The Heap


Coca-Cola
Estrella Damm
movistar
LA VANGUARDIA
WE WANT TO VOTE!
www.jnc.cat
REPSOL
BASF
TARRAGONA T'ESPERA
TOTS A PLAÇA DE NOU

The Priority Queue ADT

From previous lecture

add(key, value)	add a new element into the priority queue
min()	return the value with the minimum key
remove_min()	remove and return the value with the minimum key
length()	return the number of items in the priority queue

No commitment to any particular organisation of the data underneath

Priority queue: implementation complexity

From previous lecture

	add(k,v)	min()	remove_min()	length()	build full PQ
unsorted list	O(1)* append(E(k,v))	O(n)	O(n)	O(1)	O(n)
unsorted DLL	O(1) add at end	O(n)	O(n)	O(1)	O(n)
sorted list	O(n)	O(1)	O(1)* min at end	O(1)	$O(n^2)$
sorted DLL	O(n)	O(1)	O(1)	O(1)	$O(n^2)$
AVL tree	O(log n)	O(log n)	O(log n)	O(1)	O(n log n)

can we do any better?

Priority queue: informal analysis

- Keep the complexity of any operation to no worse than O(log n)
- Keep the complexity of min() to O(1), since it is just a reporting method, requiring no change to any data
- Prefer array-based representation over linked structure, but binary trees give us a O(log n) bound each individual operation.
- Build the initial structure efficiently

It looks like we need some compromise between a fully sorted structure (which gives low access time) and an unsorted structure (which gives low update time)

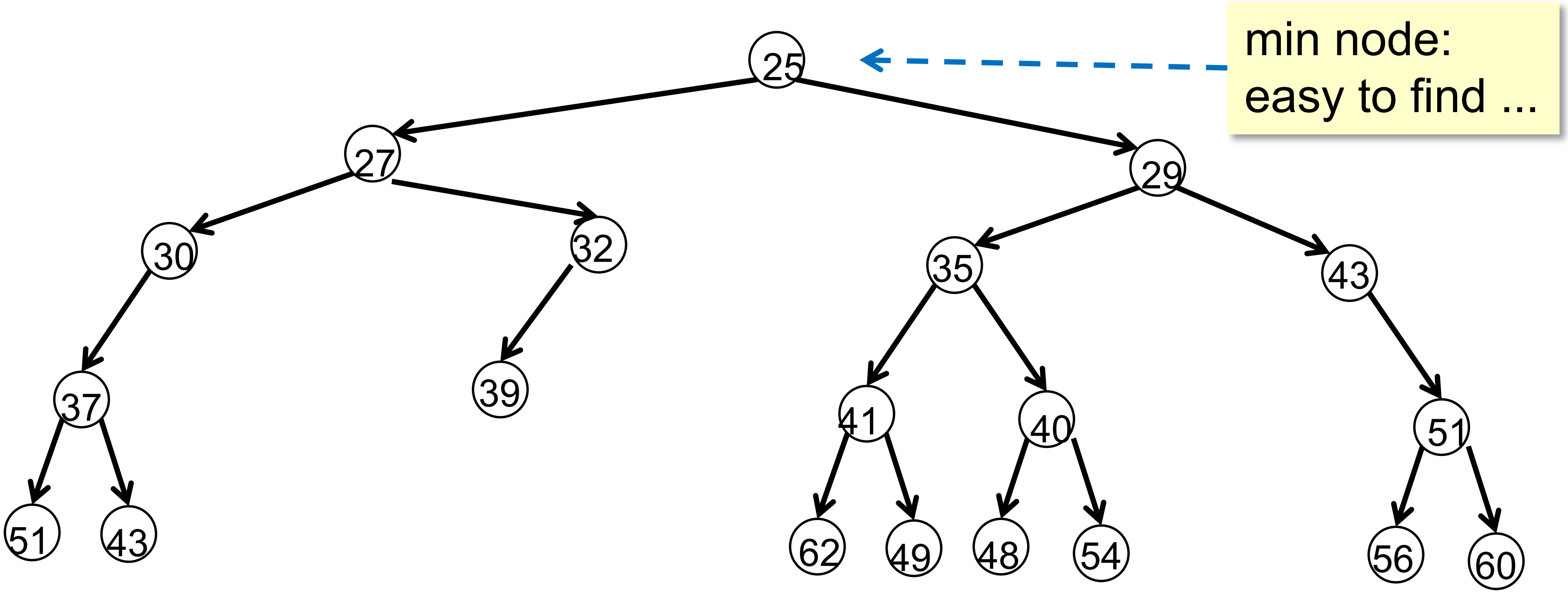
Keep the min key element at the root of the tree?

- each of its children must have a higher key

use that as the recursive definition?

Maintaining the PQ data

A binary tree where every node has lower key than its children?



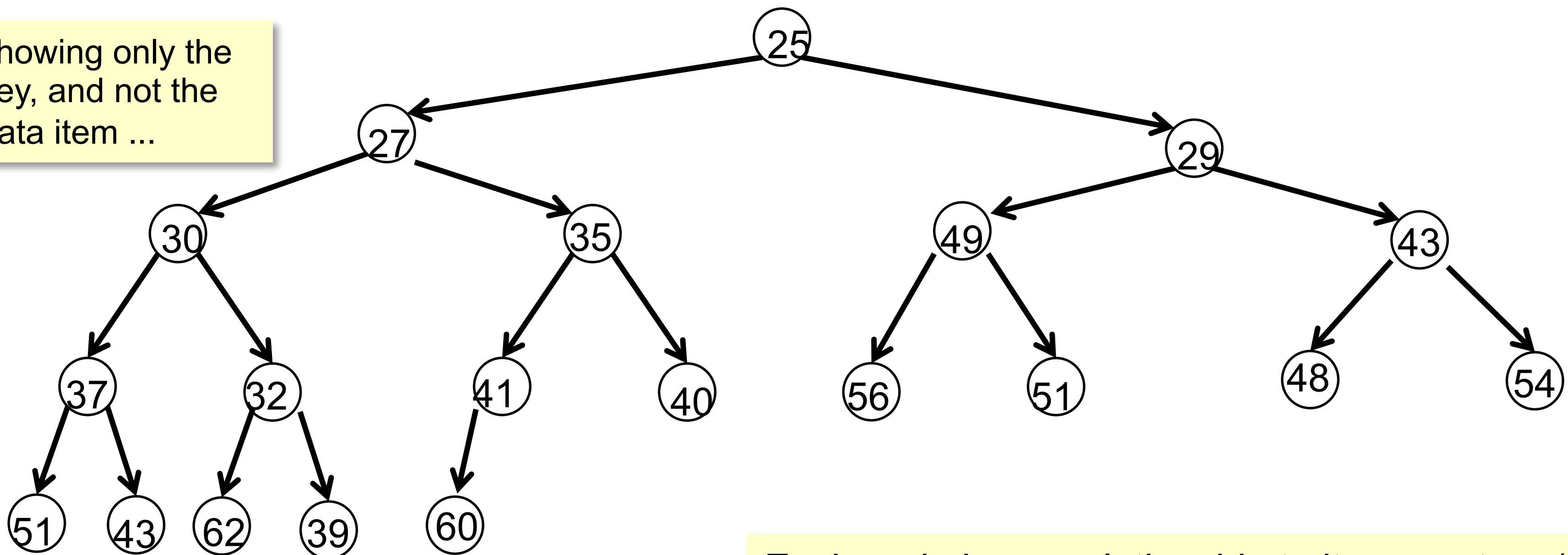
Where do we add a new element? E.g. an element with key 26
How do we rebalance the tree when we remove the top node?

The Binary Heap

A binary tree where:

- every node has lower (or equal) key than its children
- every level (except maybe the last) is complete
- the lowest level, counting from the left, has nodes in every position up to some point, and then no more nodes

showing only the key, and not the data item ...

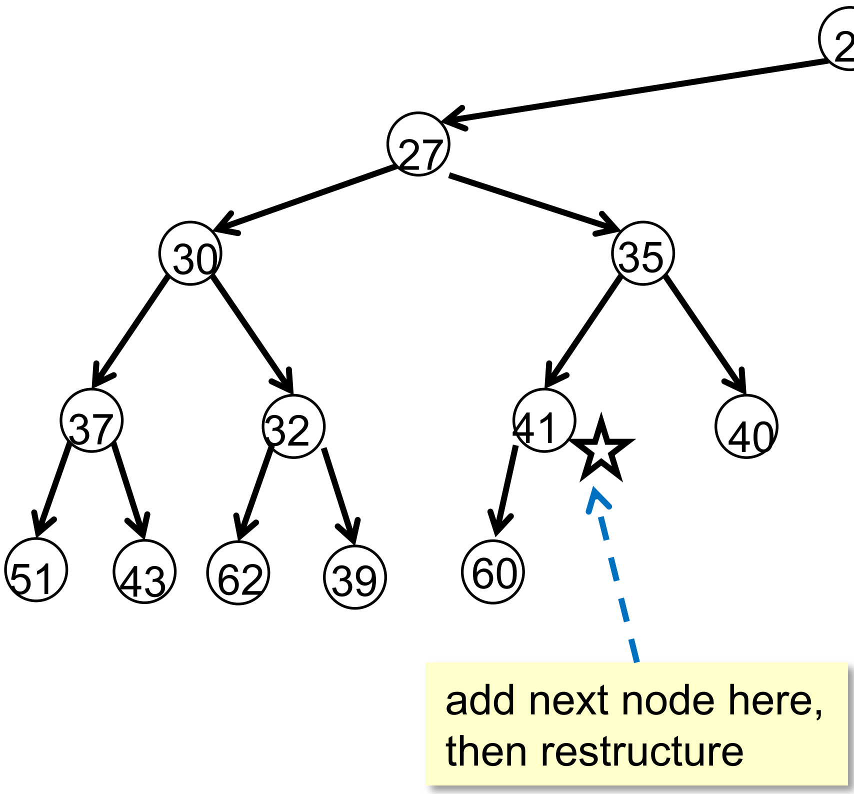


Each node has a relationship to its ancestors (it has higher key), and to its descendants (it has lower key), but no relationship to any other node.

The Binary Heap

A binary tree where:

- every node has lower (or equal) key than its children
- every level (except maybe the last) is complete
- the lowest level, counting from the left, has nodes in every position up to some point, and then no more nodes



Where do we add the next node? Must retain the heap properties.

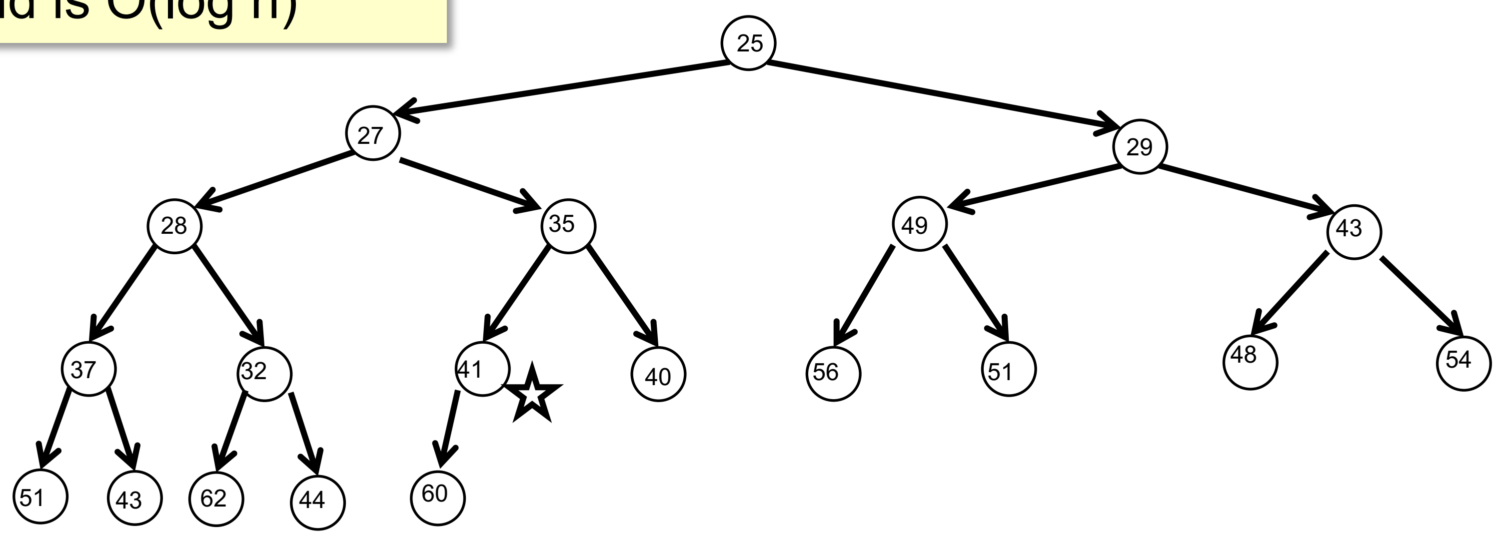
Adding to a binary heap

Create Element object
Add element in last position
Bubble element up heap
Update last position
Update heap size

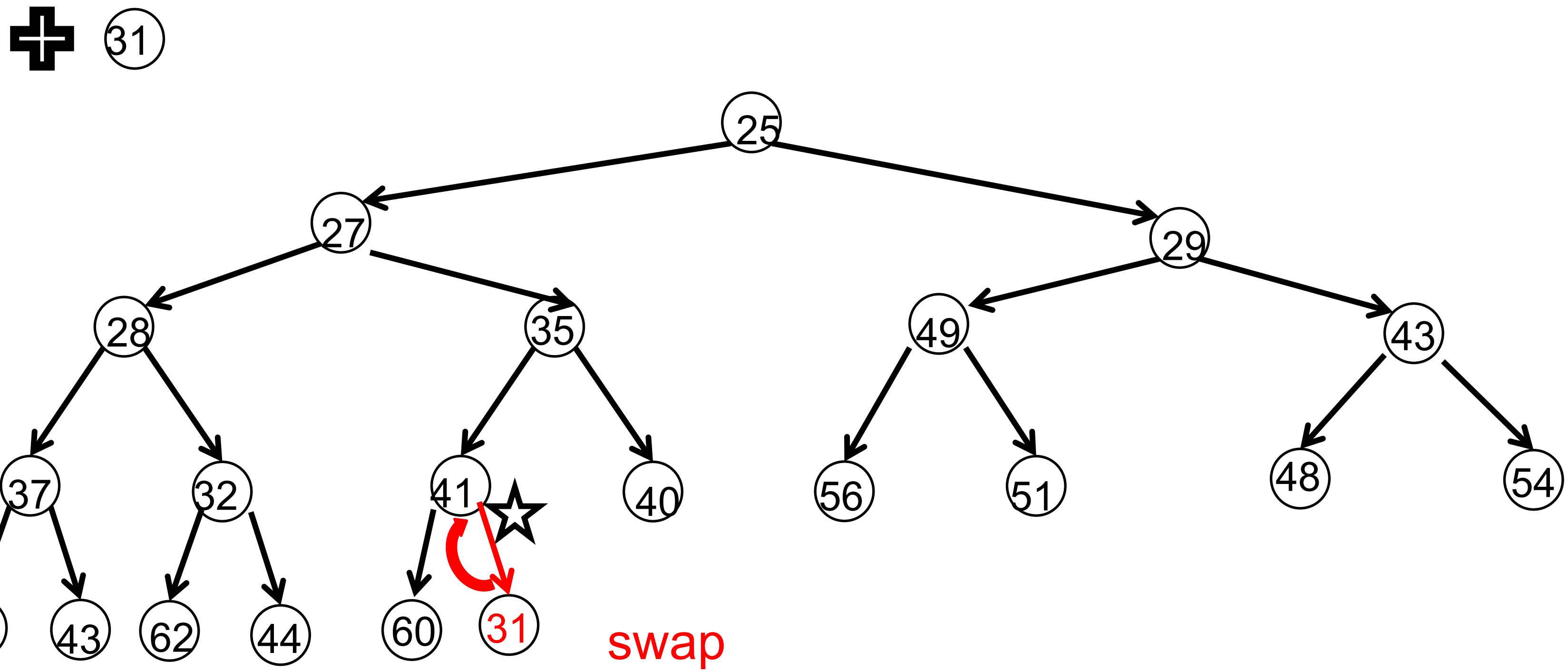
Bubble element up (recursive):
  if key < parent key
    swap element with parent
    bubble parent up heap

Bubble element up (iterative):
  while *this* key < *this*.parent key
    swap *this* and *this*.parent
    set *this* to parent

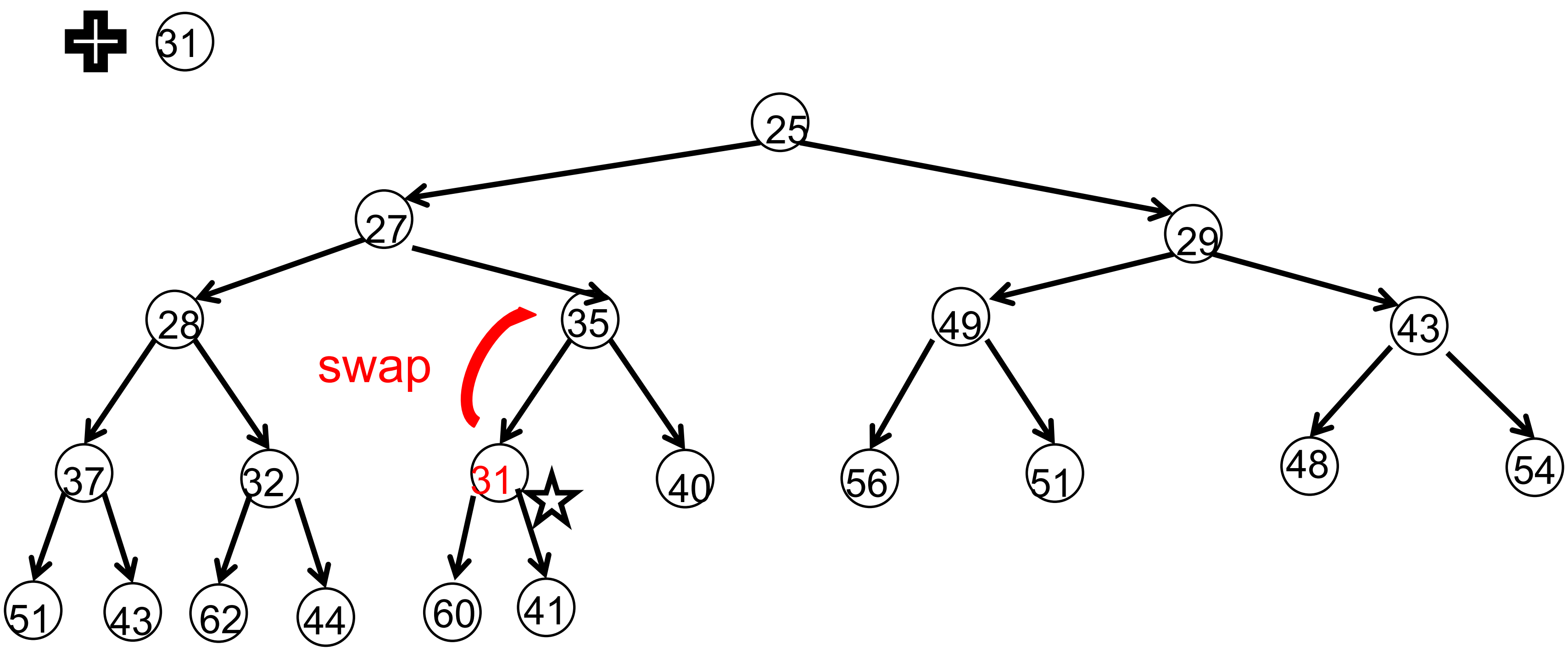
Each swap is O(1)
At most O(log n) swaps
So add is O(log n)



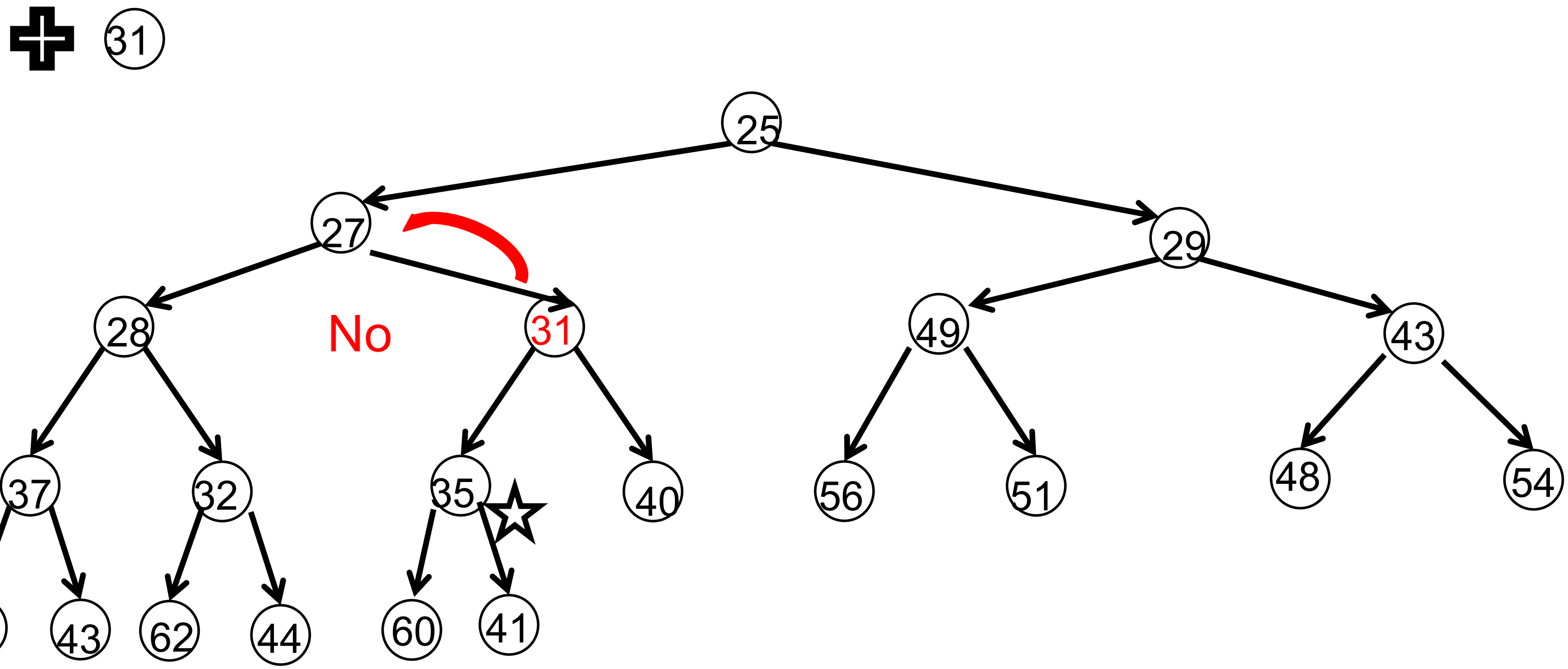
Adding to a binary heap



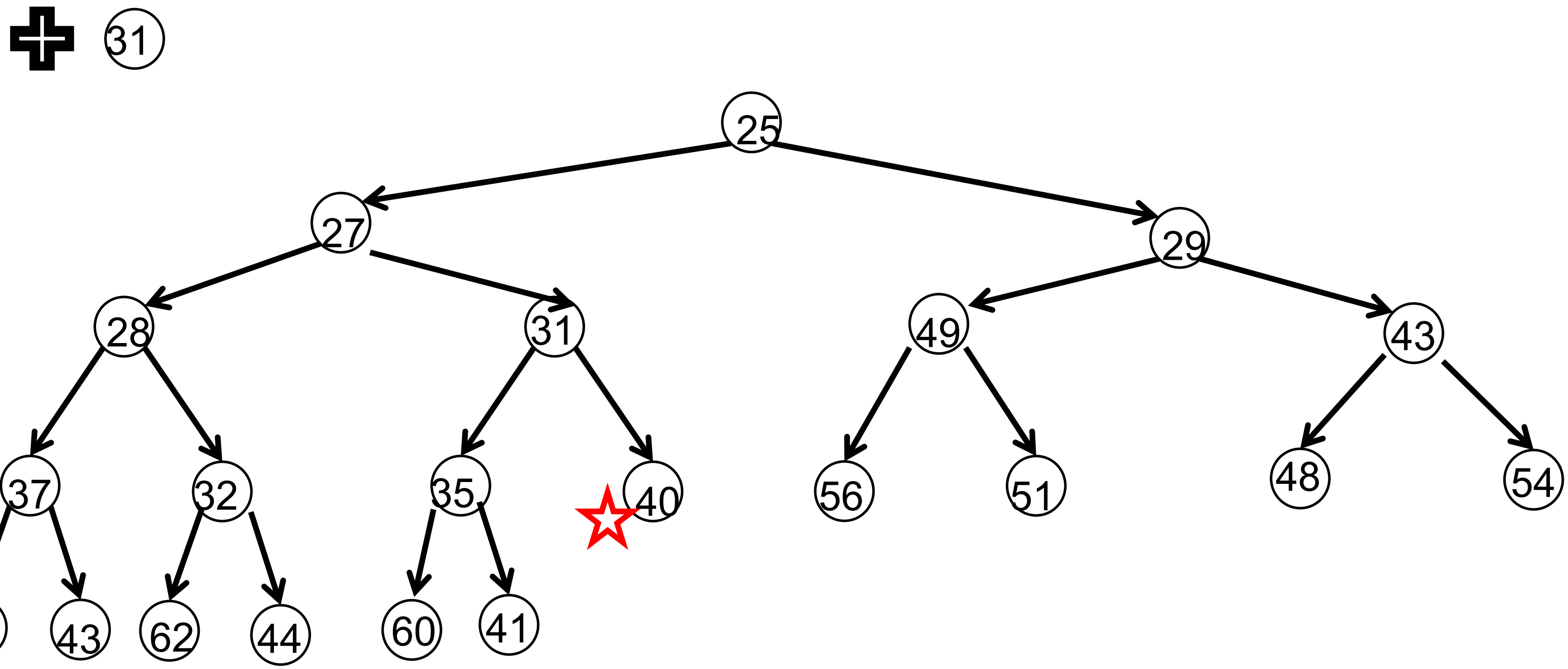
Adding to a binary heap



Adding to a binary heap

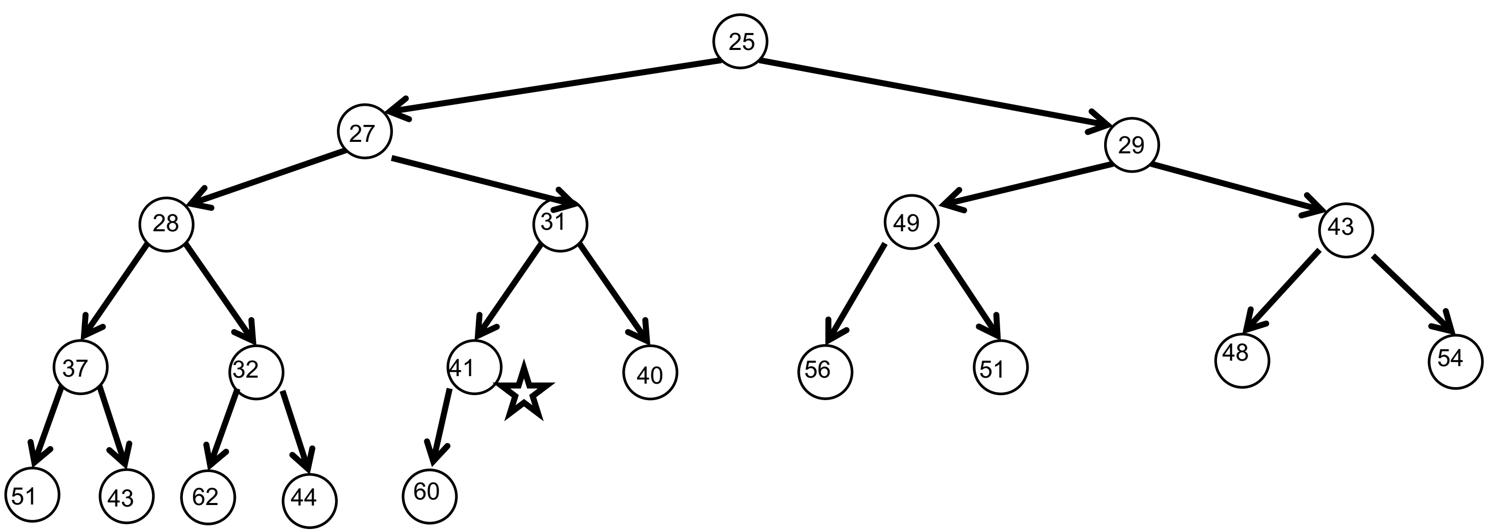


Adding to a binary heap



Removing top from a binary heap

Apply the same reasoning – we know what the tree shape will be, so do the minimal change, then recursively move the key that changed position until the heap is restored?

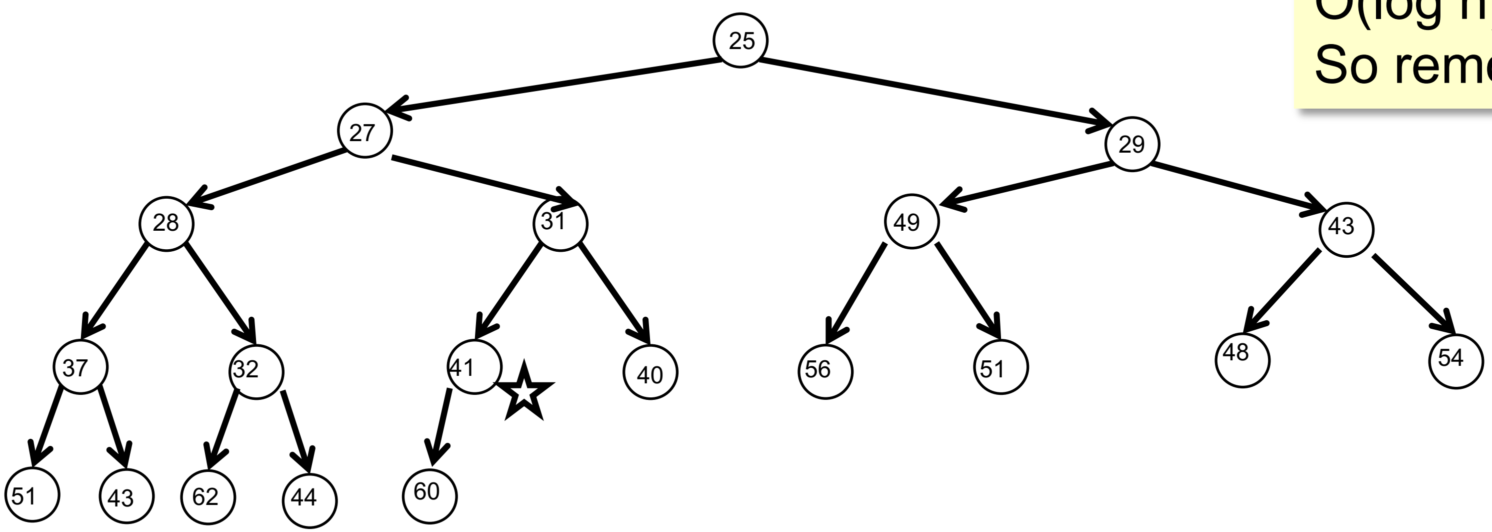


Removing top from a binary heap

Extract the root value
Copy the last element into the root
Remove last node.
Bubble root element down
Update last position
Update size

Bubble element down (recursive):
    if children
        target = child with min key
        if this key > target key
            swap element with target
            bubble element down heap

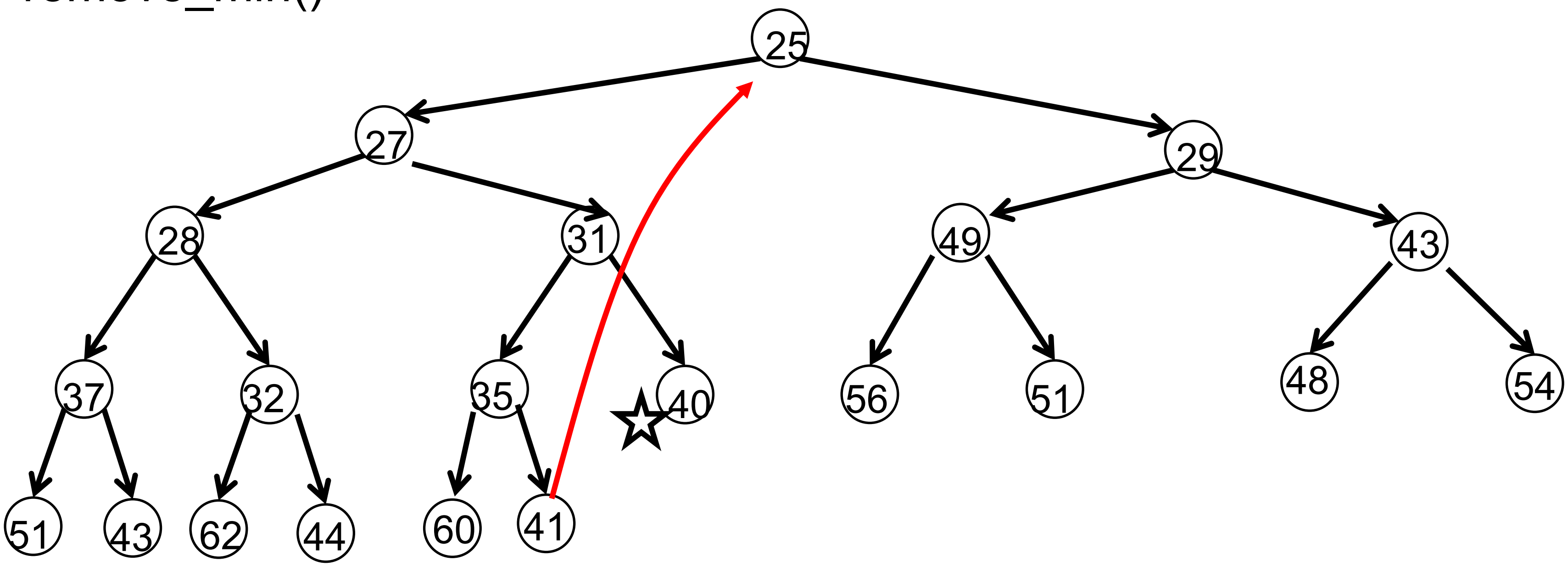
Finding min key is O(1)
Swap is O(1)
O(log n) swaps
So remove is O(log n)



Removing top from a binary heap

(25,*value*)

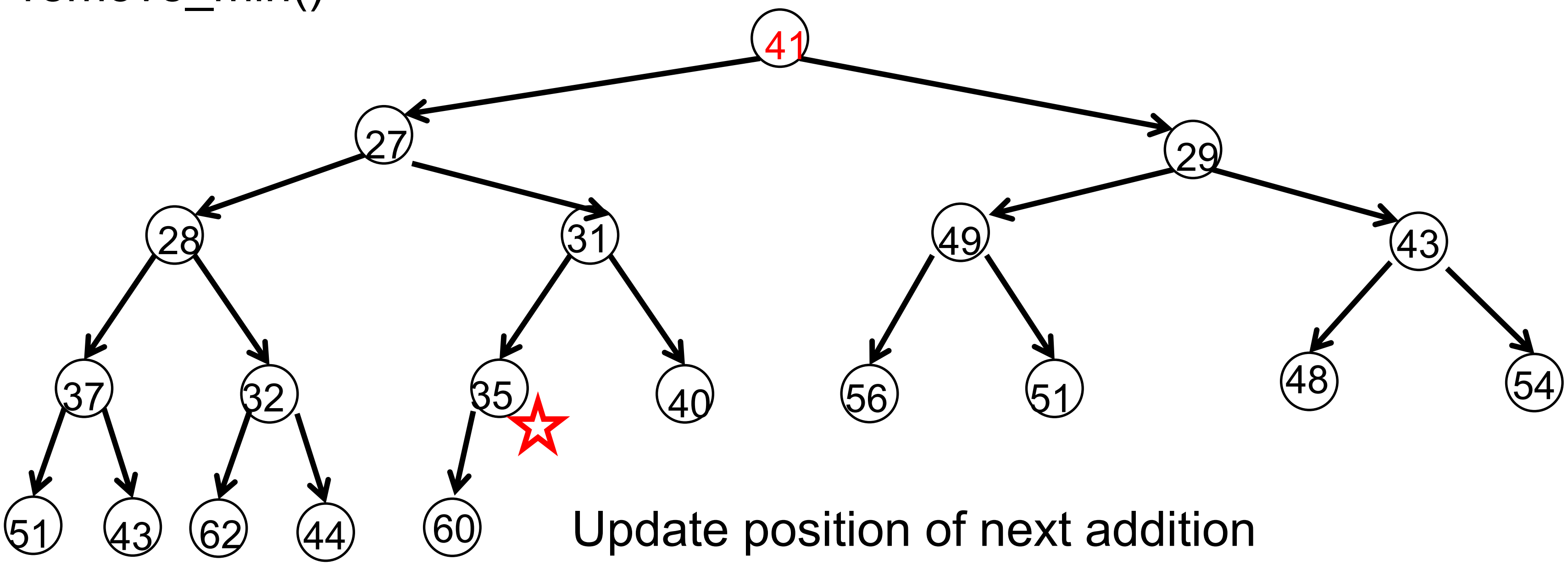
remove_min()



Removing top from a binary heap

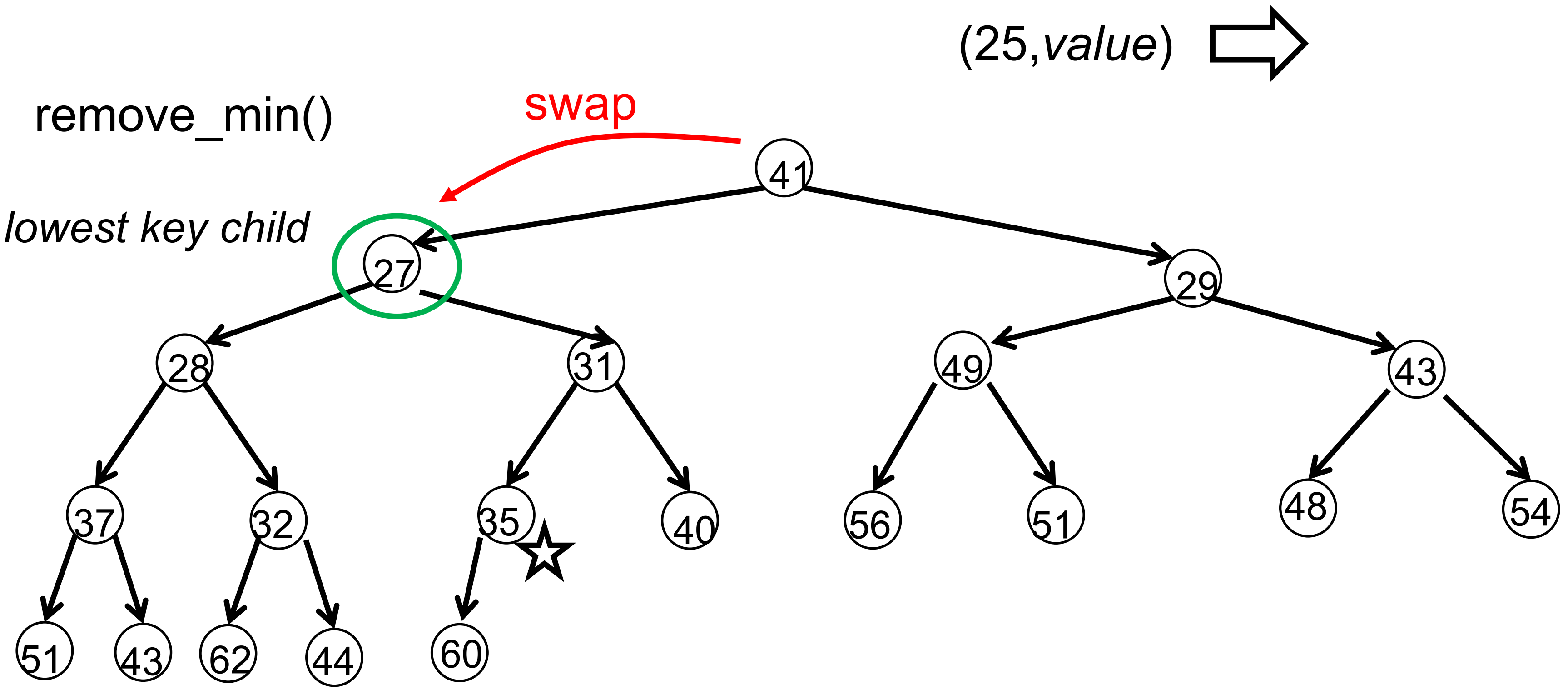
(25,*value*)

remove_min()

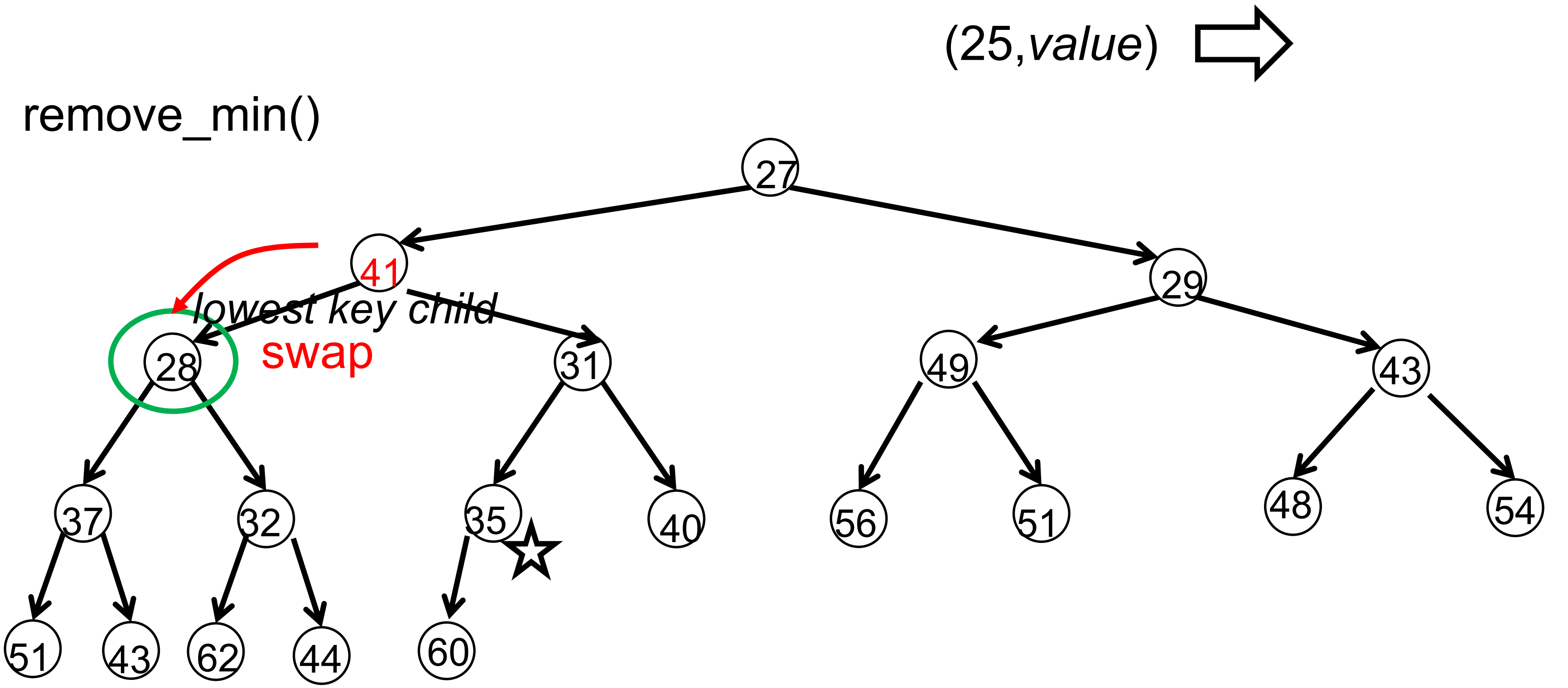


Update position of next addition

Removing top from a binary heap



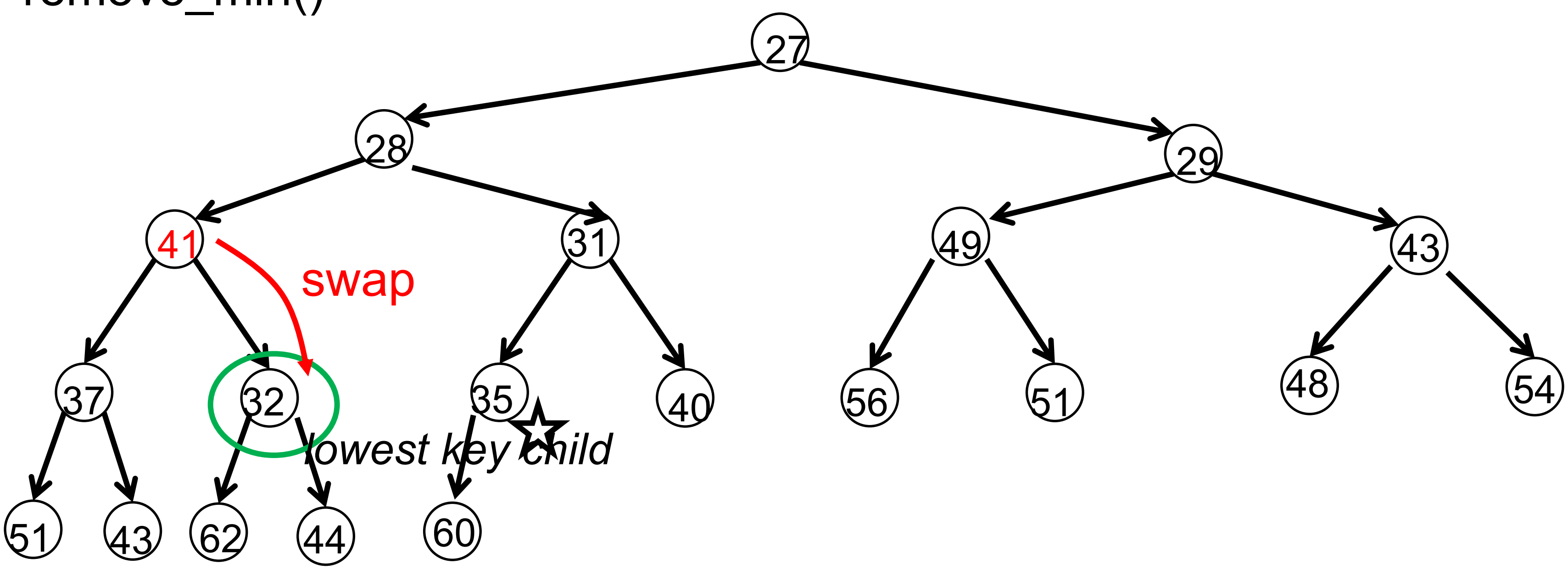
Removing top from a binary heap



Removing top from a binary heap

(25,*value*)

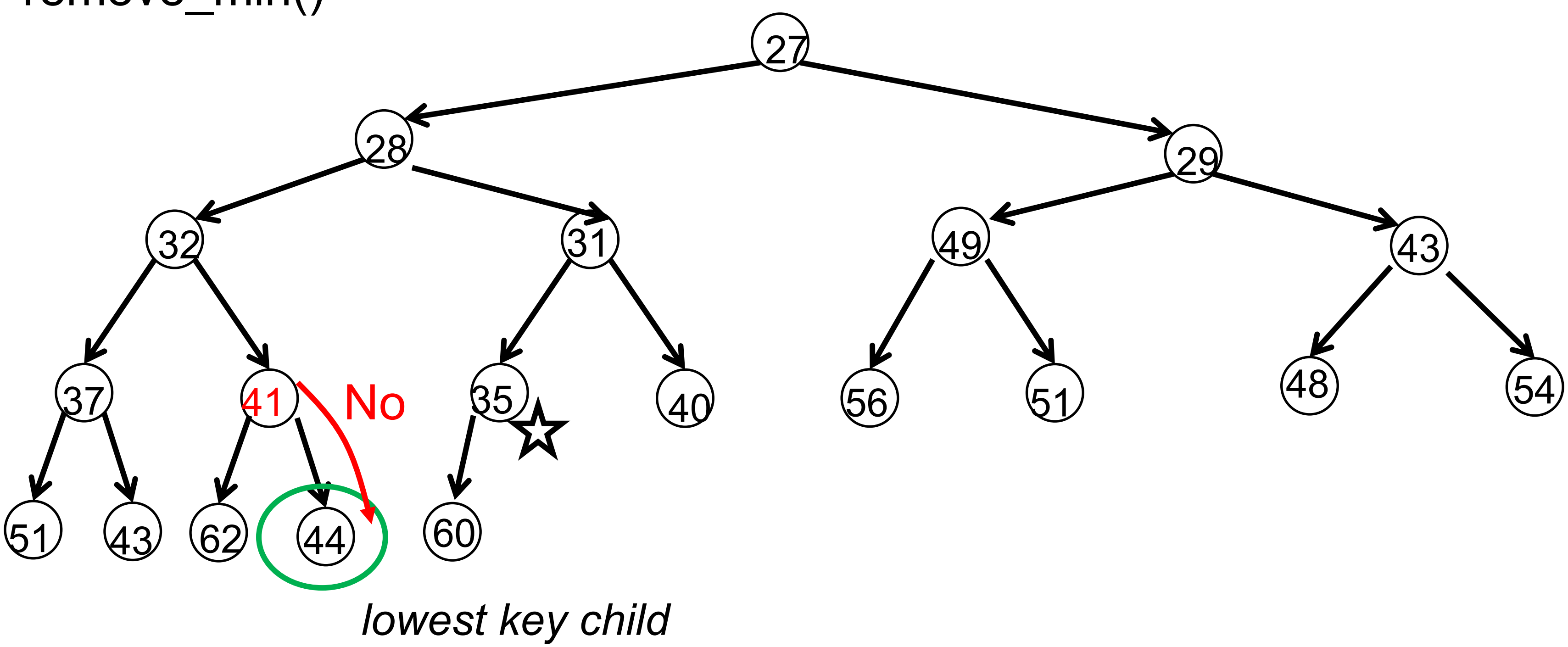
remove_min()



Removing top from a binary heap

(25,*value*)

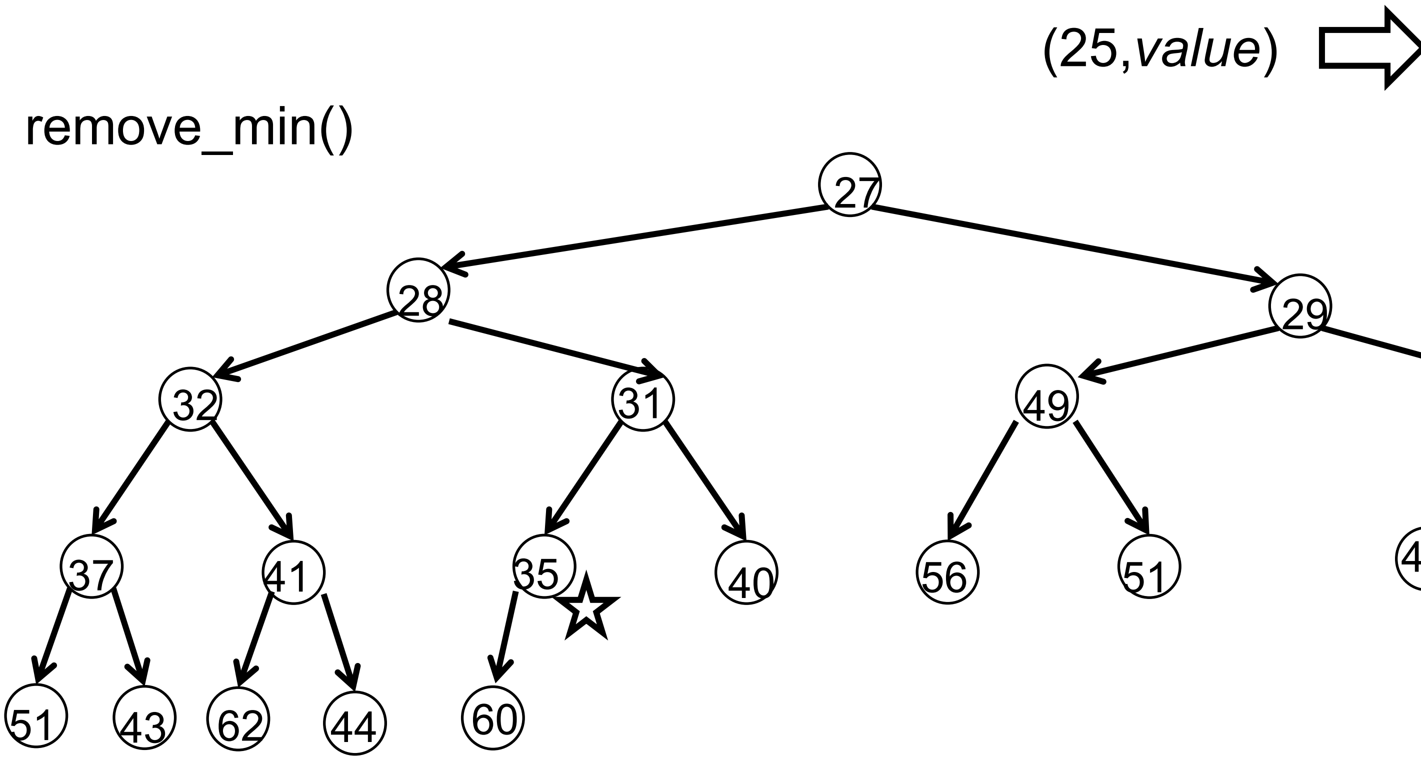
remove_min()



Removing top from a binary heap

(25,*value*)

remove_min()

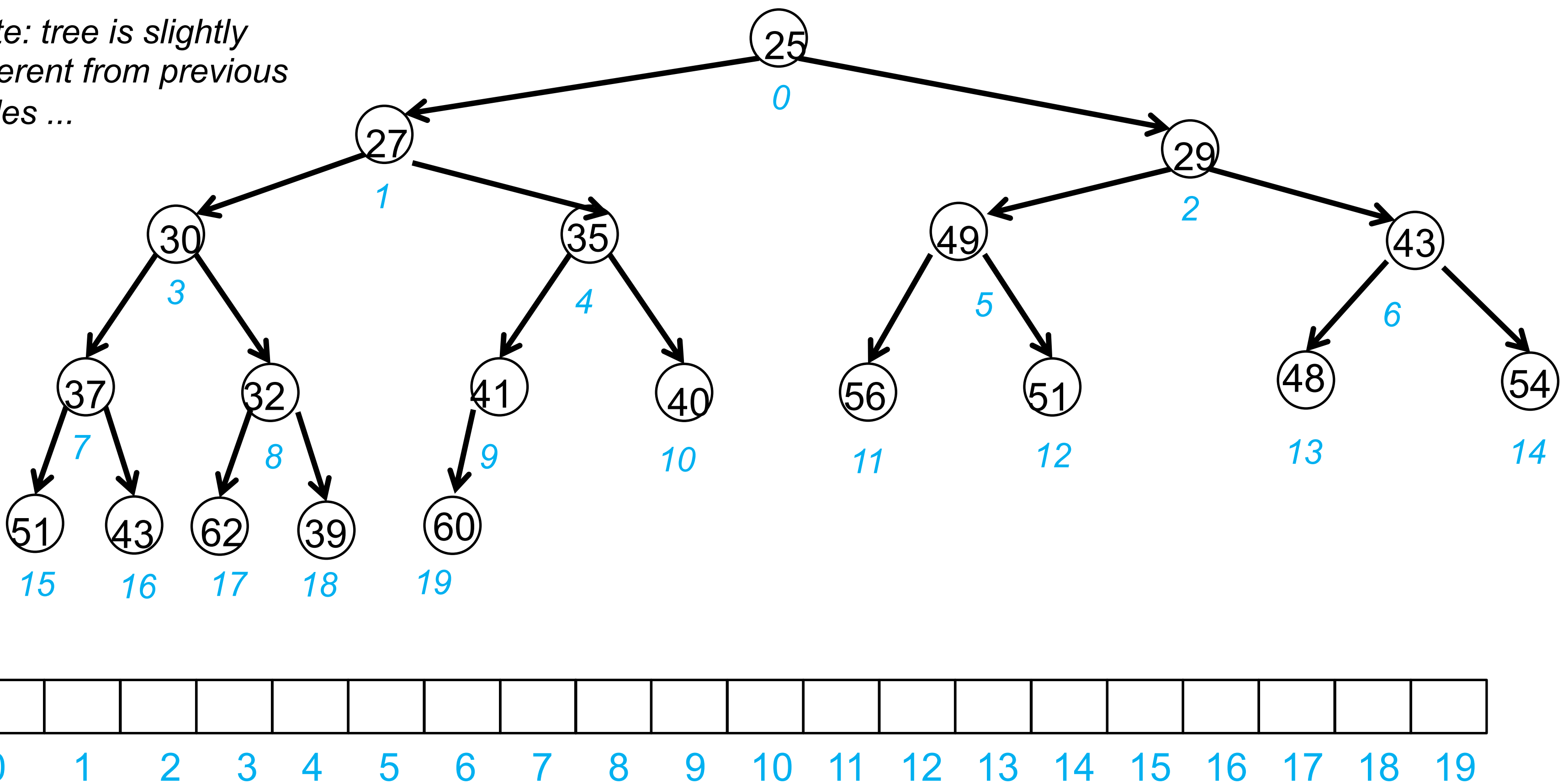


updating the last position might be expensive ...

Do we have to use a linked tree?

- *Prefer array-based representation over linked structure, but binary trees give us a O(log n) bound each individual operation.*

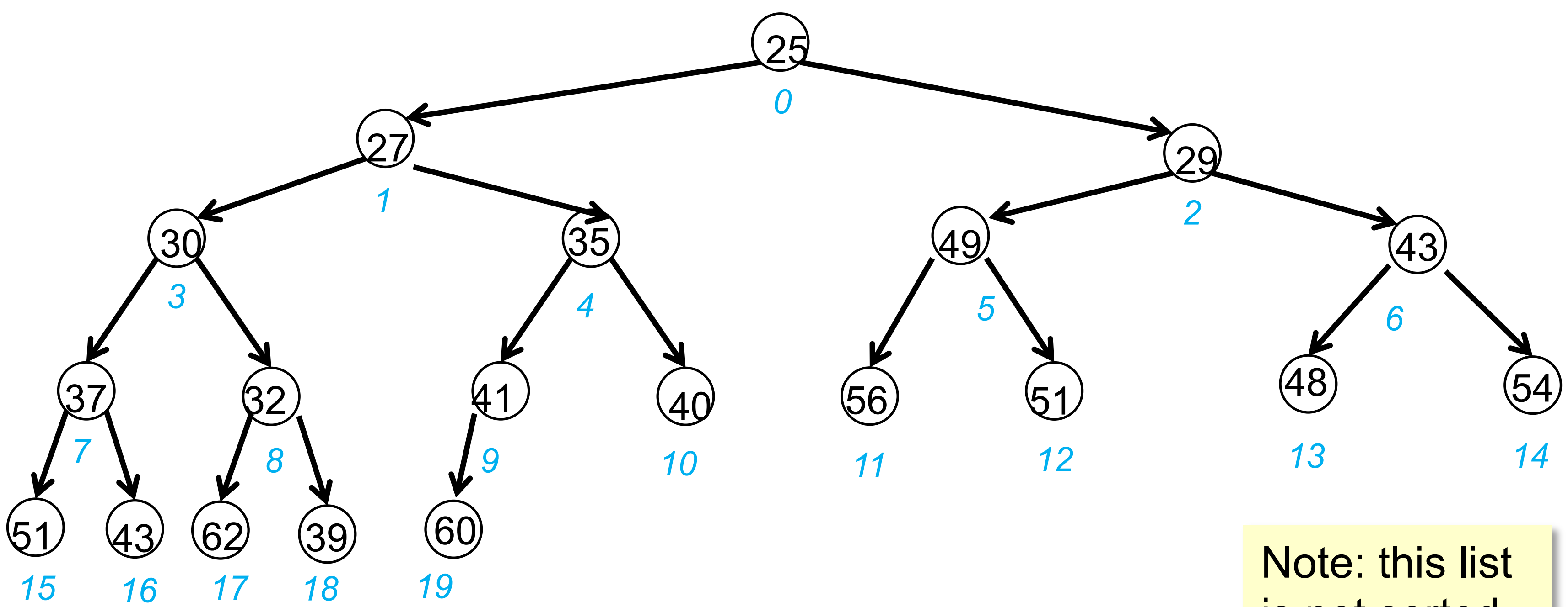
Note: tree is slightly different from previous slides ...



Represent the tree using an array-based list

Root node is at index 0.
Next item to be added is at index *size*.
Last item is at index *size-1*

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



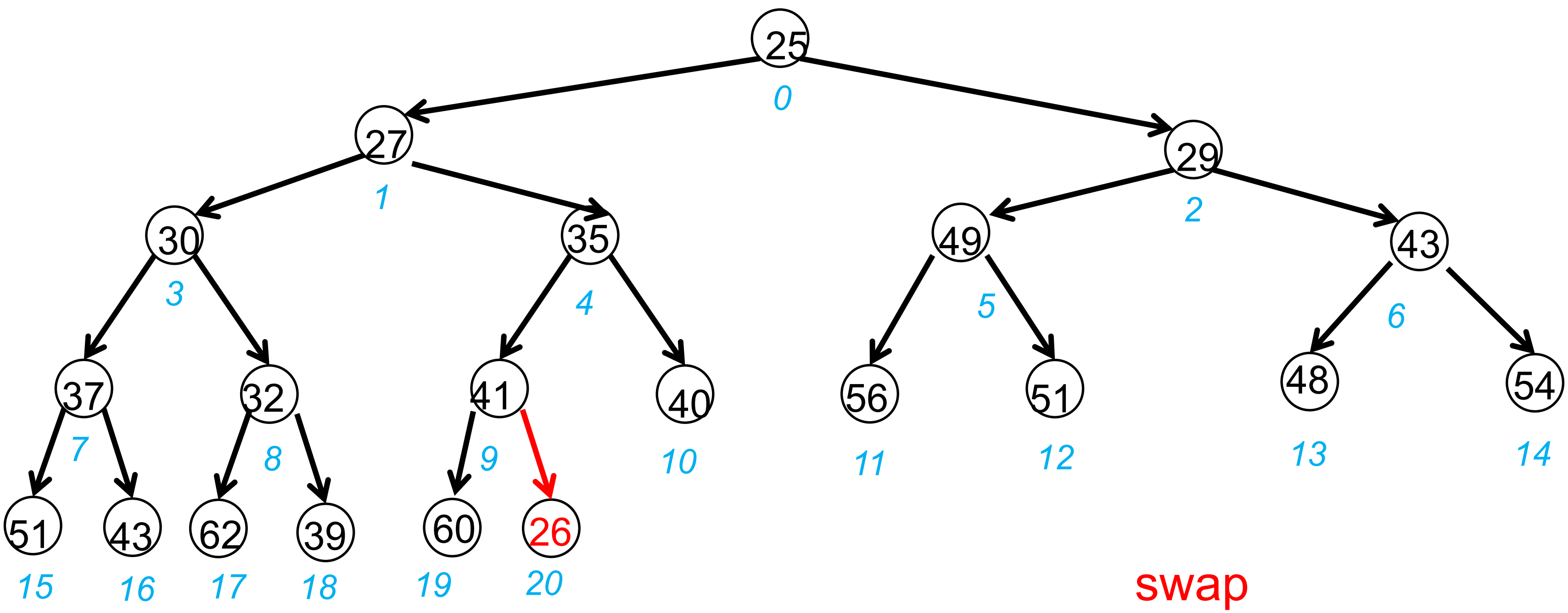
Note: this list is not sorted

25	27	29	30	35	49	43	37	32	41	40	56	51	48	54	51	43	62	39	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Represent the tree using an array-based list

Add 26

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



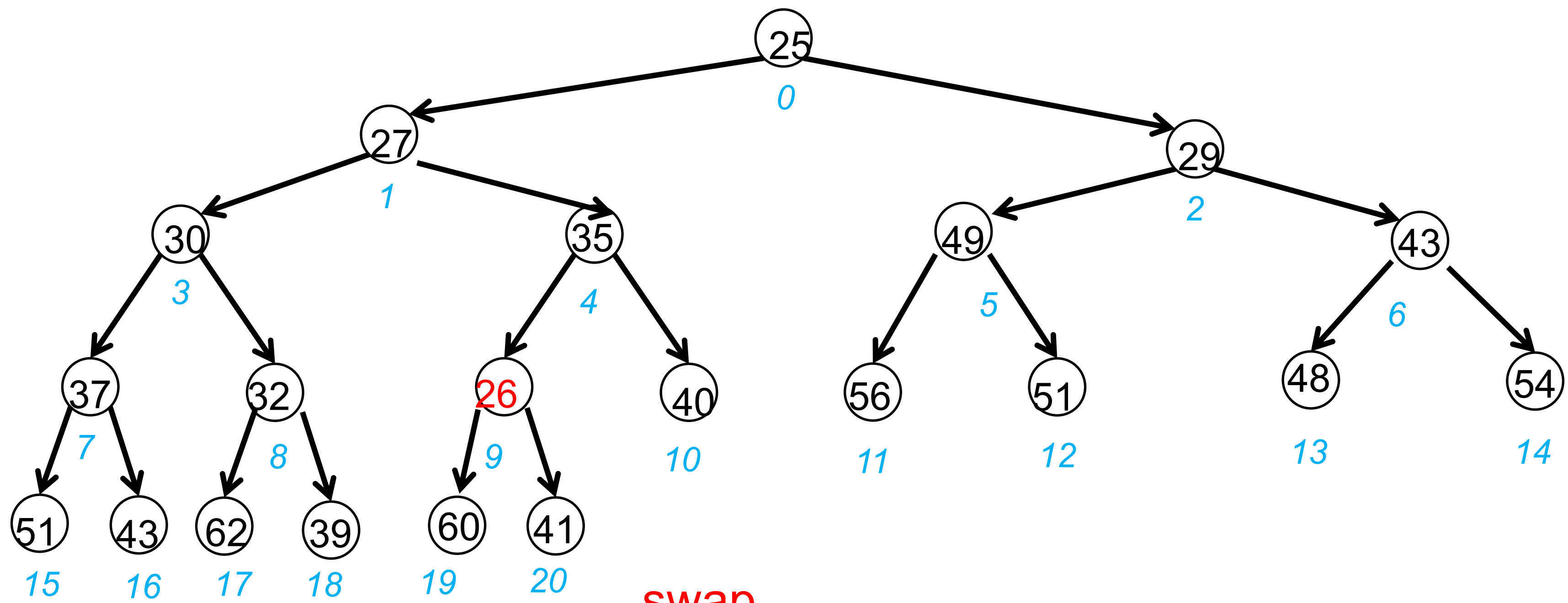
swap

25	27	29	30	35	49	43	37	32	41	40	56	51	48	54	51	43	62	39	60	26
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Represent the tree using an array-based list

Add 26

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



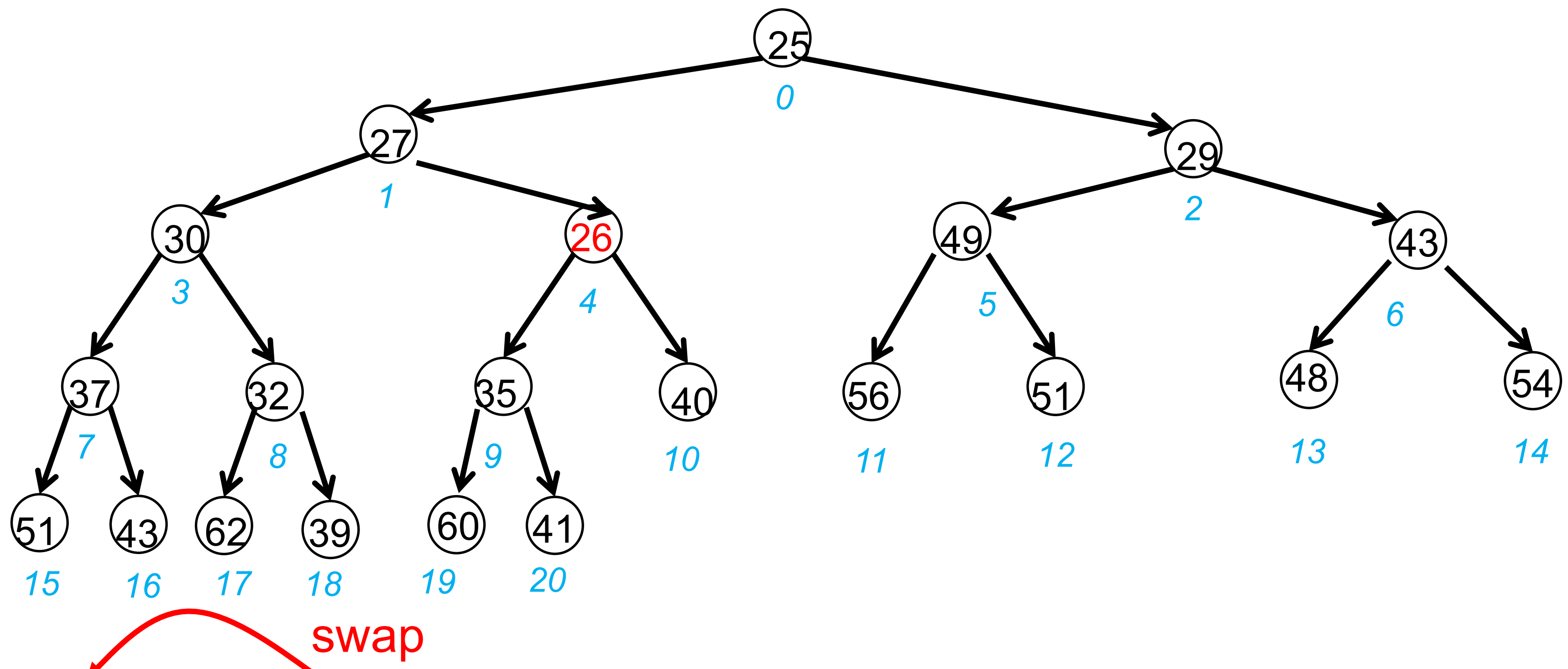
swap

25	27	29	30	35	49	43	37	32	26	40	56	51	48	54	51	43	62	39	60	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Represent the tree using an array-based list

Add 26

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2

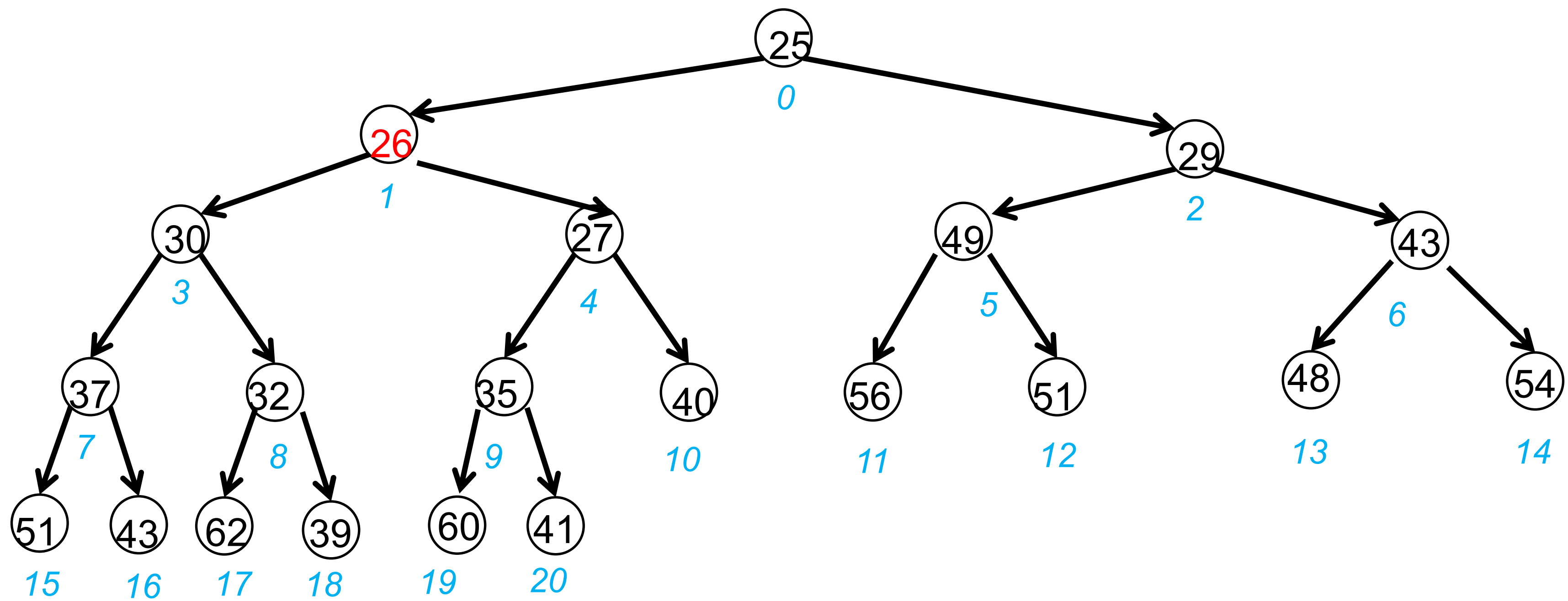


25	27	29	30	26	49	43	37	32	35	40	56	51	48	54	51	43	62	39	60	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Represent the tree using an array-based list

Add 26

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



No

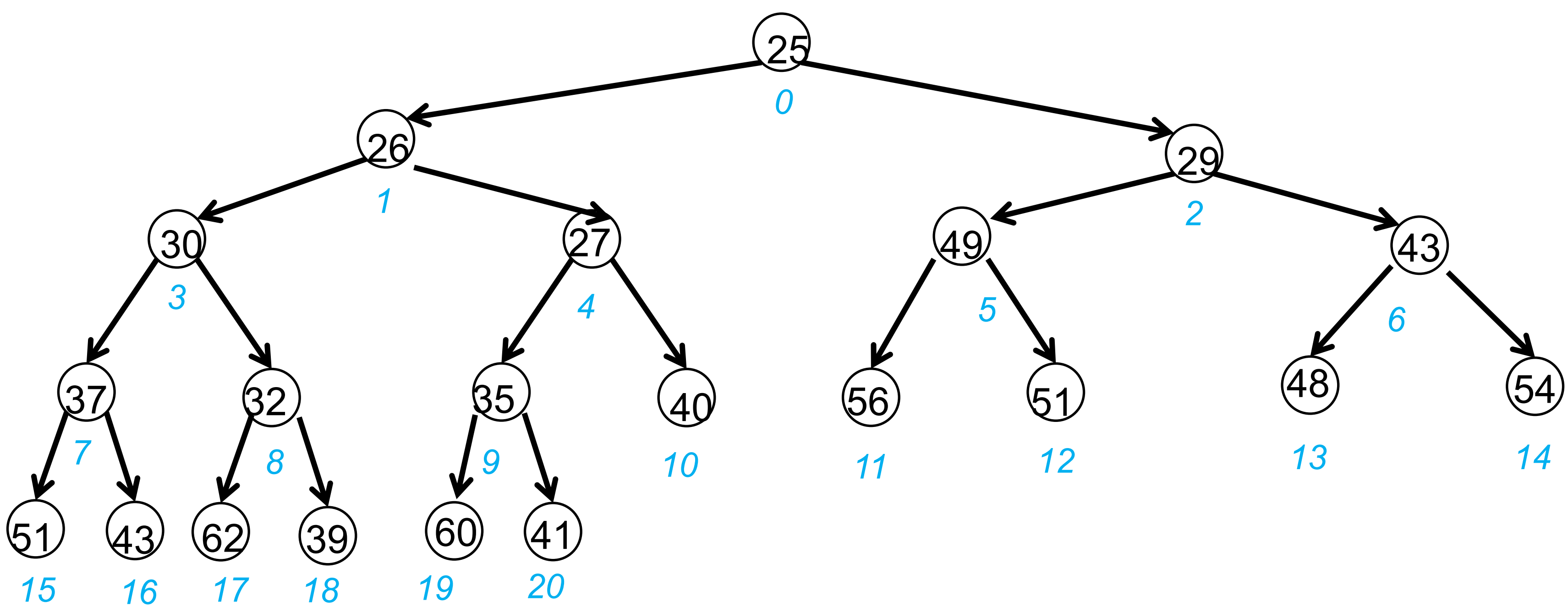
25	26	29	30	27	49	43	37	32	35	40	56	51	48	54	51	43	62	39	60	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Represent the tree using an array-based list

Remove min

25's value

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



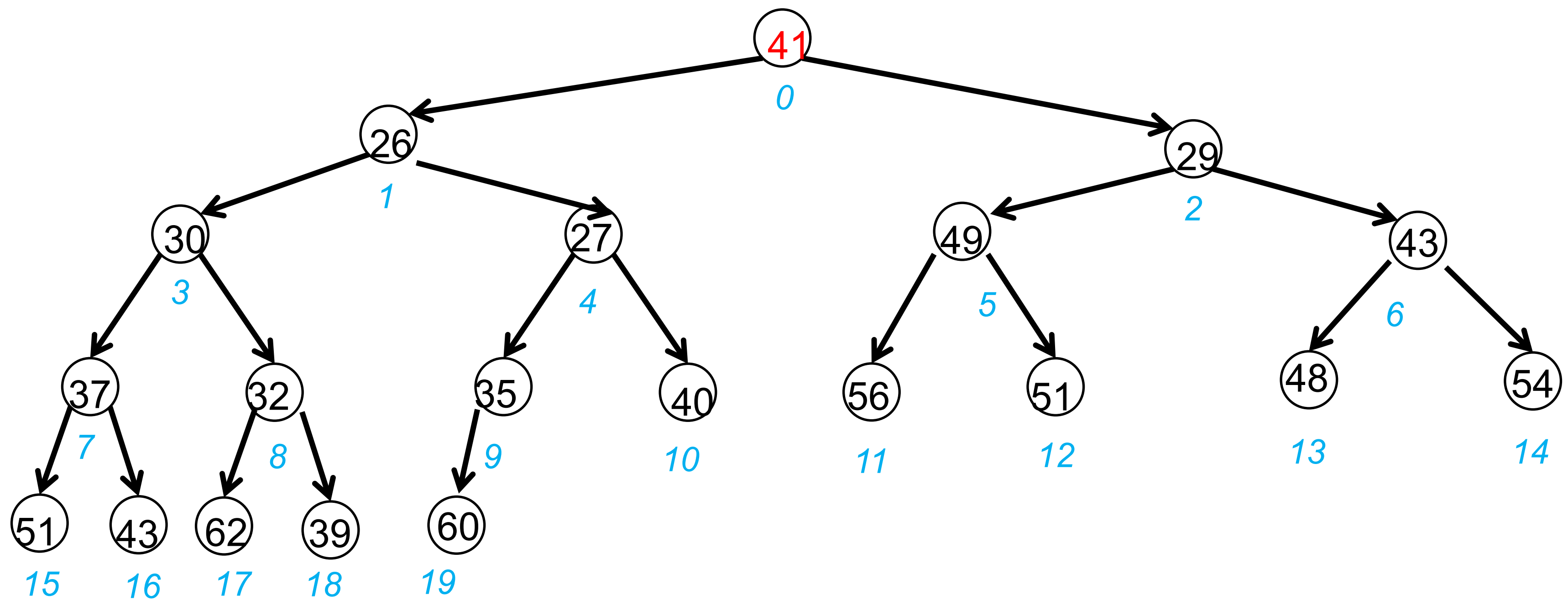
25	26	29	30	27	49	43	37	32	35	40	56	51	48	54	51	43	62	39	60	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Represent the tree using an array-based list

Remove min

25's value

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



swap

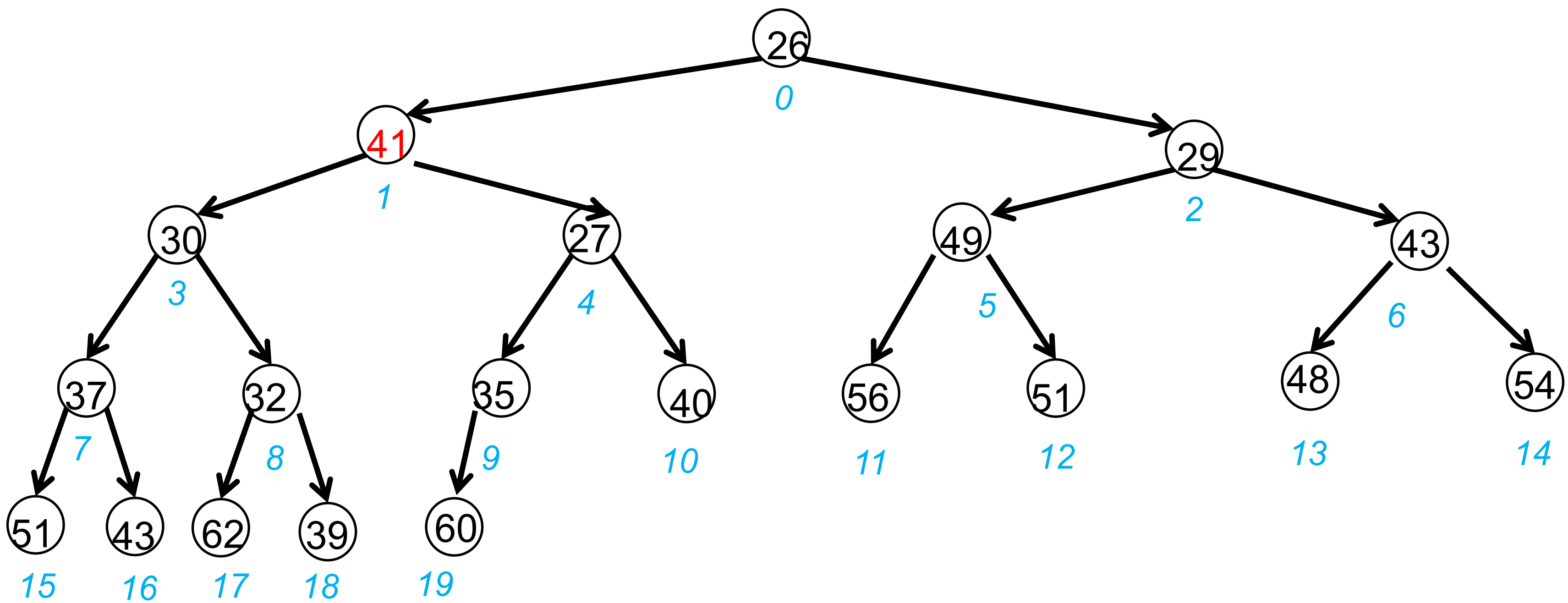
41	26	29	30	27	49	43	37	32	35	40	56	51	48	54	51	43	62	39	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Represent the tree using an array-based list

Remove min

25's value

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



swap

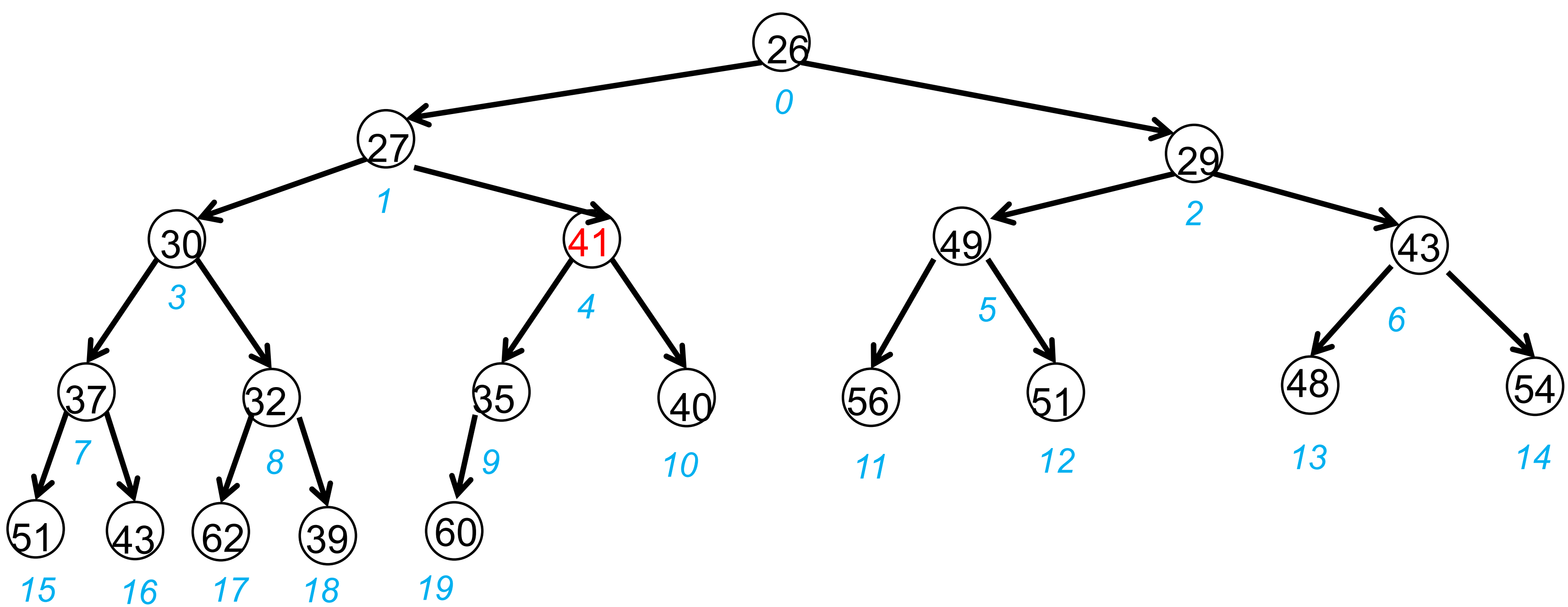
26	41	29	30	27	49	43	37	32	35	40	56	51	48	54	51	43	62	39	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Represent the tree using an array-based list

Remove min

25's value

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



swap

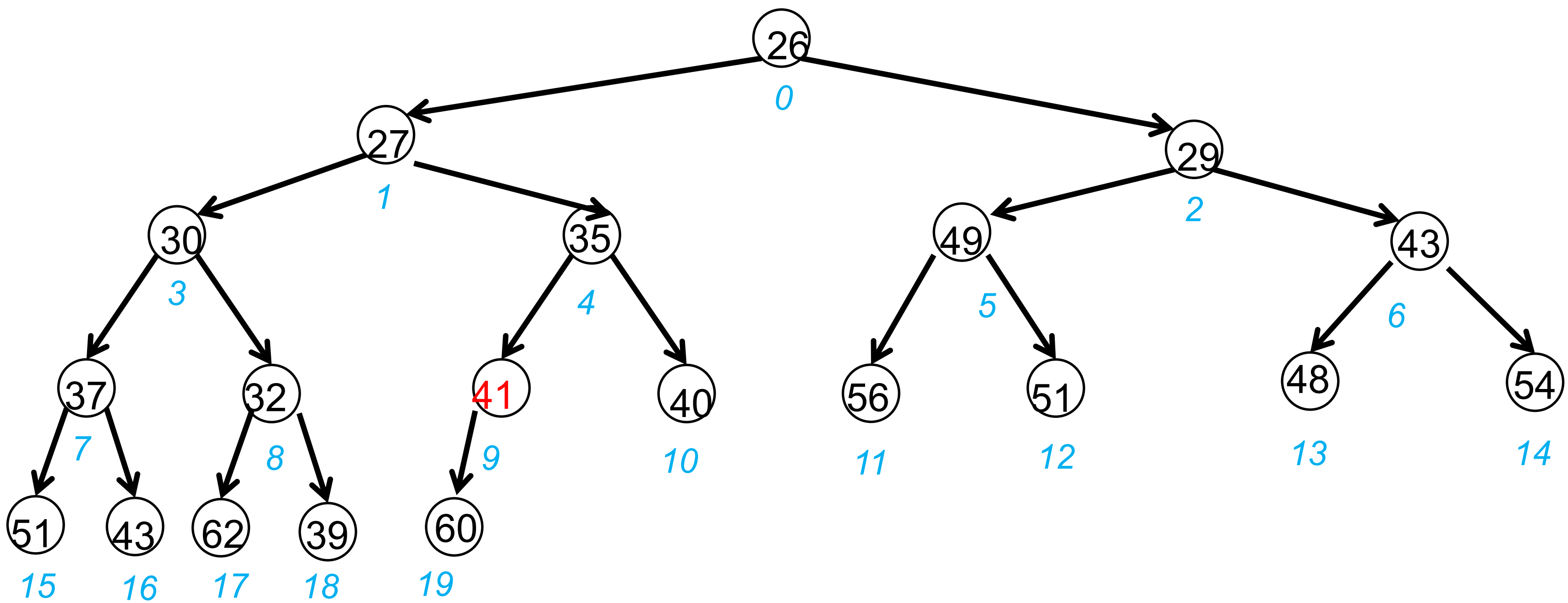
26	27	29	30	41	49	43	37	32	35	40	56	51	48	54	51	43	62	39	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Represent the tree using an array-based list

Remove min

25's value

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2



No

26	27	29	30	35	49	43	37	32	41	40	56	51	48	54	51	43	62	39	60
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Examples

left(i) = 2*i + 1
right(i) = 2*i + 2
parent(i) = (i-1)//2

Priority queue: implementation complexity

	add(k,v)	min()	remove_min()	length()	build full PQ
unsorted list	O(1)* append(E(k,v))	O(n)	O(n)	O(1)	O(n)
unsorted DLL	O(1) add at end	O(n)	O(n)	O(1)	O(n)
sorted list	O(n)	O(1)	O(1) min at end	O(1)	$O(n^2)$
sorted DLL	O(n)	O(1)	O(1)	O(1)	$O(n^2)$
AVL tree	O(log n)	O(log n)	O(log n)	O(1)	O(n log n)
Binary heap	O(log n)*	O(1)	O(log n)*	O(1)	O(n log n) or O(n)

Next Lecture

Dictionaries