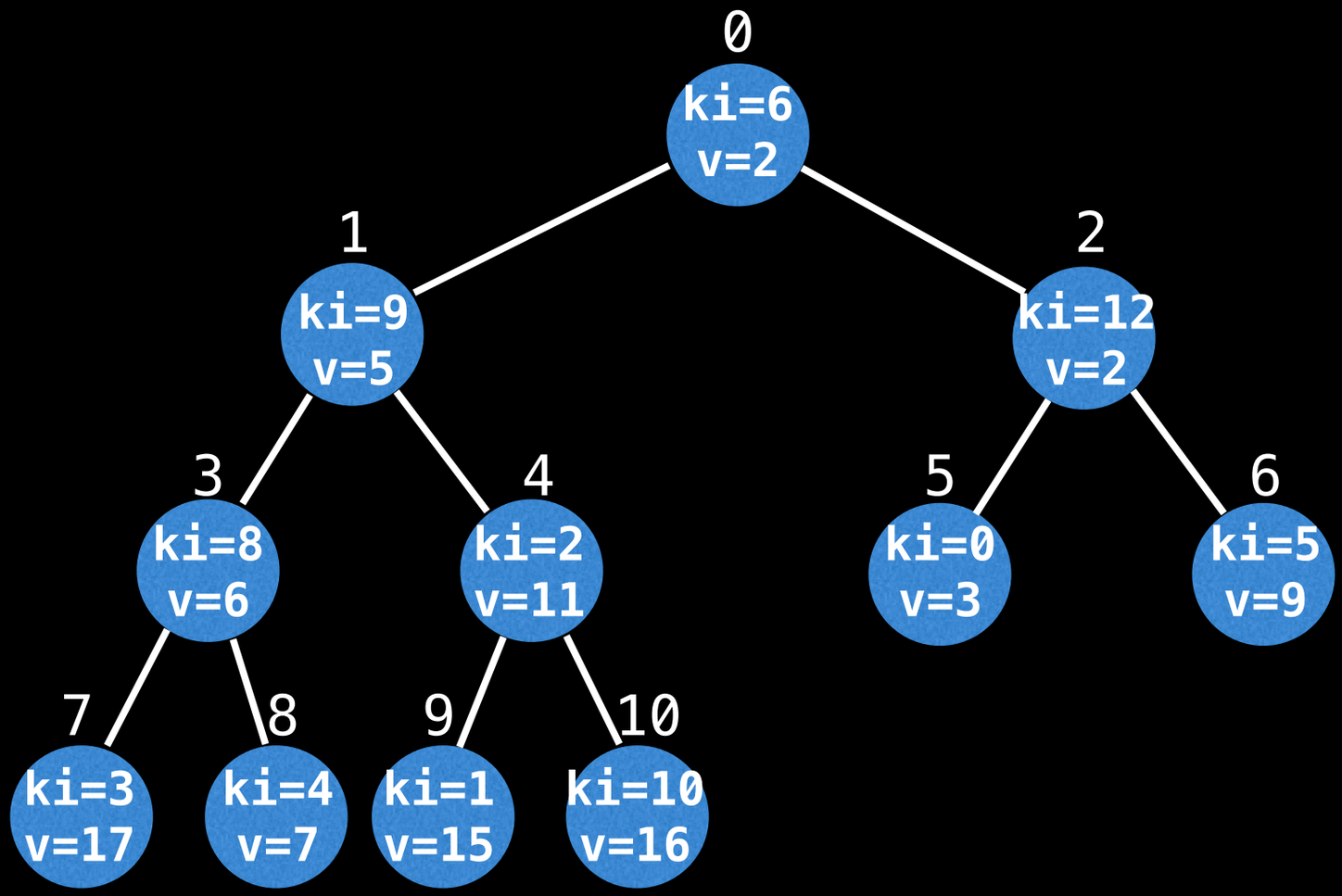
Sink Pseudo Code

```
# Sinks the node at index i by swapping
# itself with the smallest of the left
# or the right child node.
function sink(i):
  while true:
    left  = 2*i + 1
    right = 2*i + 2
    smallest = left

    if right < sz and less(right, left):
      smallest = right

    if left >= sz or less(i, smallest):
      break

    swap(smallest, i)
    i = smallest
```

Sink Pseudo Code

```
# Sinks the node at index i by swapping
# itself with the smallest of the left
# or the right child node.
function sink(i):
  while true:
    left  = 2*i + 1
    right = 2*i + 2
    smallest = left

    if right < sz and less(right, left):
      smallest = right

    if left >= sz or less(i, smallest):
      break

    swap(smallest, i)
    i = smallest
```

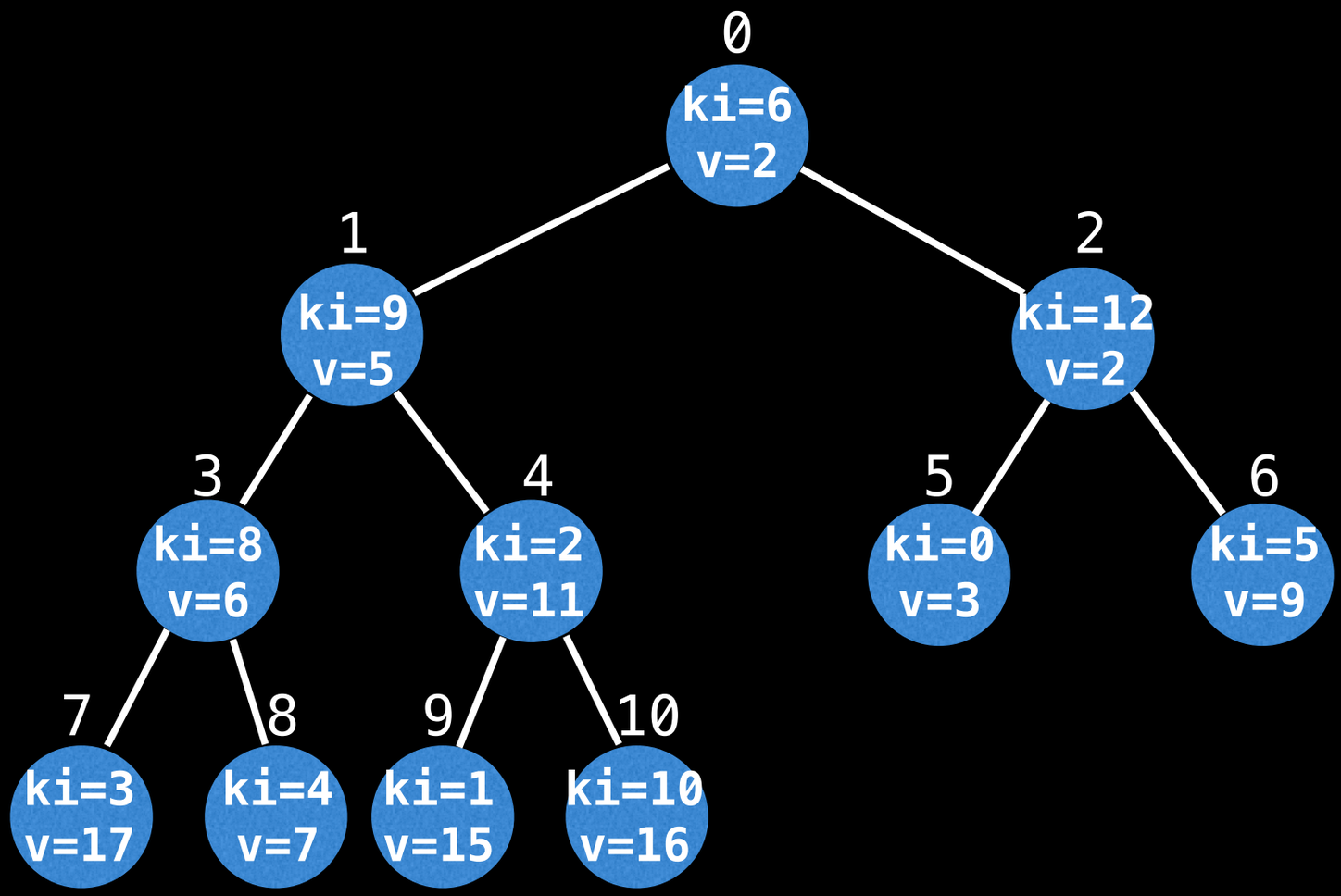
Updates



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

Similar to removals, updates in a min indexed binary heap also take **O(log(n))** due to **O(1)** lookup time to find the node and **O(log(n))** time to adjust where the key-value pair should appear in the heap.

Updates

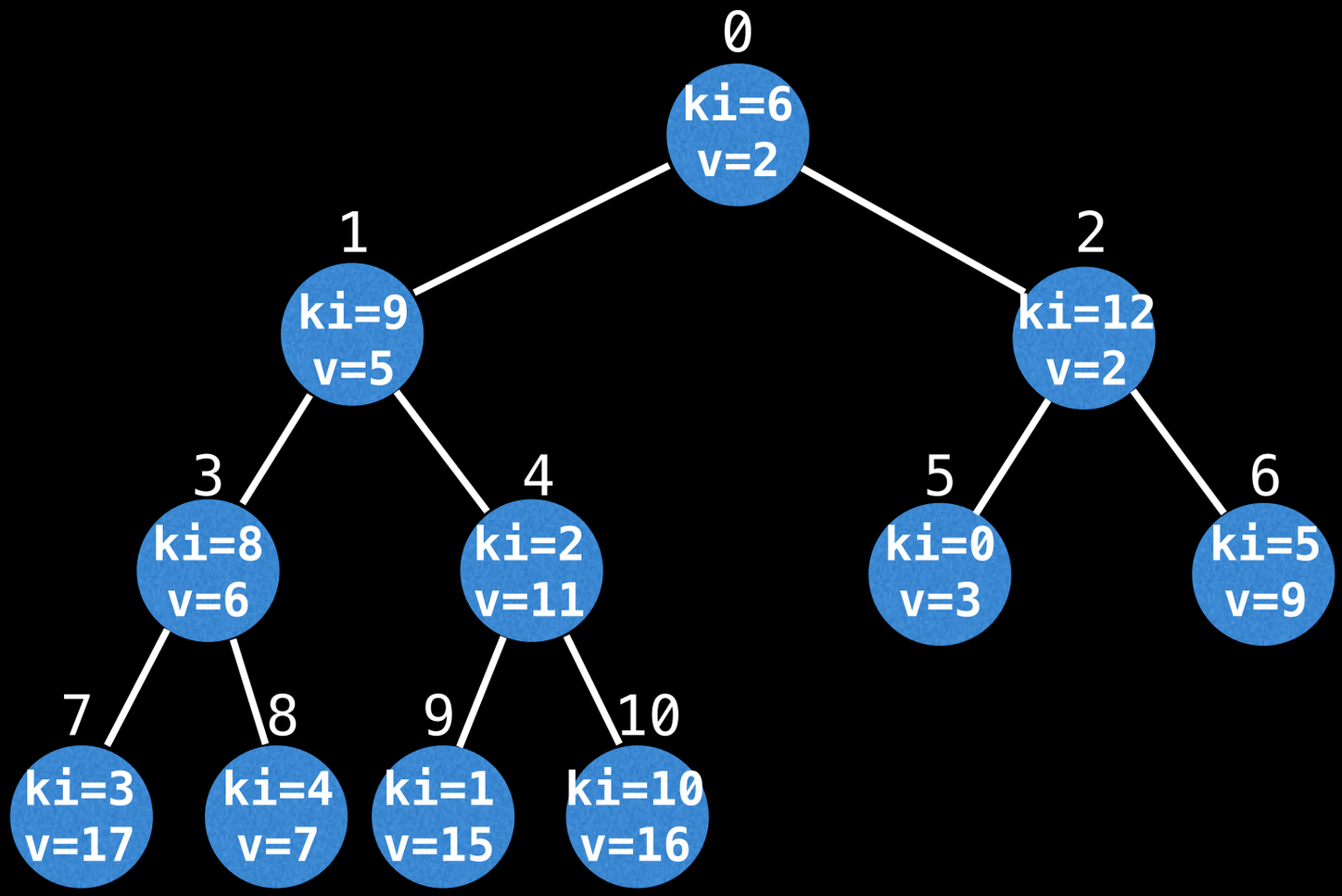


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	4	7	8	6	0	-1	3	1	10	-1	2	-1	-1
im	6	9	12	8	2	0	5	3	4	1	10	-1	-1	-1	-1

Update “Carly” to have a new value of 1

Updates

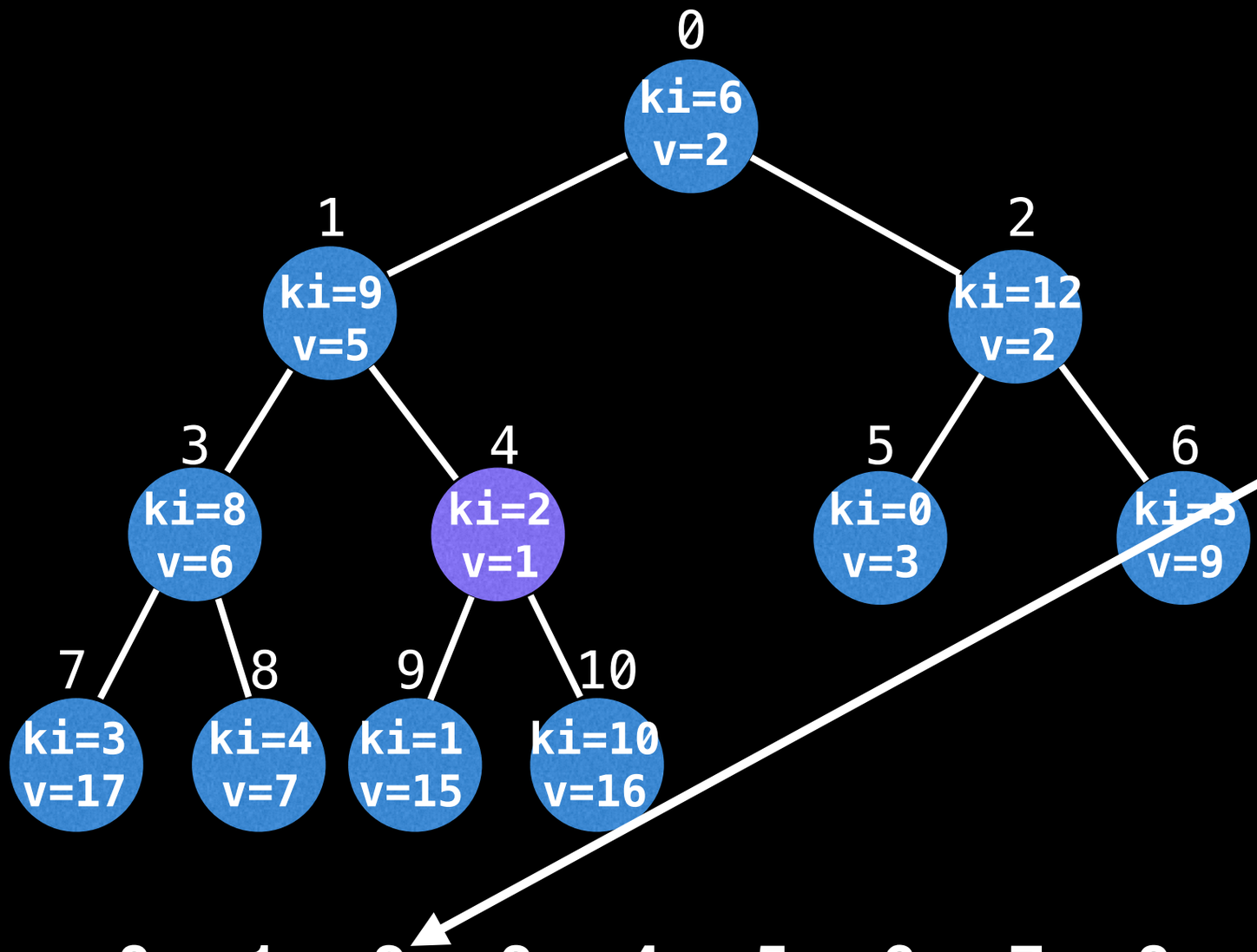


Name	ki
Anna	0
Bella	1
Carly →	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	11	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	4	7	8	6	0	-1	3	1	10	-1	2	-1	-1
im	6	9	12	8	2	0	5	3	4	1	10	-1	-1	-1	-1

Update “Carly” to have a new value of 1

Updates

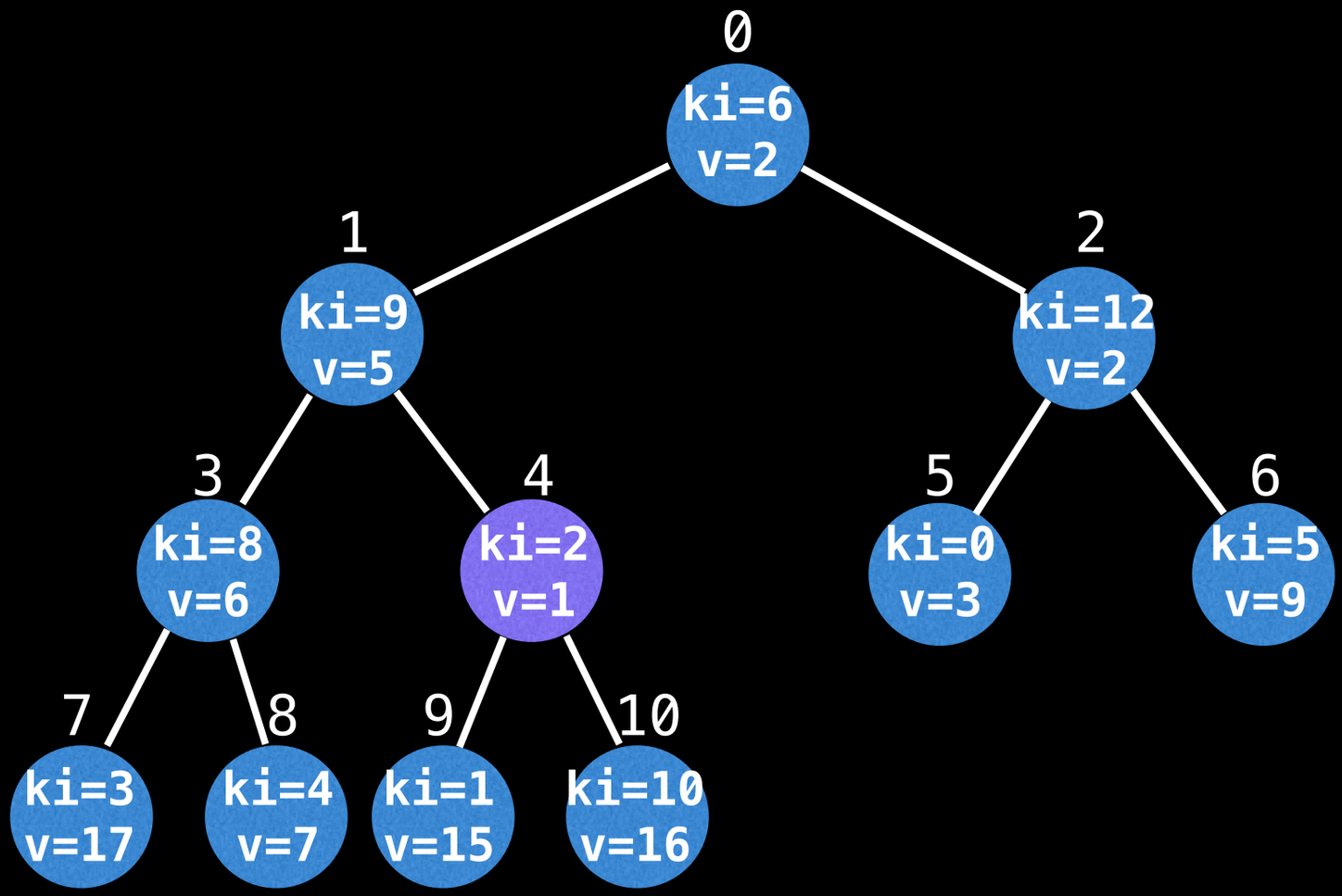


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	1	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	4	7	8	6	0	-1	3	1	10	-1	2	-1	-1
im	6	9	12	8	2	0	5	3	4	1	10	-1	-1	-1	-1

Update values array at ki to have value 1

Updates

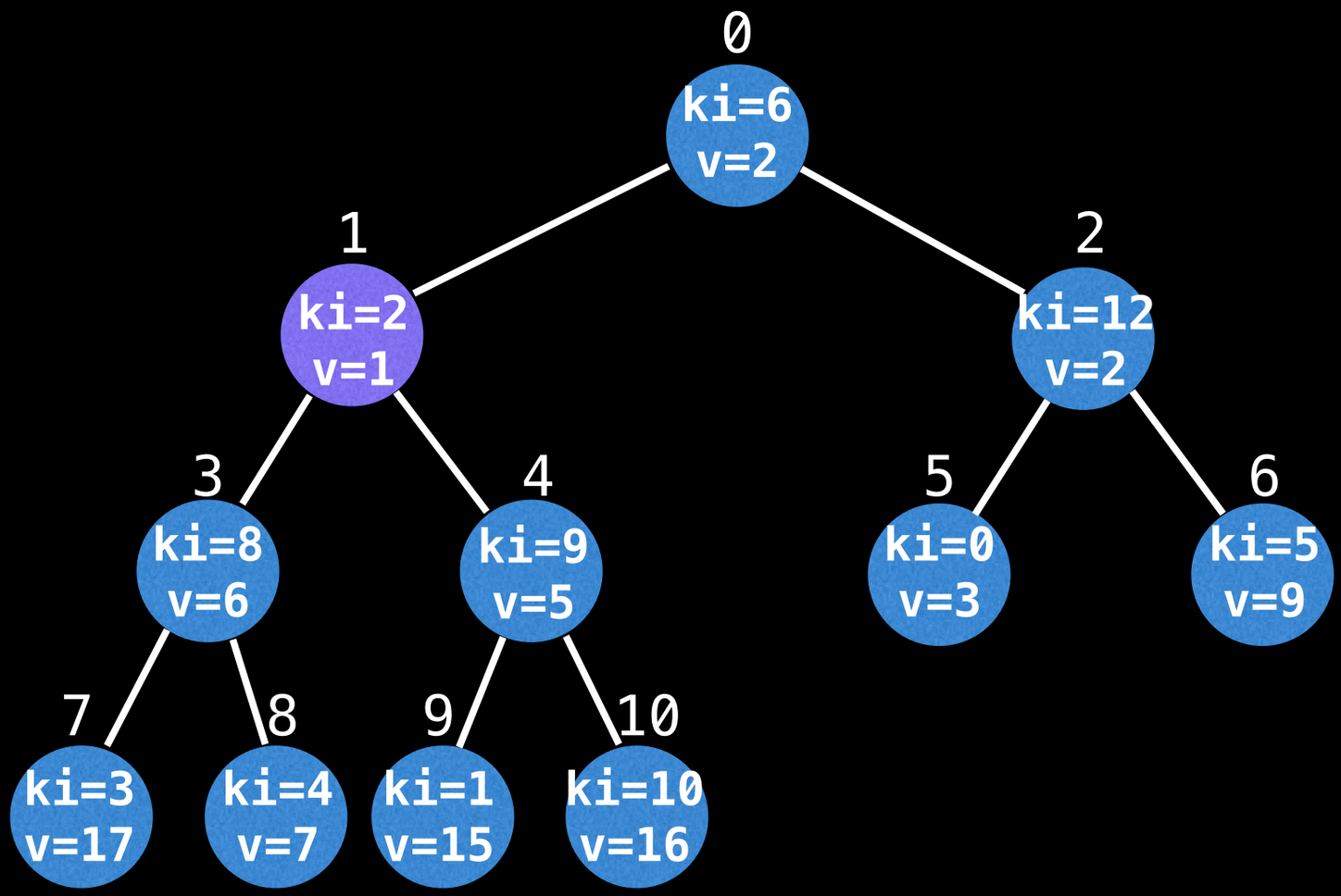


Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	1	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	4	7	8	6	0	-1	3	1	10	-1	2	-1	-1
im	6	9	12	8	2	0	5	3	4	1	10	-1	-1	-1	-1

Finally satisfy heap invariant by moving node either up or down the heap.

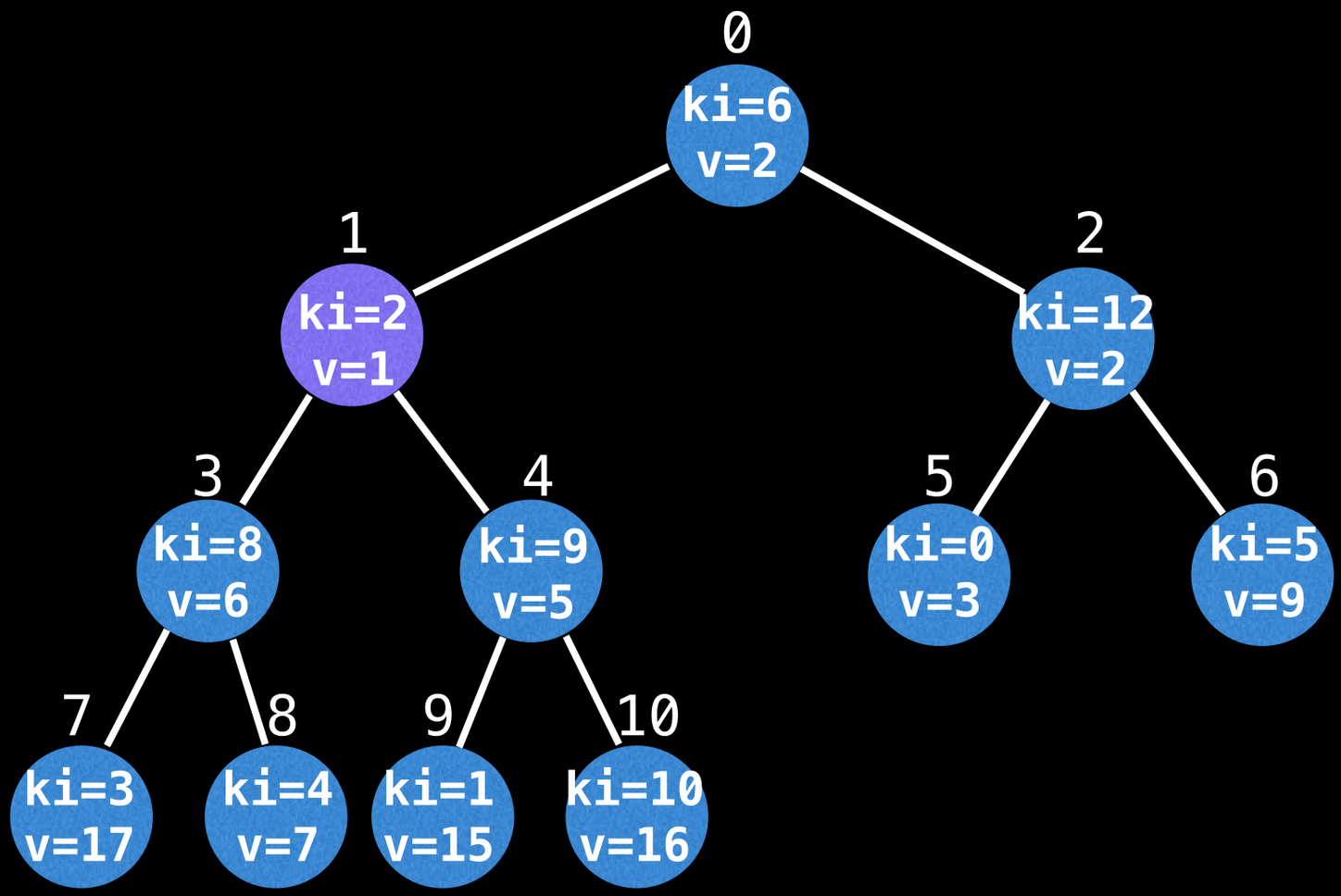
Updates



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	1	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	1	7	8	6	0	-1	3	4	10	-1	2	-1	-1
im	6	2	12	8	9	0	5	3	4	1	10	-1	-1	-1	-1

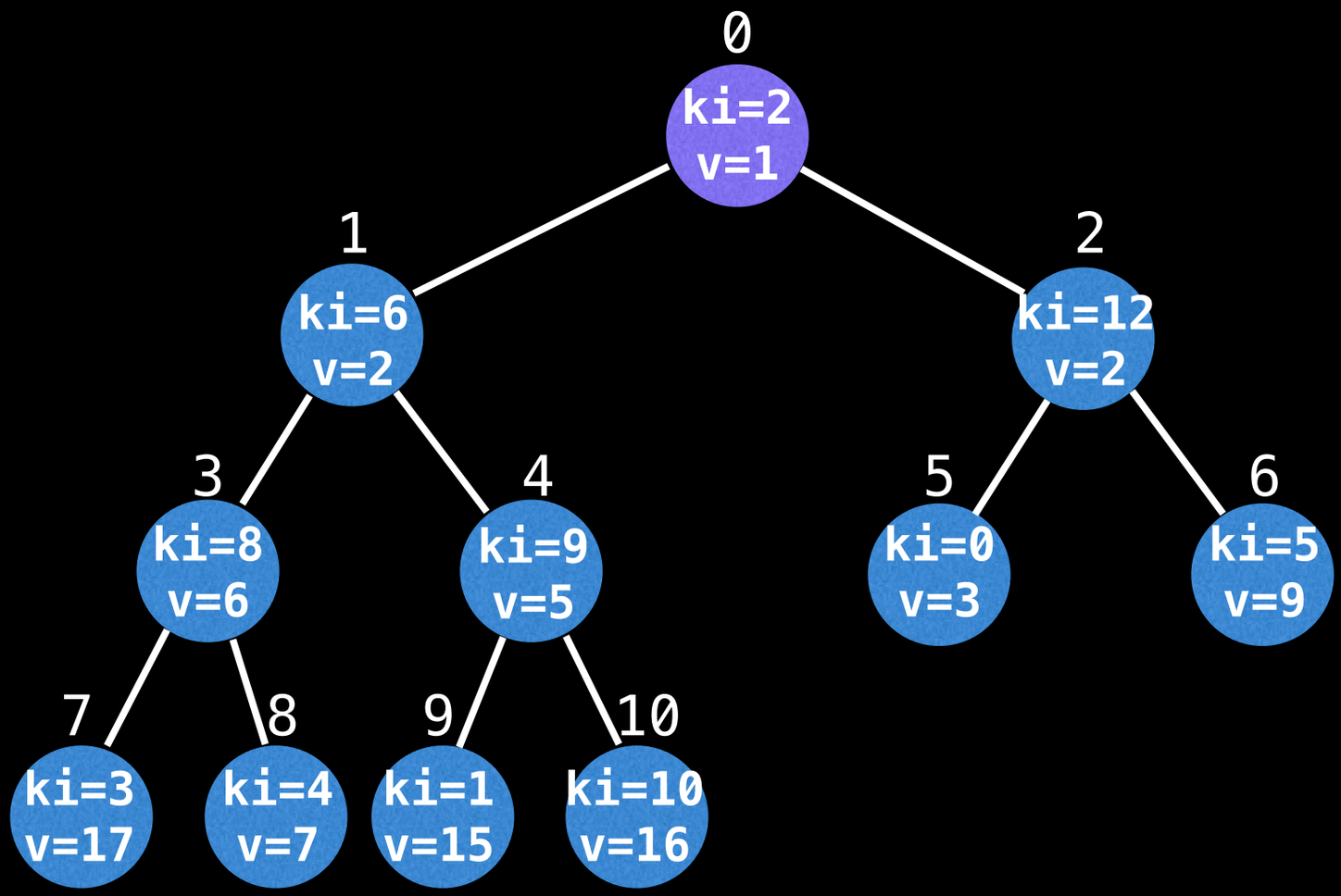
Updates



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	1	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	1	7	8	6	0	-1	3	4	10	-1	2	-1	-1
im	6	2	12	8	9	0	5	3	4	1	10	-1	-1	-1	-1

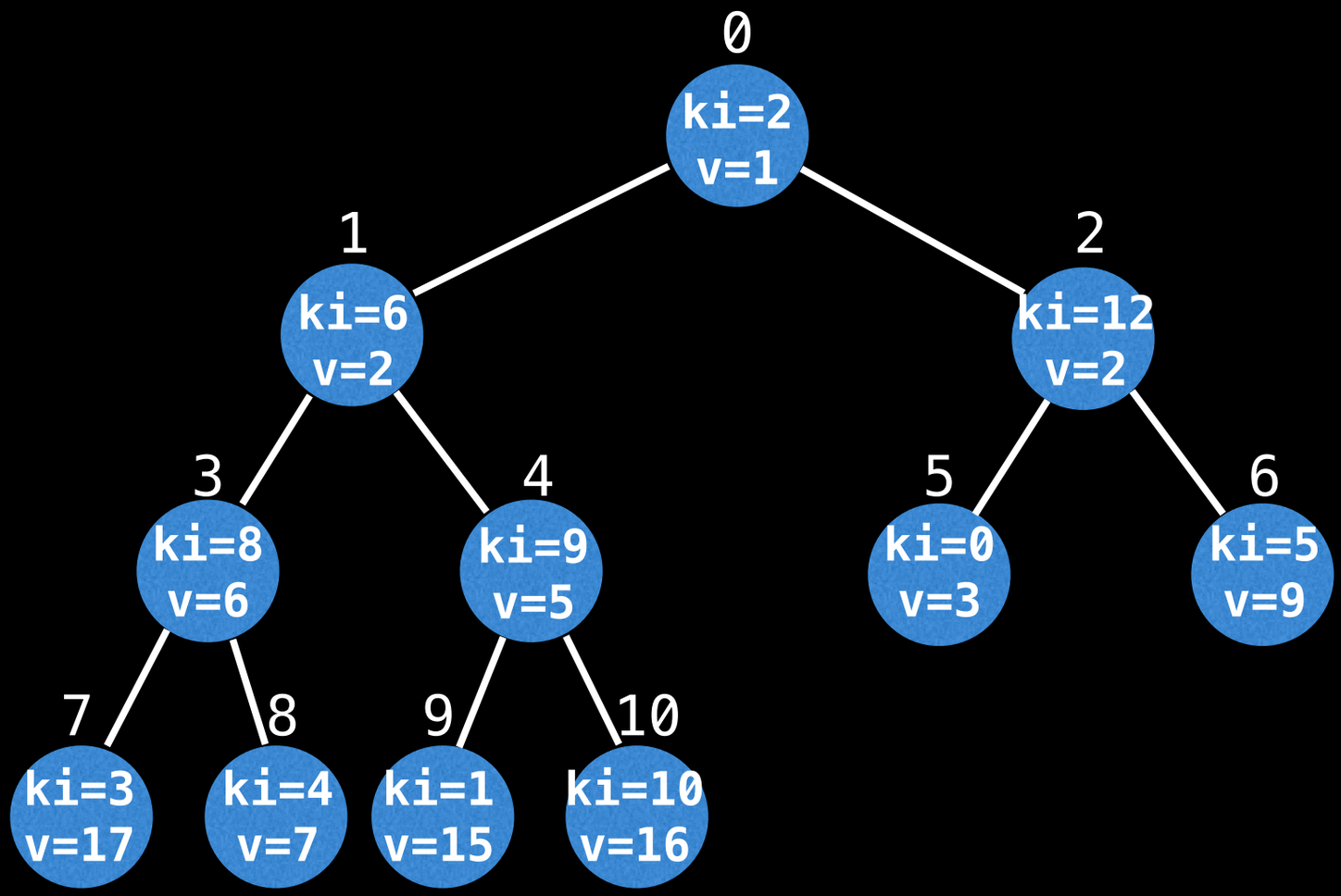
Updates



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	1	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	0	7	8	6	1	-1	3	4	10	-1	2	-1	-1
im	2	6	12	8	9	0	5	3	4	1	10	-1	-1	-1	-1

Updates



Name	ki
Anna	0
Bella	1
Carly	2
Dylan	3
Emily	4
Fred	5
George	6
Henry	7
Isaac	8
James	9
Kelly	10
Laura	11
Mary	12

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
vals	3	15	1	17	7	9	2	∅	6	5	16	∅	2	-1	-1
pm	5	9	0	7	8	6	1	-1	3	4	10	-1	2	-1	-1
im	2	6	12	8	9	0	5	3	4	1	10	-1	-1	-1	-1

Update Pseudo Code

```
# Updates the value of a key in the binary
# heap. The key index must exist and the
# value must not be null.
function update(ki, value):
    i = pm[ki]
    values[ki] = value
    sink(i)
    swim(i)
```

Decrease and Increase key

In many applications (e.g Dijkstra's and Prims algorithm) it is often useful to only update a given key to make its value either always smaller (or larger). In the event that a worse value is given the value in the IPQ should not be updated.

In such situations it is useful to define a more restrictive form of update operation we call **increaseKey**(ki, v) and **decreaseKey**(ki, v)

Increase/Decrease key pseudo code

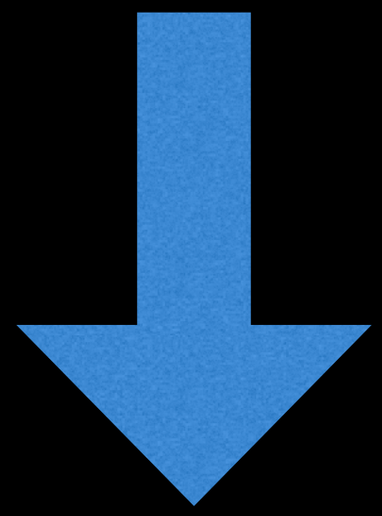
```
# For both these functions assume ki and value
# are valid inputs and we are dealing with a
# min indexed binary heap.
function decreaseKey(ki, value):
  if less(value, values[ki]):
    values[ki] = value
    swim(pm[ki])


function increaseKey(ki, value):
  if less(values[ki], value):
    values[ki] = value
    sink(pm[ki])
```

Source Code Link

Implementation **source code** and **slides** can be found at the following link:

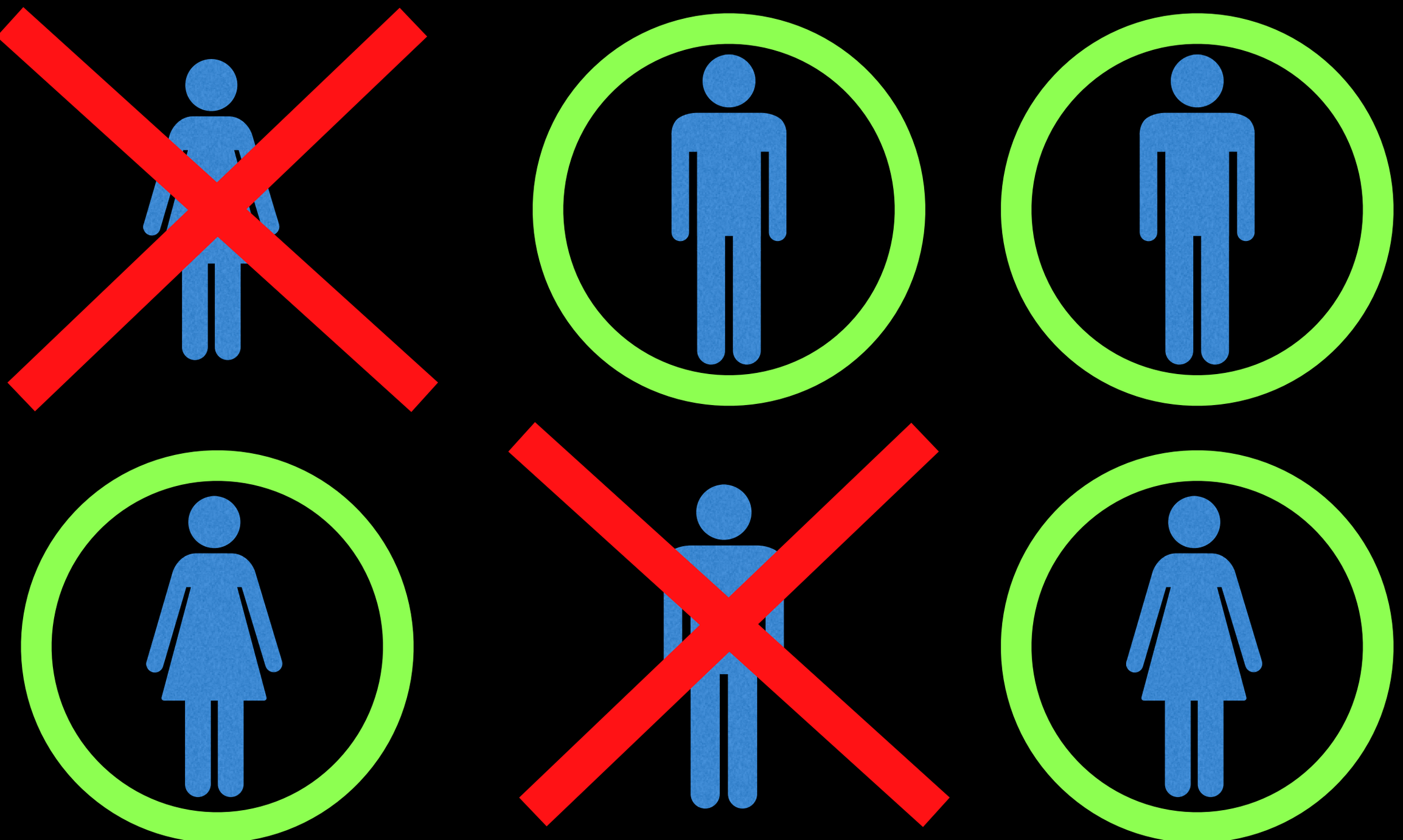
github.com/williamfiset/data-structures

Link in the description:

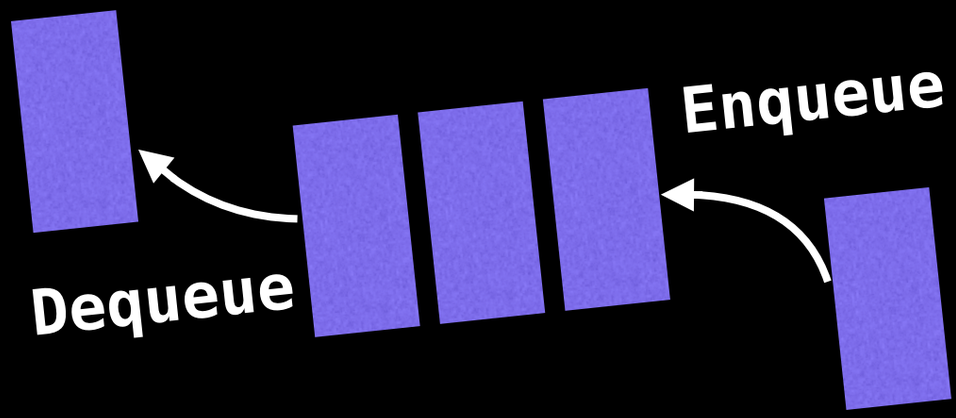


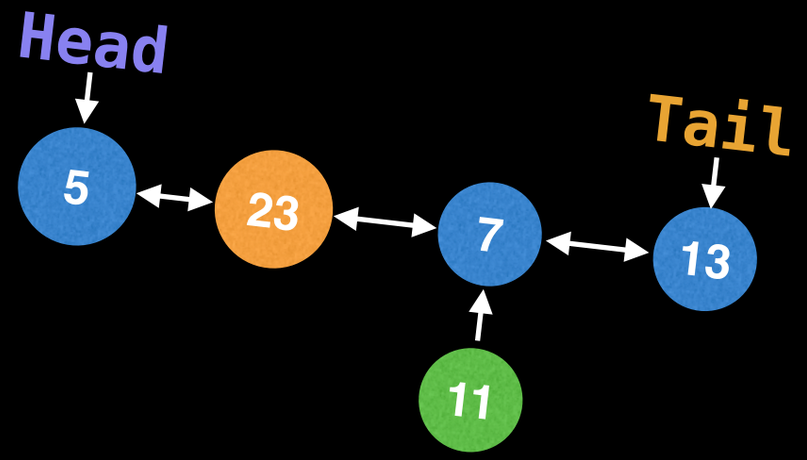
Next Video: IPQ source code

Indexed Priority Queues (UPDATED)

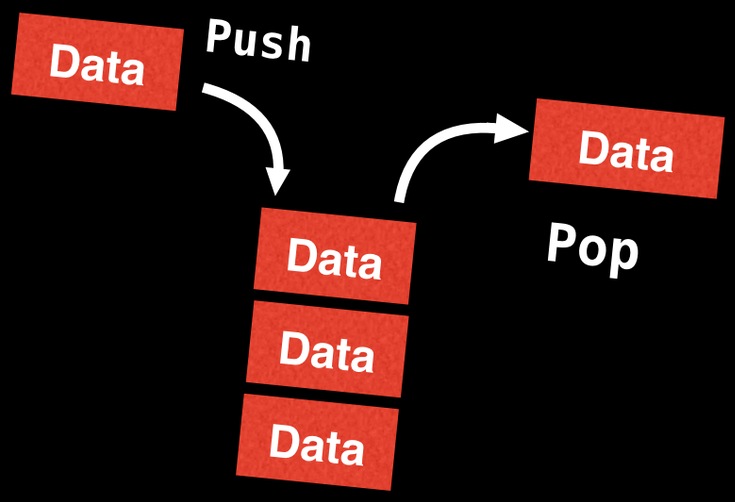


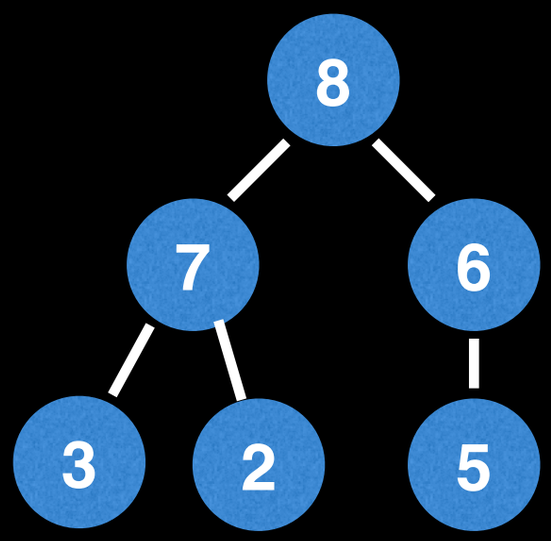
Indexed PQ Source Code

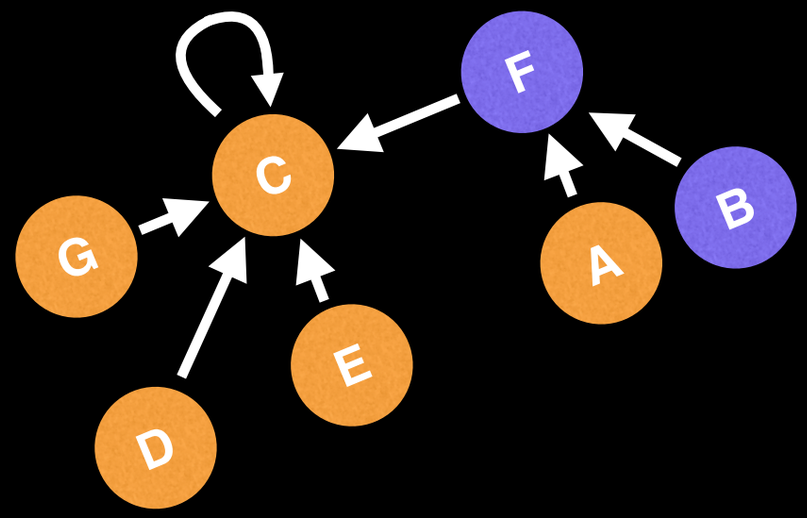




Data Structure Video Series







Indexed Priority Queue Source Code

William Fiset

Source Code Link

Implementation **source code** and **slides** can be found at the following link:

github.com/williamfiset/data-structures

Link in the description: