

Sorting III

CS143: lecture 14

Konstantinos Ameranis, July 23

Sorting

So far...

- $O(n^2)$: Selection, insertion, bubble sorts
- $O(n \log n)$: Merge, quick sorts
- Final data structure / sorting algorithm: heap
- Note: the "heap" data structure has nothing to do with "heap" memory.

Priority Queue

Priority Queue

- All elements have a *priority*.

Priority Queue

- All elements have a *priority*.
- Insert elements into the data structure in any order; retrieve elements in the order of their priority (usually highest first).

Priority Queue

- All elements have a *priority*.
- Insert elements into the data structure in any order; retrieve elements in the order of their priority (usually highest first).
- Examples irl:

Priority Queue

- All elements have a *priority*.
- Insert elements into the data structure in any order; retrieve elements in the order of their priority (usually highest first).
- Examples irl:
 - Boarding airplanes

Priority Queue

- All elements have a *priority*.
- Insert elements into the data structure in any order; retrieve elements in the order of their priority (usually highest first).
- Examples include:
 - Boarding airplanes
 - Hospital triage

Priority Queue

- All elements have a *priority*.
- Insert elements into the data structure in any order; retrieve elements in the order of their priority (usually highest first).
- Examples include:
 - Boarding airplanes
 - Hospital triage
 - Covid vaccine eligibility

Priority Queue

Priority Queue

- Core operations:

Priority Queue

- Core operations:
 - `insert(v) ;` -- insert an element (assume that priority is in `v`)

Priority Queue

- Core operations:
 - `insert(v) ;` -- insert an element (assume that priority is in `v`)
 - `get_top() ;` -- return the element with top priority

Priority Queue

- Core operations:
 - `insert(v) ;` -- insert an element (assume that priority is in `v`)
 - `get_top() ;` -- return the element with top priority
 - `remove_top() ;` -- remove the element with top priority

Priority Queue

- Core operations:
 - `insert(v)` ; -- insert an element (assume that priority is in `v`)
 - `get_top()` ; -- return the element with top priority
 - `remove_top()` ; -- remove the element with top priority
- Others: `size()` `is_empty()` `update_priority()` ... ignored for now

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
<code>ArrayList</code>			

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
<code>ArrayList</code>	$O(1)$		

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
<code>ArrayList</code>	$O(1)$	$O(n)$	

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
<code>ArrayList</code>	$O(1)$	$O(n)$	$O(n)$

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList			

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$		

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List			

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$		

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST			

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$		

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST			

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$		

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$

Heap

- A heap is an efficient implementation of priority queues.
- A heap is a binary tree satisfying the following two constraints:
 - Shape property
 - Value property

Heap

Shape property

Heap

Shape property

- A heap is a complete binary tree.

Heap

Shape property

- A heap is a complete binary tree.
 - A binary tree is complete if and only if:

Heap

Shape property

- A heap is a complete binary tree.
 - A binary tree is complete if and only if:
 - Each level is full, except possibly the last level;

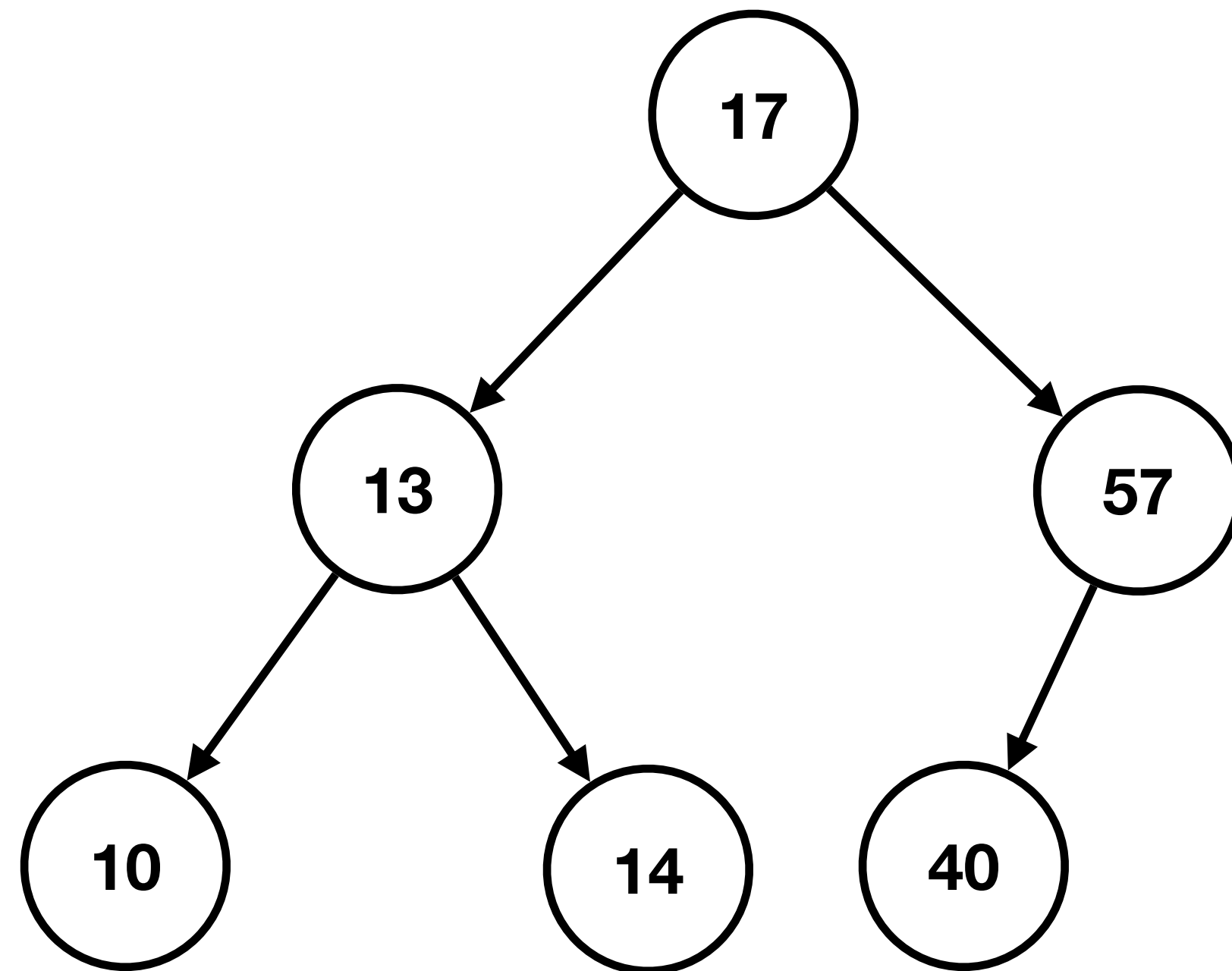
Heap

Shape property

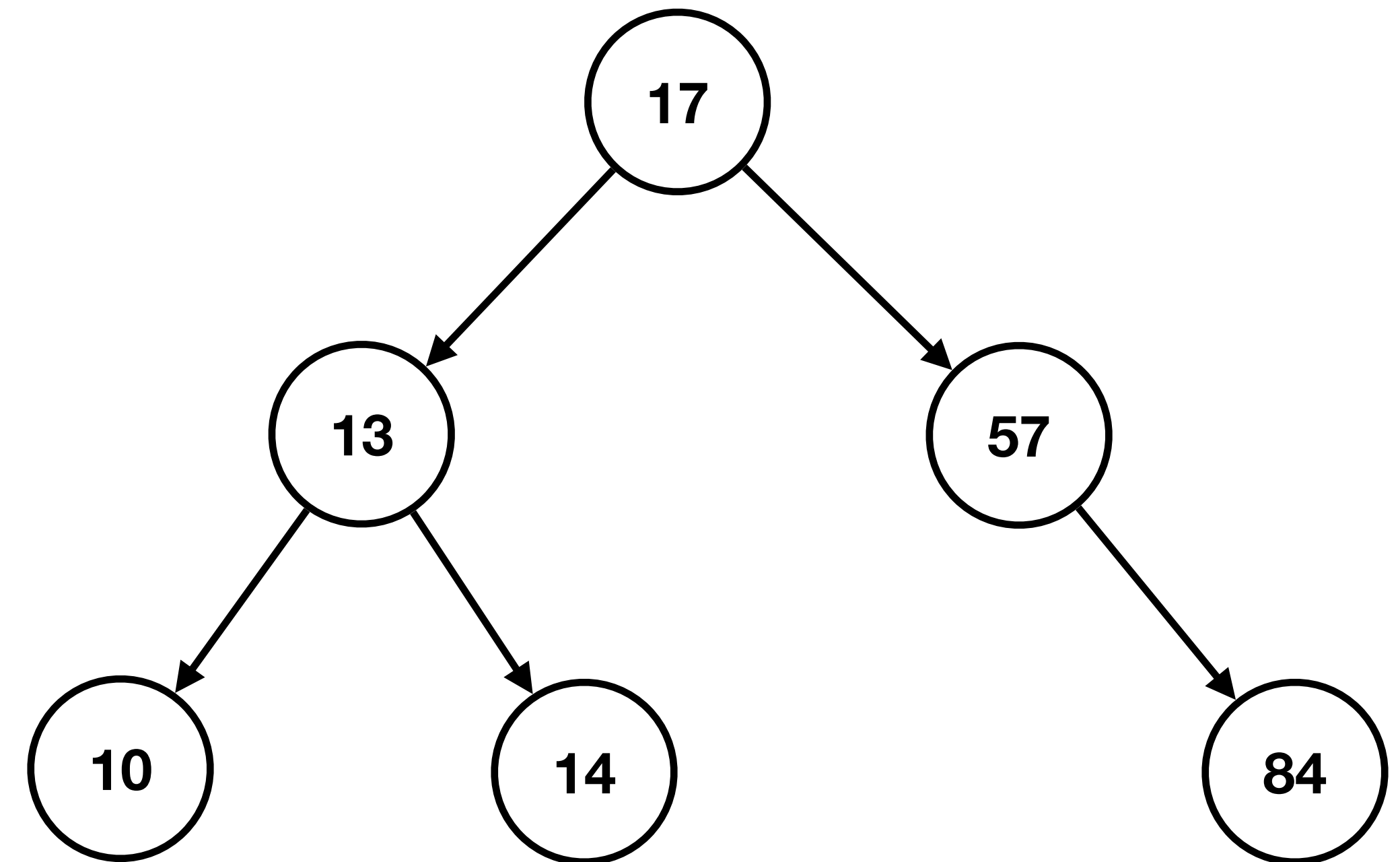
- A heap is a complete binary tree.
 - A binary tree is complete if and only if:
 - Each level is full, except possibly the last level;
 - The last level is filled from left to right

Heap

Shape property



Complete



Incomplete

Heap

Value Property

Heap

Value Property

- In a heap:

Heap

Value Property

- In a heap:
 - Parents have *higher priority* than their children: (two flavors)

Heap

Value Property

- In a heap:
 - Parents have *higher priority* than their children: (two flavors)
 - *Min-heap*: parent value is less than any child value.

Heap

Value Property

- In a heap:
 - Parents have *higher priority* than their children: (two flavors)
 - *Min-heap*: parent value is less than any child value.
 - *Max-heap*: parent value is greater than any child value. (We use this flavor in this class)

Heap

Value Property

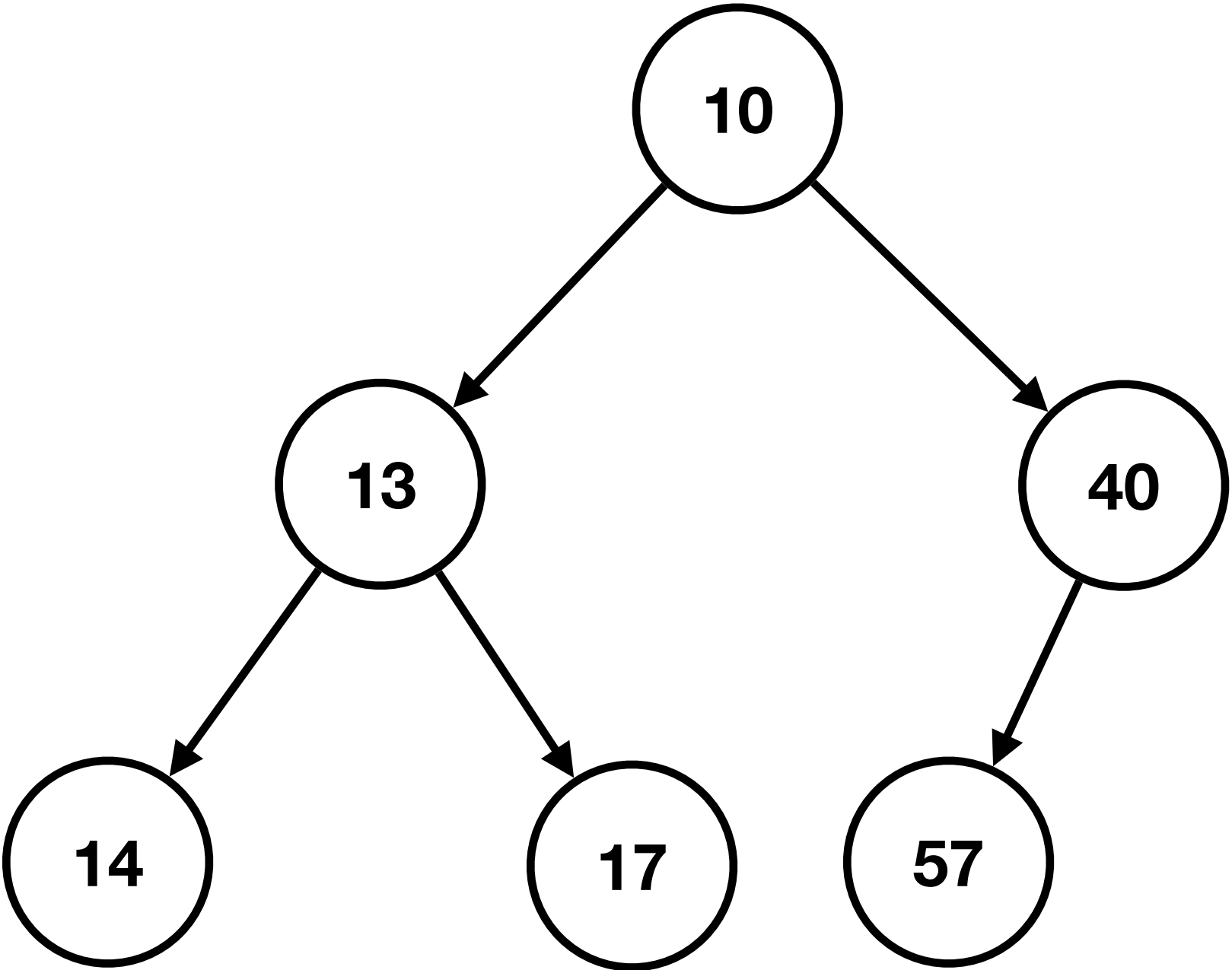
- In a heap:
 - Parents have *higher priority* than their children: (two flavors)
 - *Min-heap*: parent value is less than any child value.
 - *Max-heap*: parent value is greater than any child value. (We use this flavor in this class)
 - Order between children does not matter.

Heap

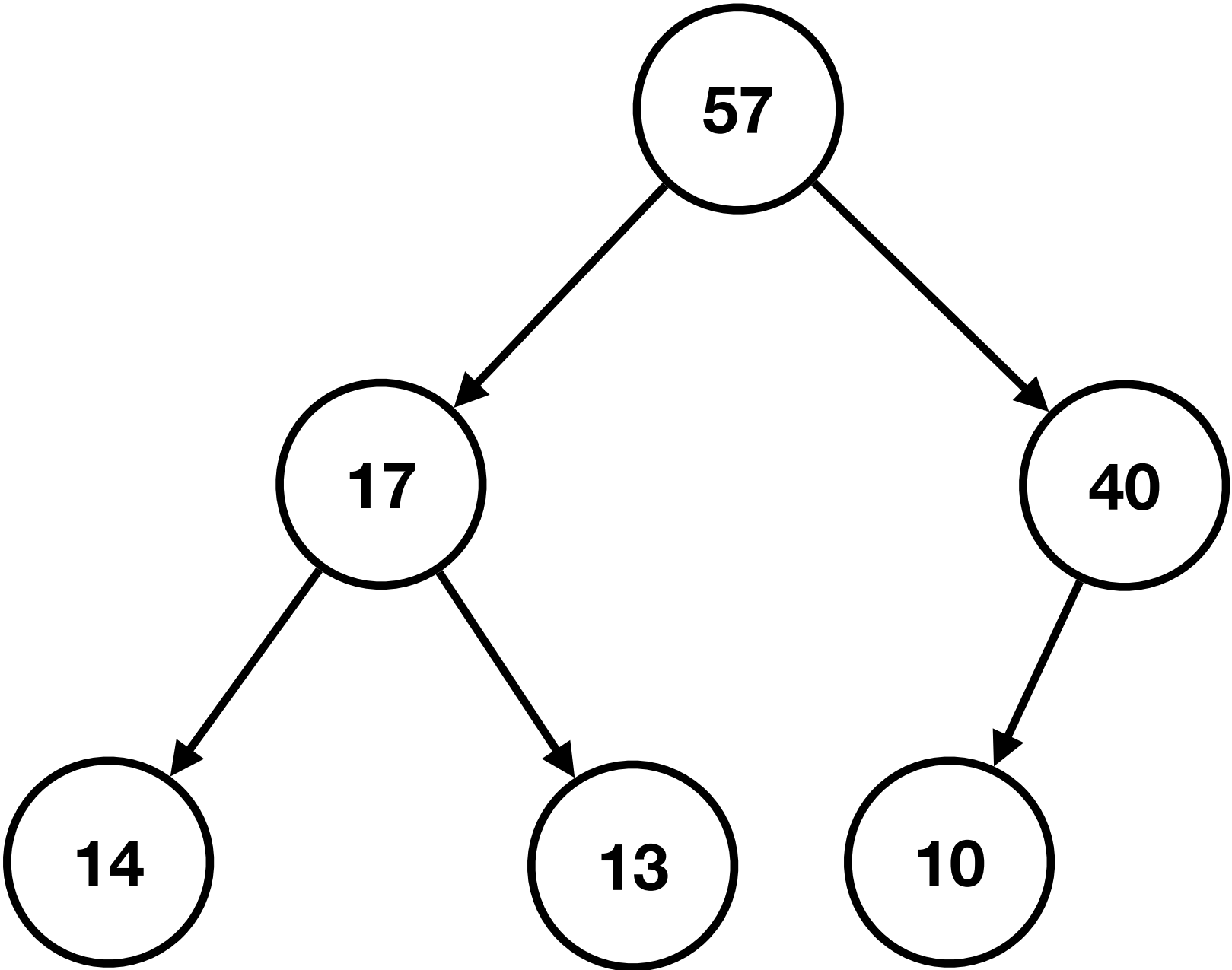
Value Property

- In a heap:
 - Parents have *higher priority* than their children: (two flavors)
 - *Min-heap*: parent value is less than any child value.
 - *Max-heap*: parent value is greater than any child value. (We use this flavor in this class)
 - Order between children does not matter.
- *This is not a BST.*

Heap

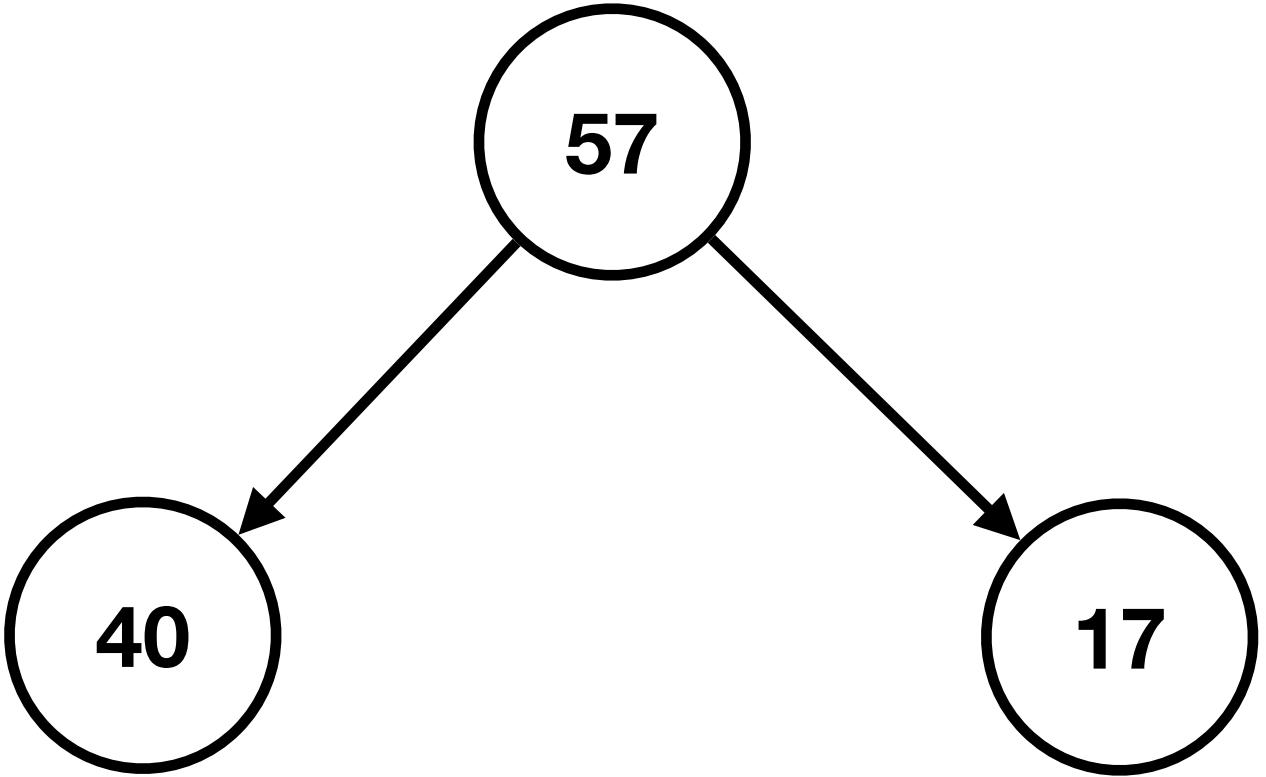
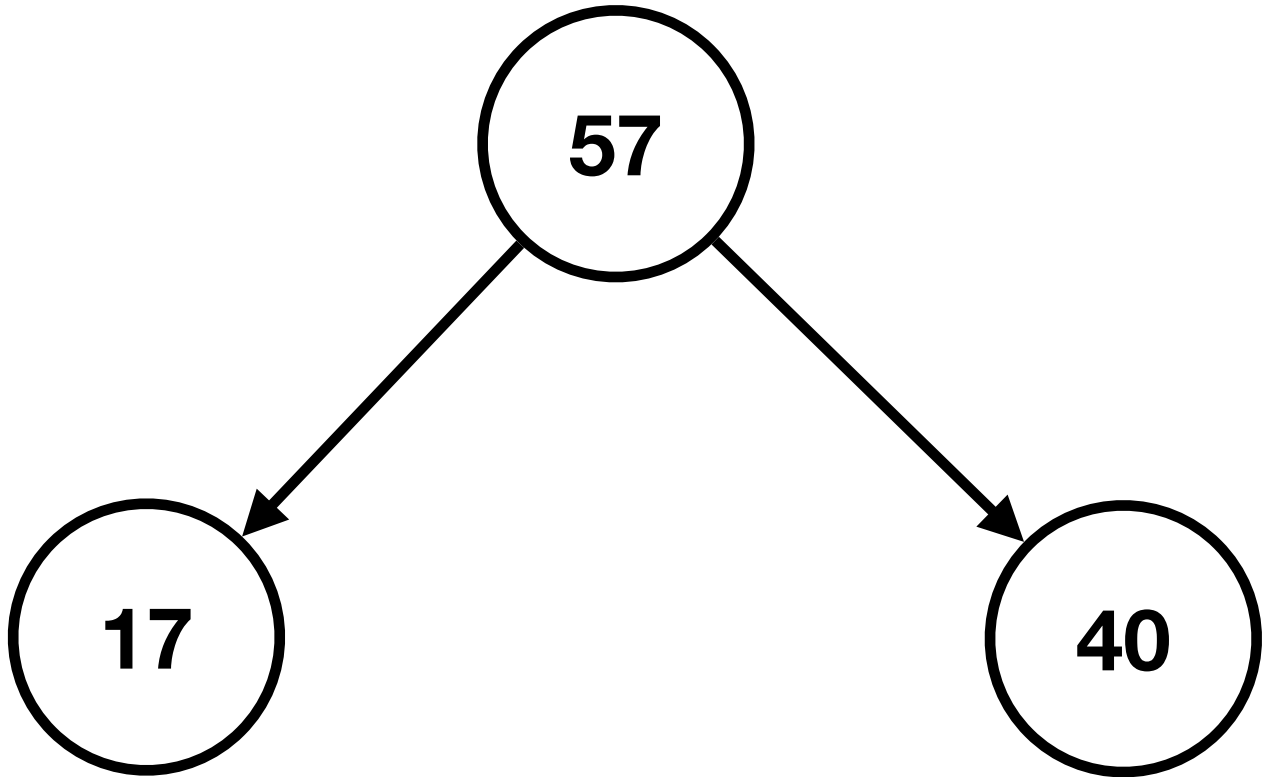


Min-heap

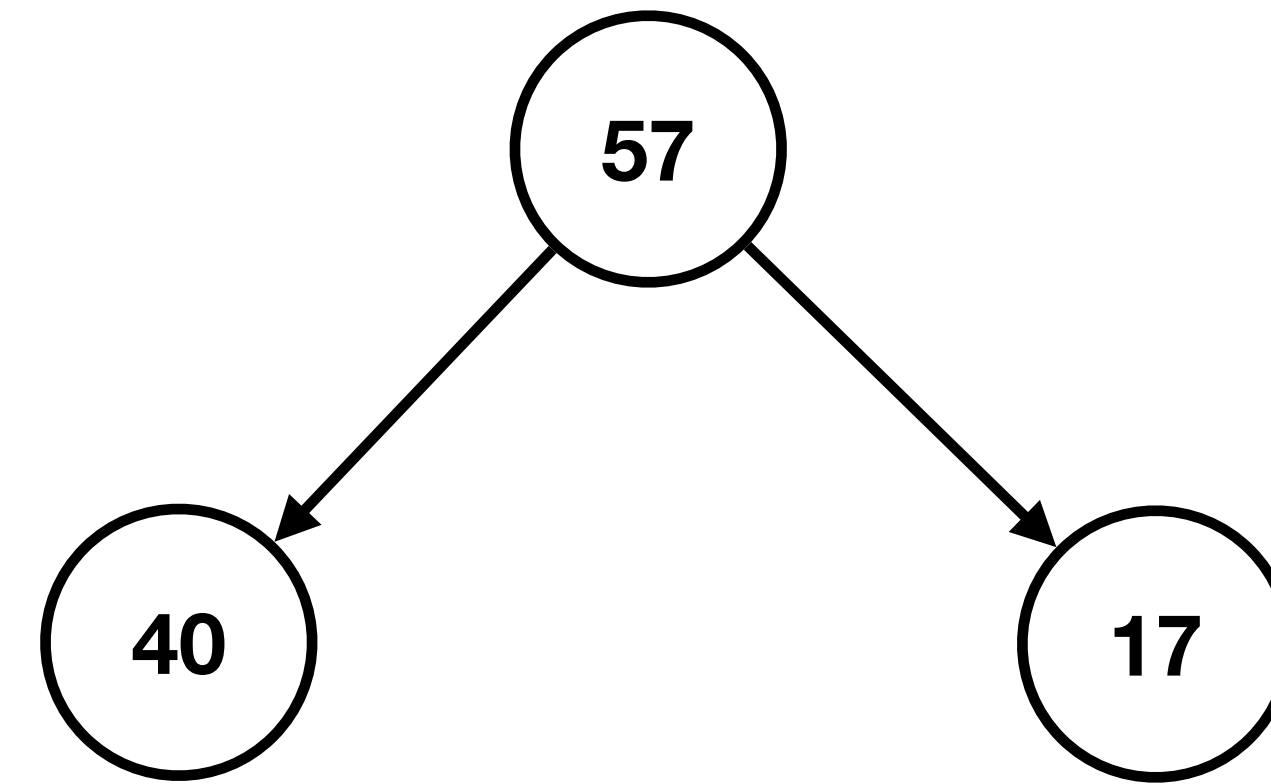
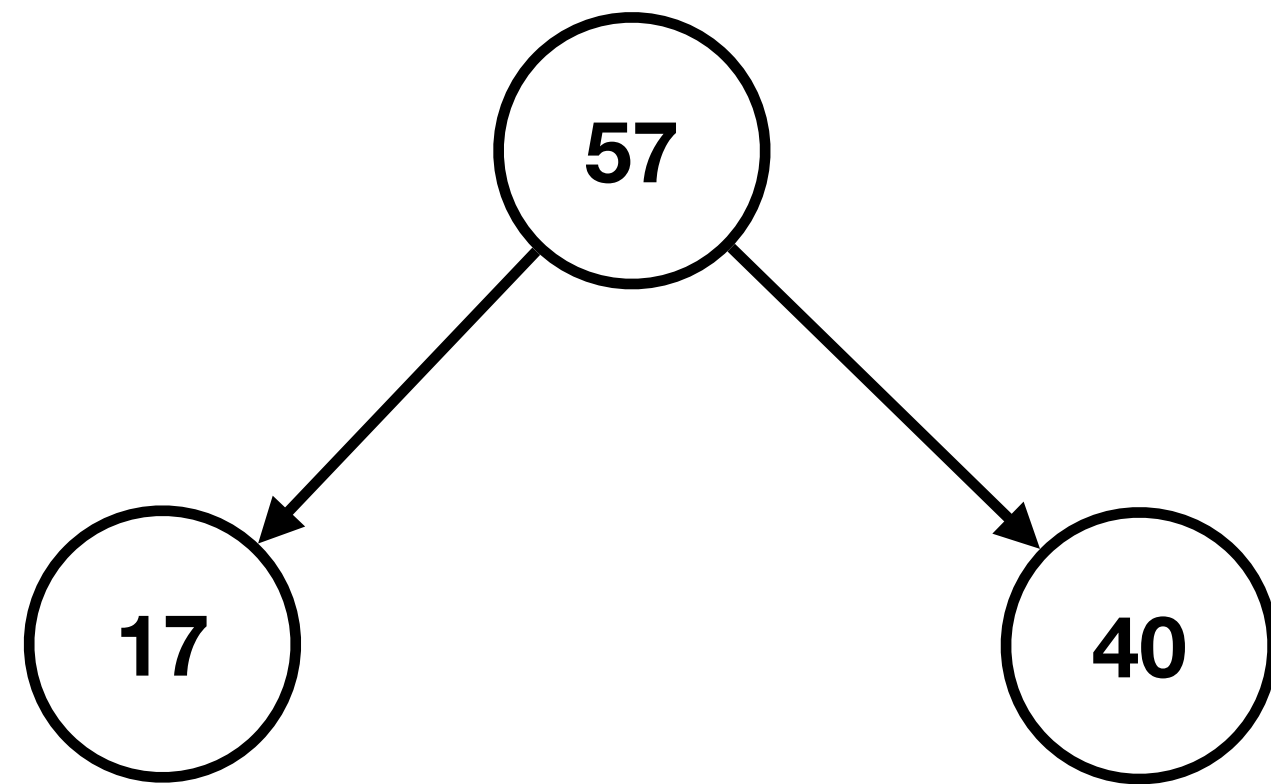


Max-heap

Heap

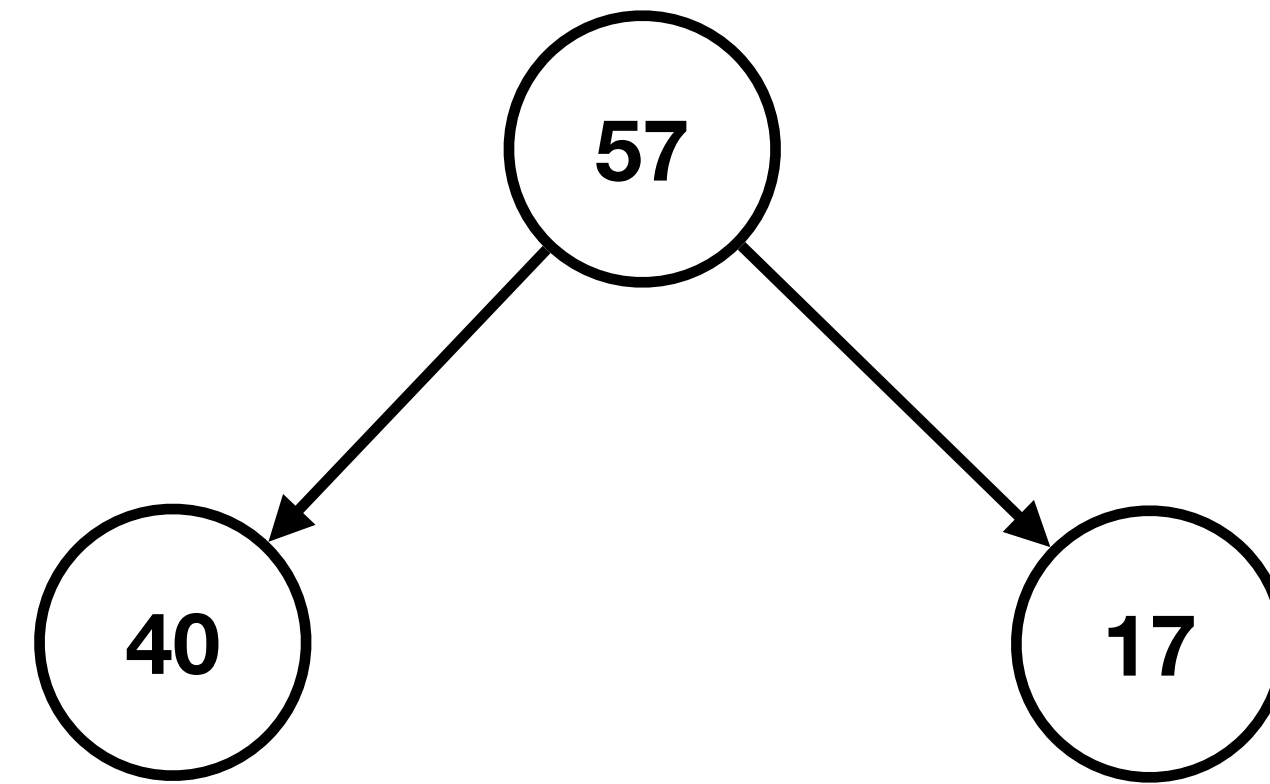
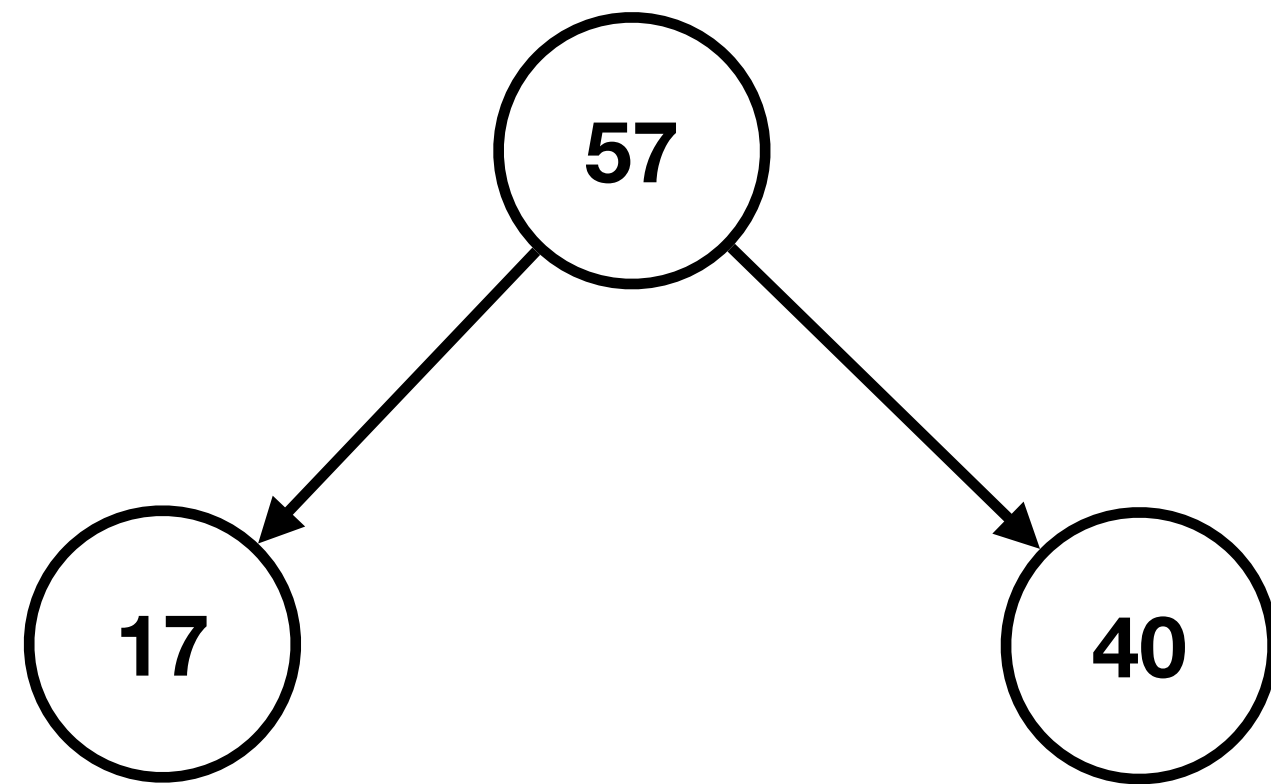


Heap



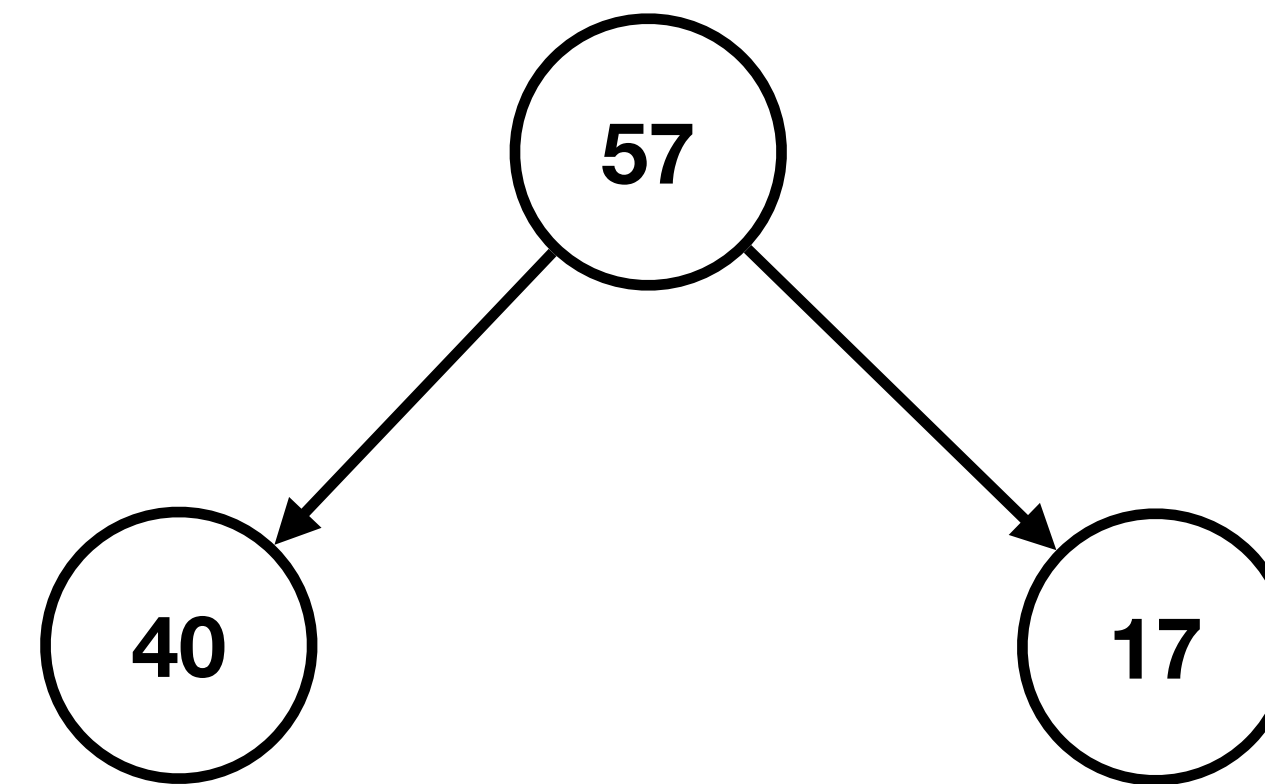
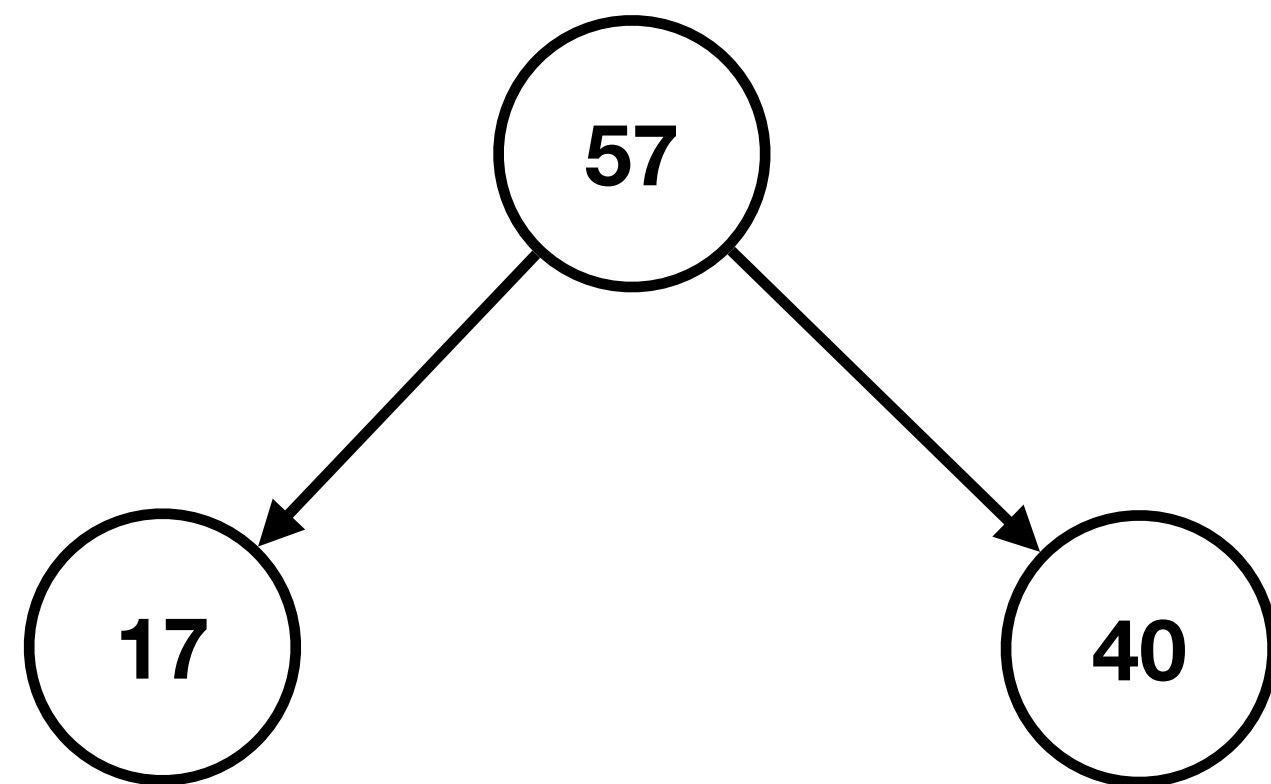
- *Not a BST because:*

Heap



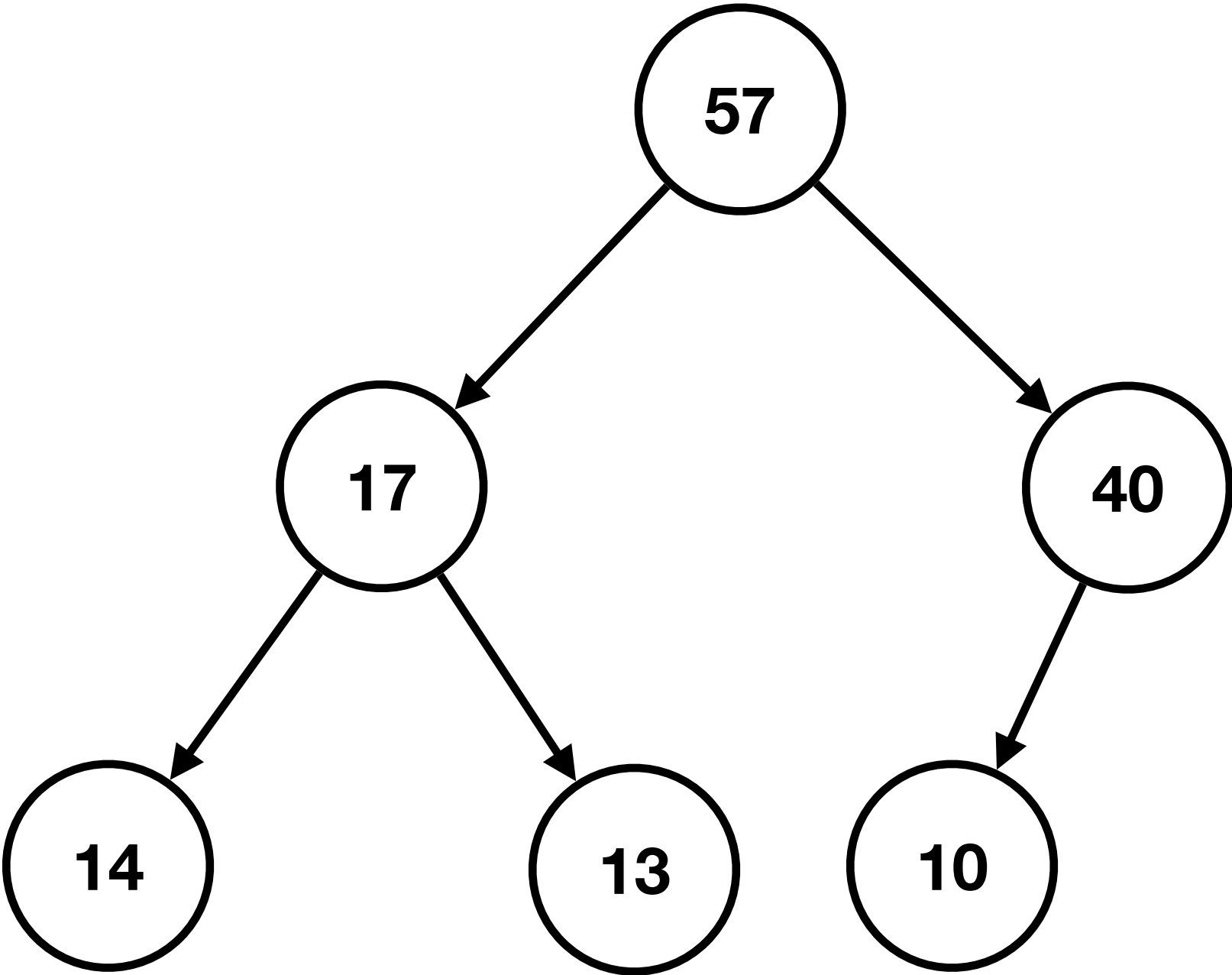
- *Not a BST because:*
 - Parents is greater than both children.

Heap

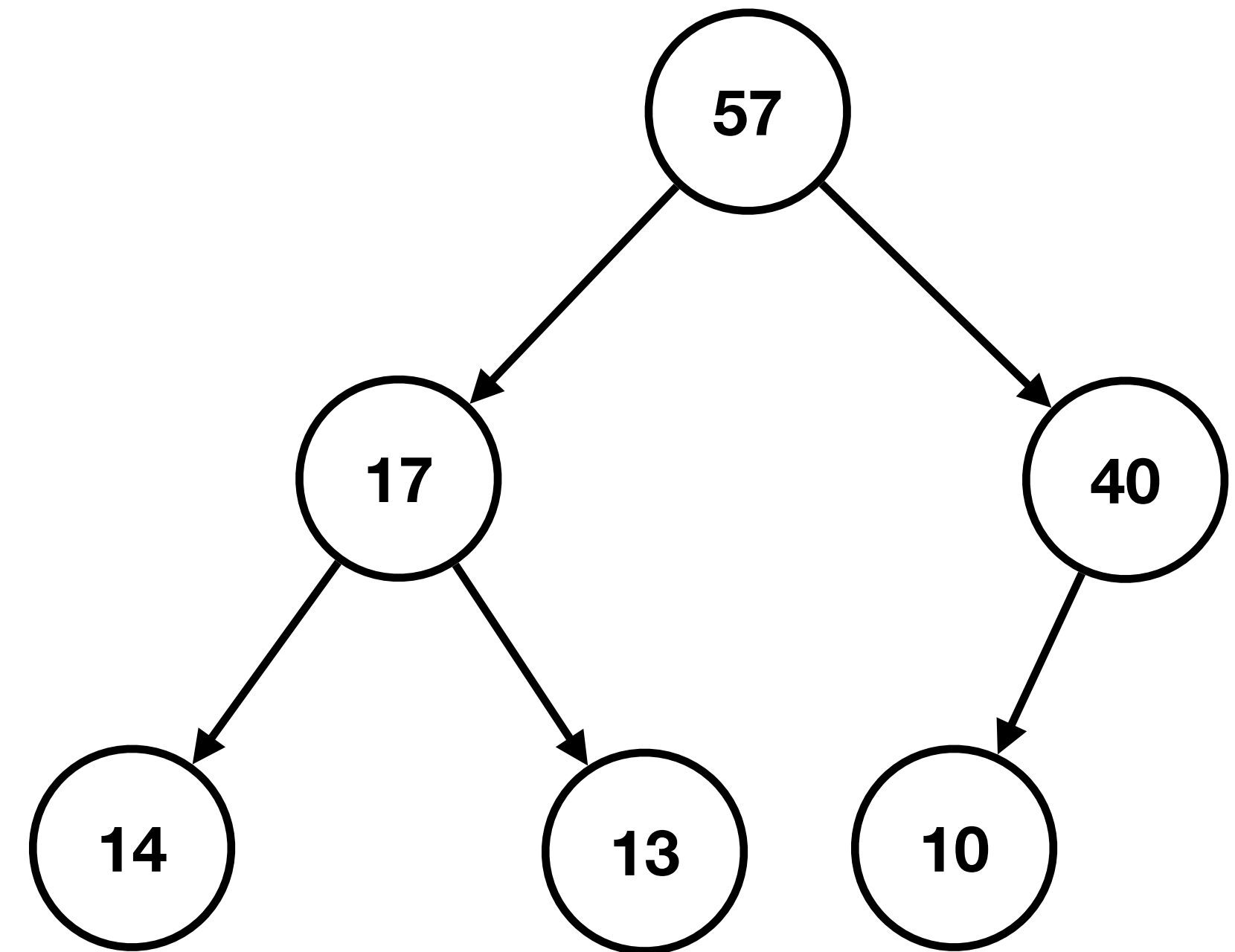


- *Not a BST because:*
 - Parents is greater than both children.
 - No guaranteed ordering between children.

Heap

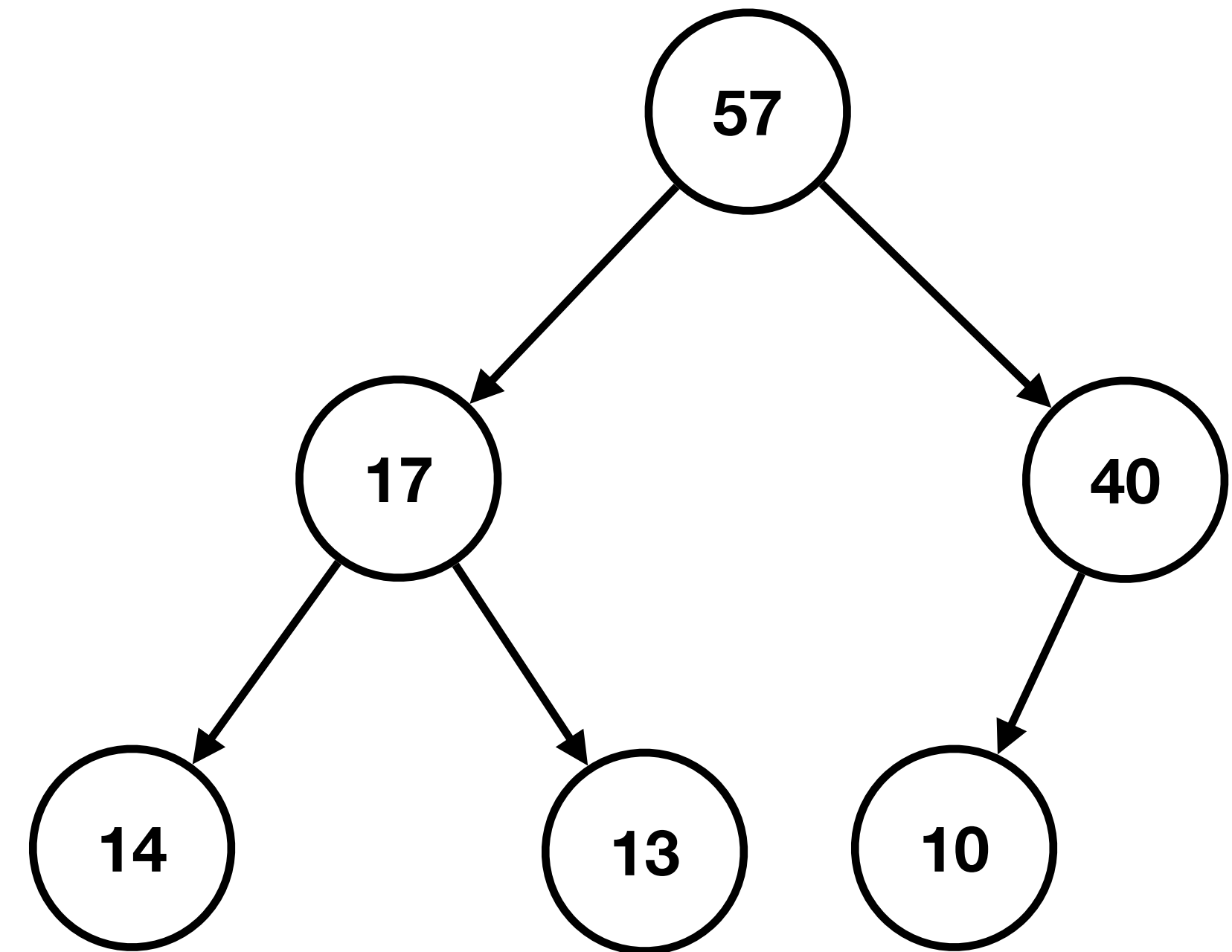


Heap



- Being complete means every 6-element heap has this shape.

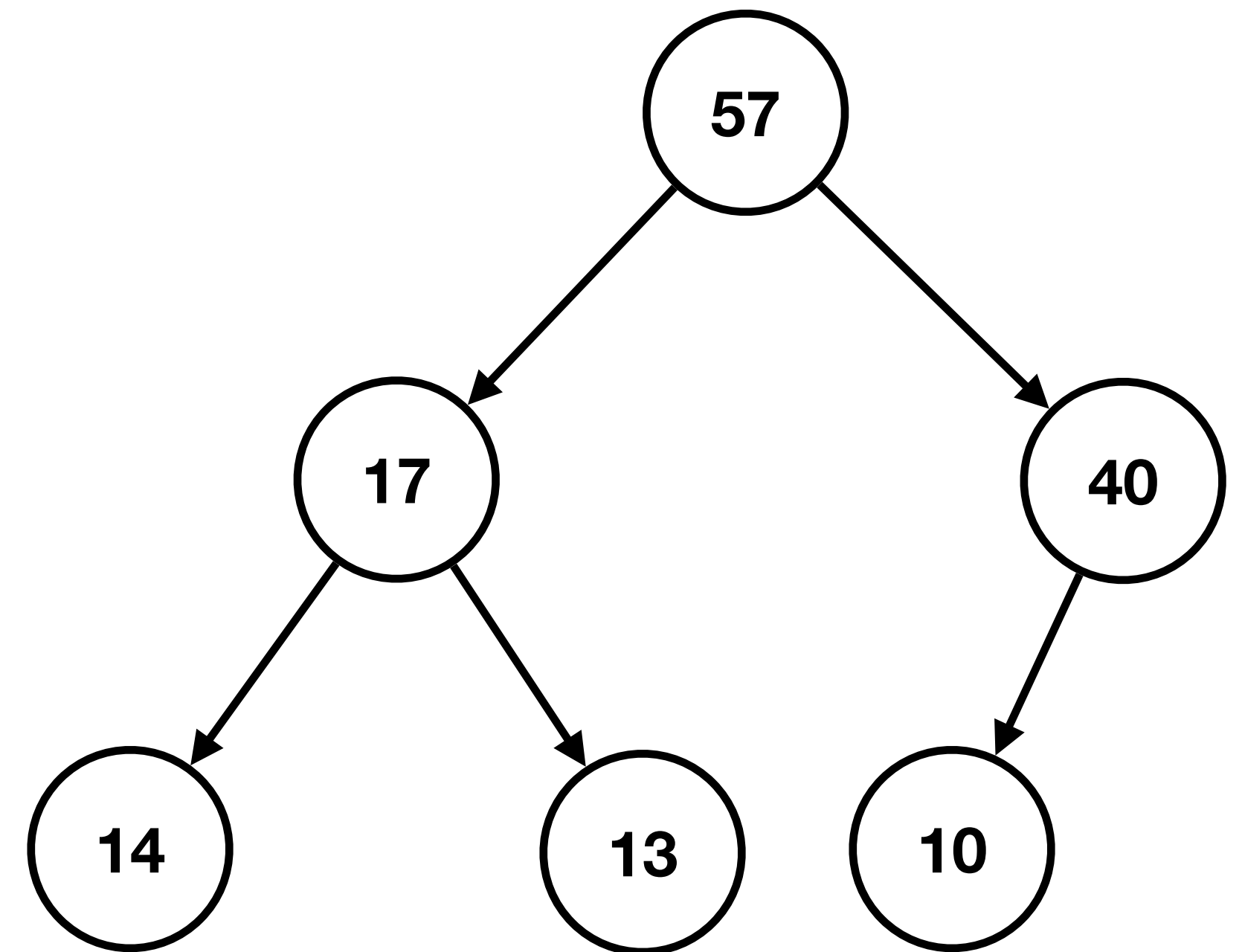
Heap



- Being complete means every 6-element heap has this shape.
- If a 7th element is added, the new trees will always have the same shape.

Heap

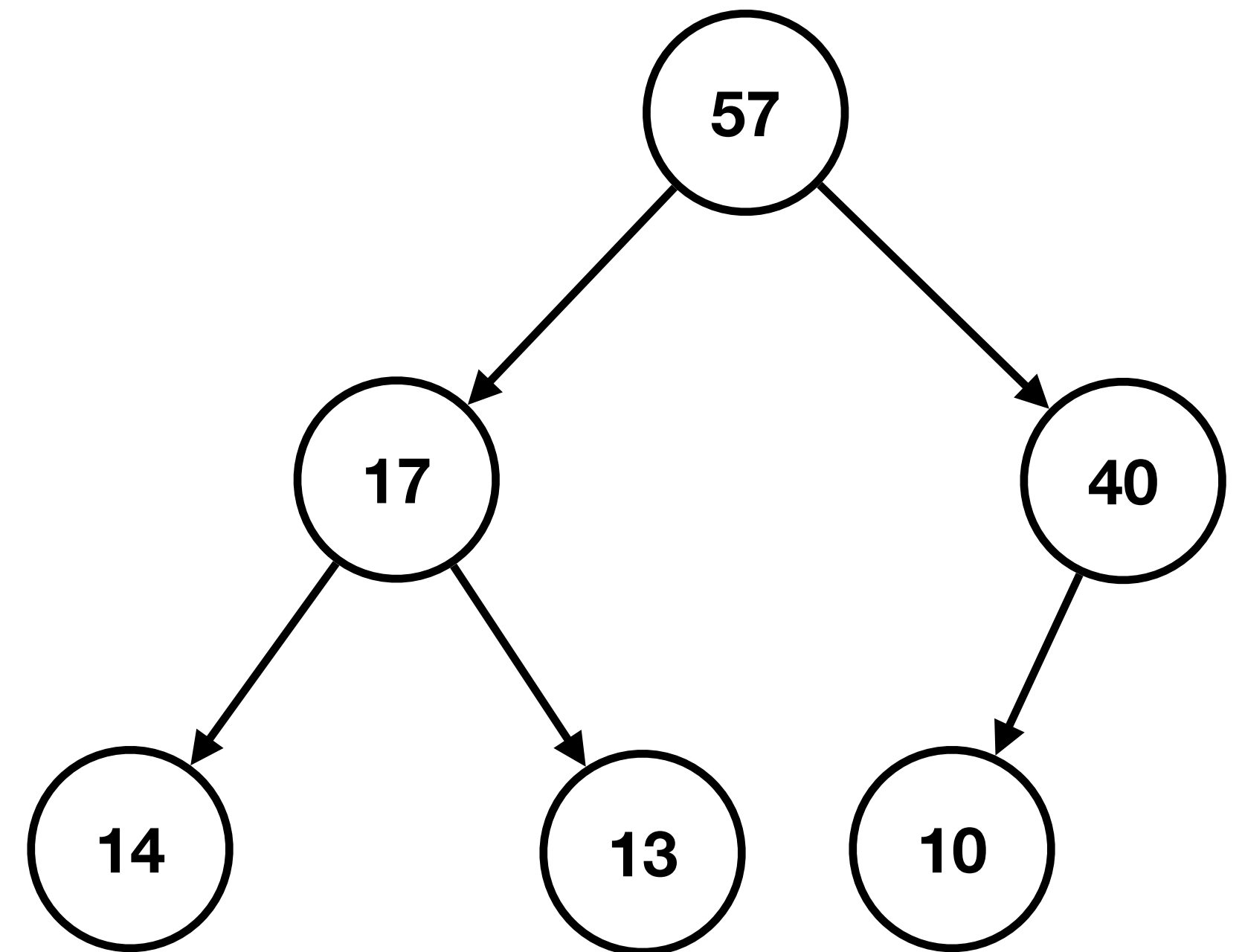
What is the best way to store this?



Heap

What is the best way to store this?

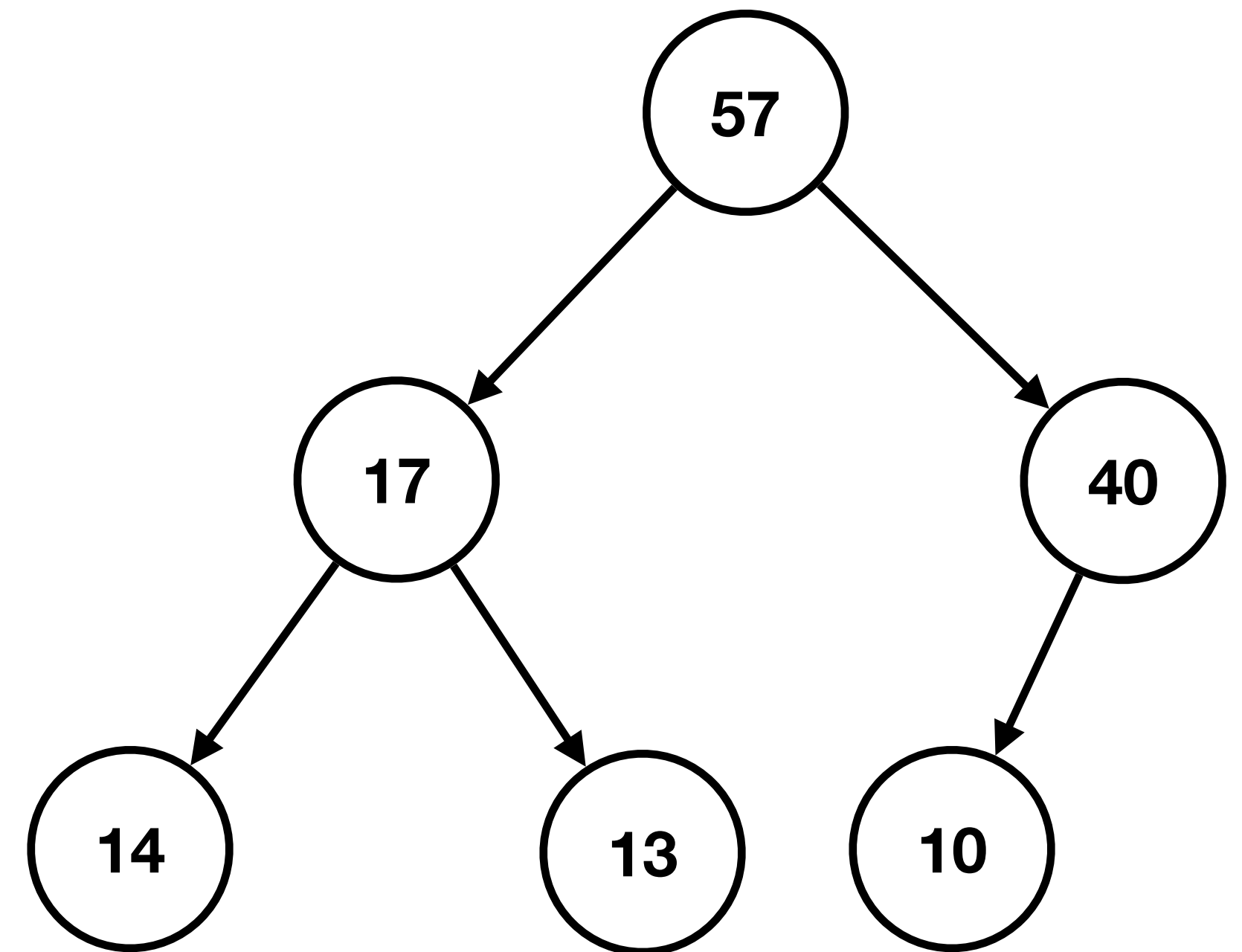
- Could use nodes and pointers...



Heap

What is the best way to store this?

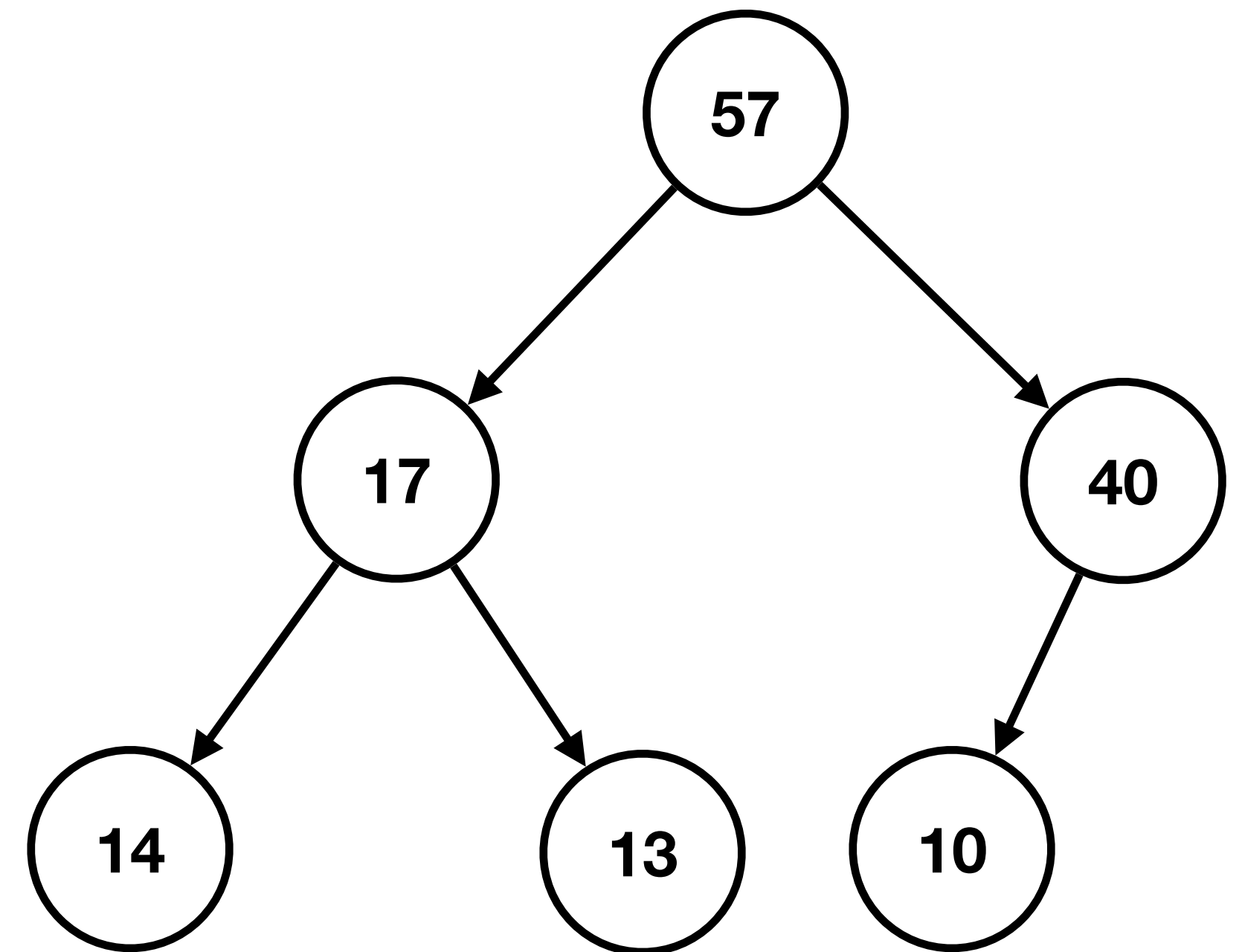
- Could use nodes and pointers...
- Or, we can use a data structure that provides constant-time access to elements:



Heap

What is the best way to store this?

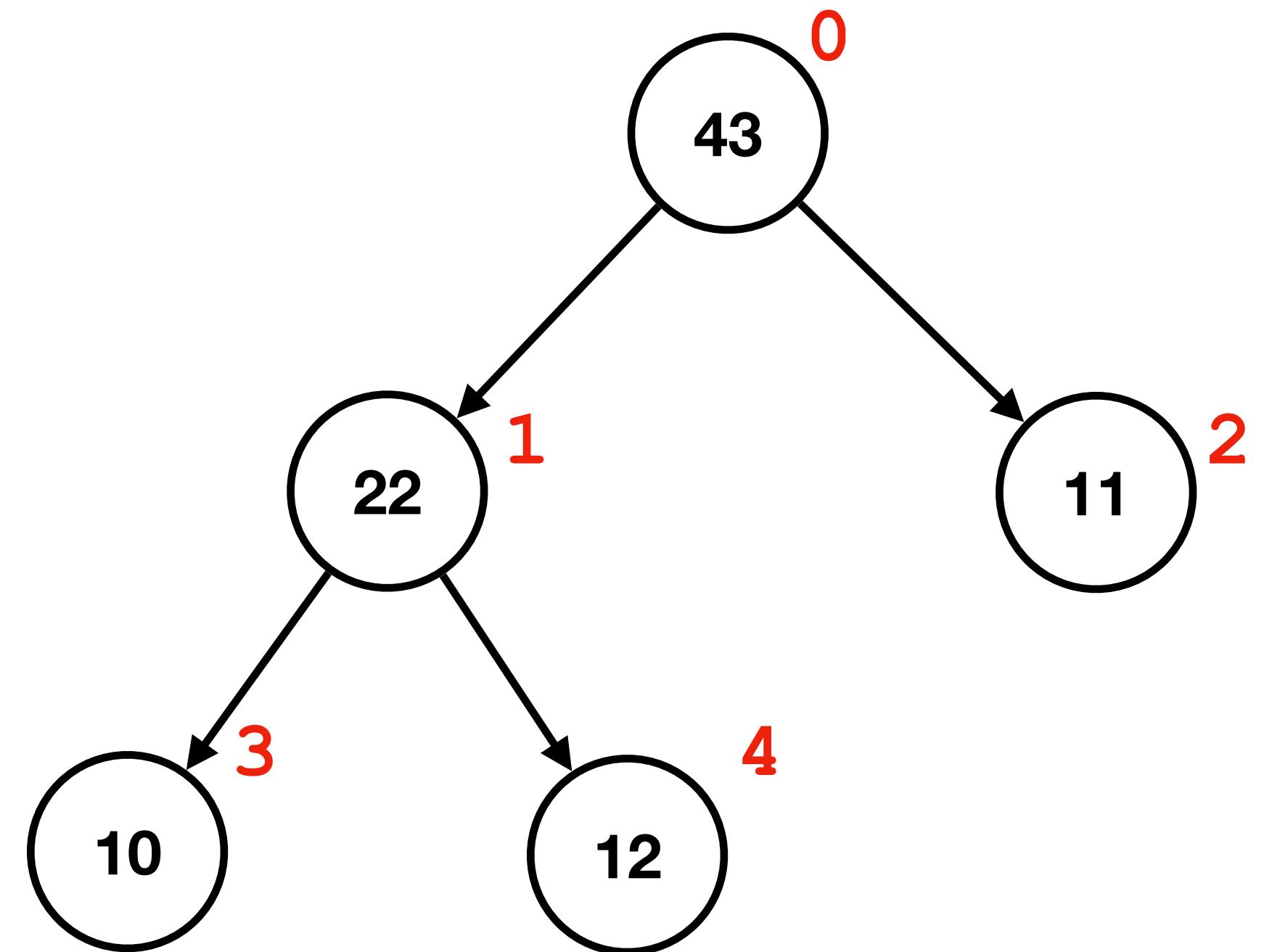
- Could use nodes and pointers...
- Or, we can use a data structure that provides constant-time access to elements:
- array



Heap

What is the best way to store this?

- Being complete means we can store the elements row by row.

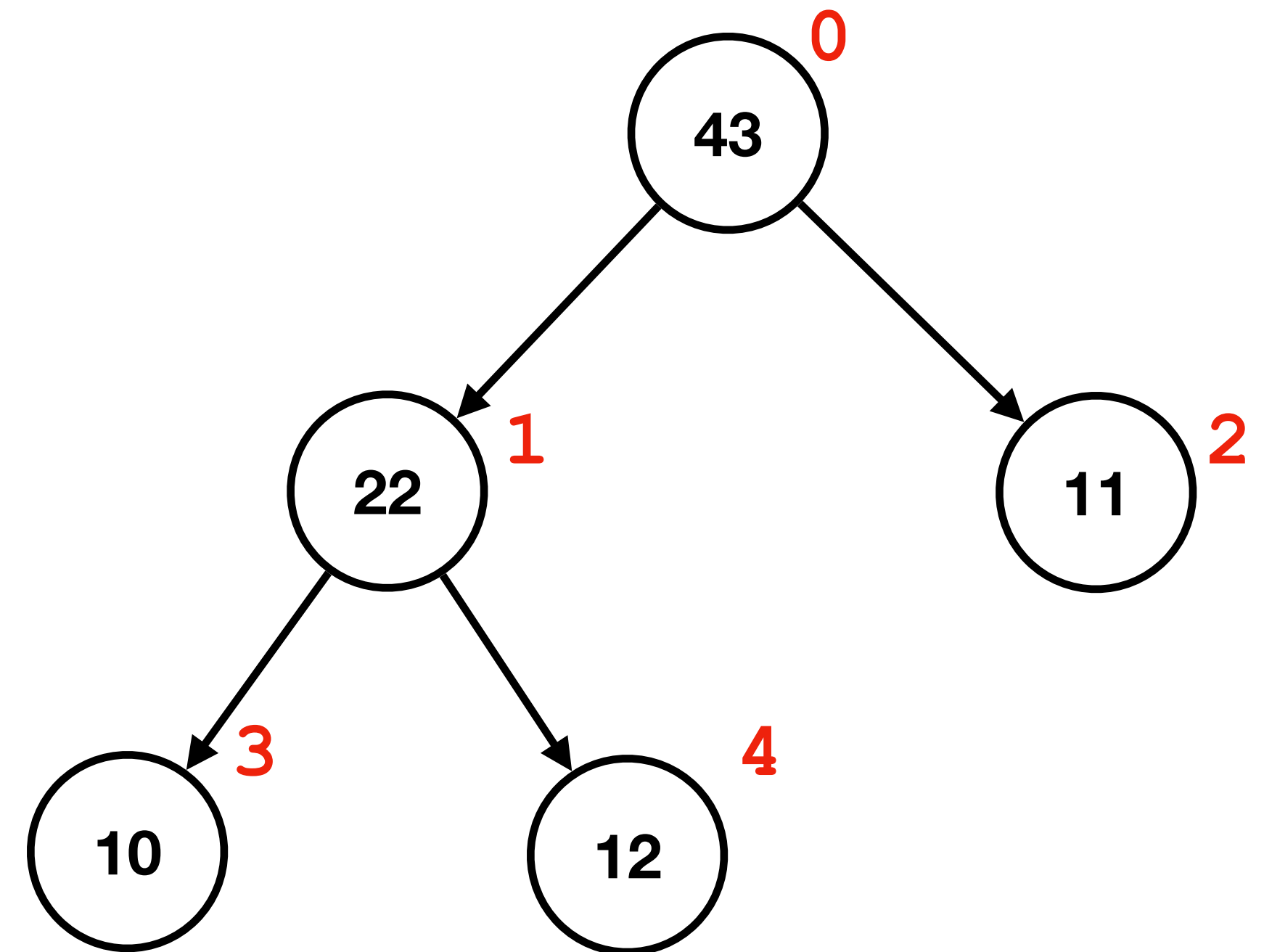


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

What is the best way to store this?

- Root at index 0
- For an element at position i :
 - left child: $2i + 1$
 - right child: $2i + 2$
 - parent: $\lfloor (i - 1)/2 \rfloor$



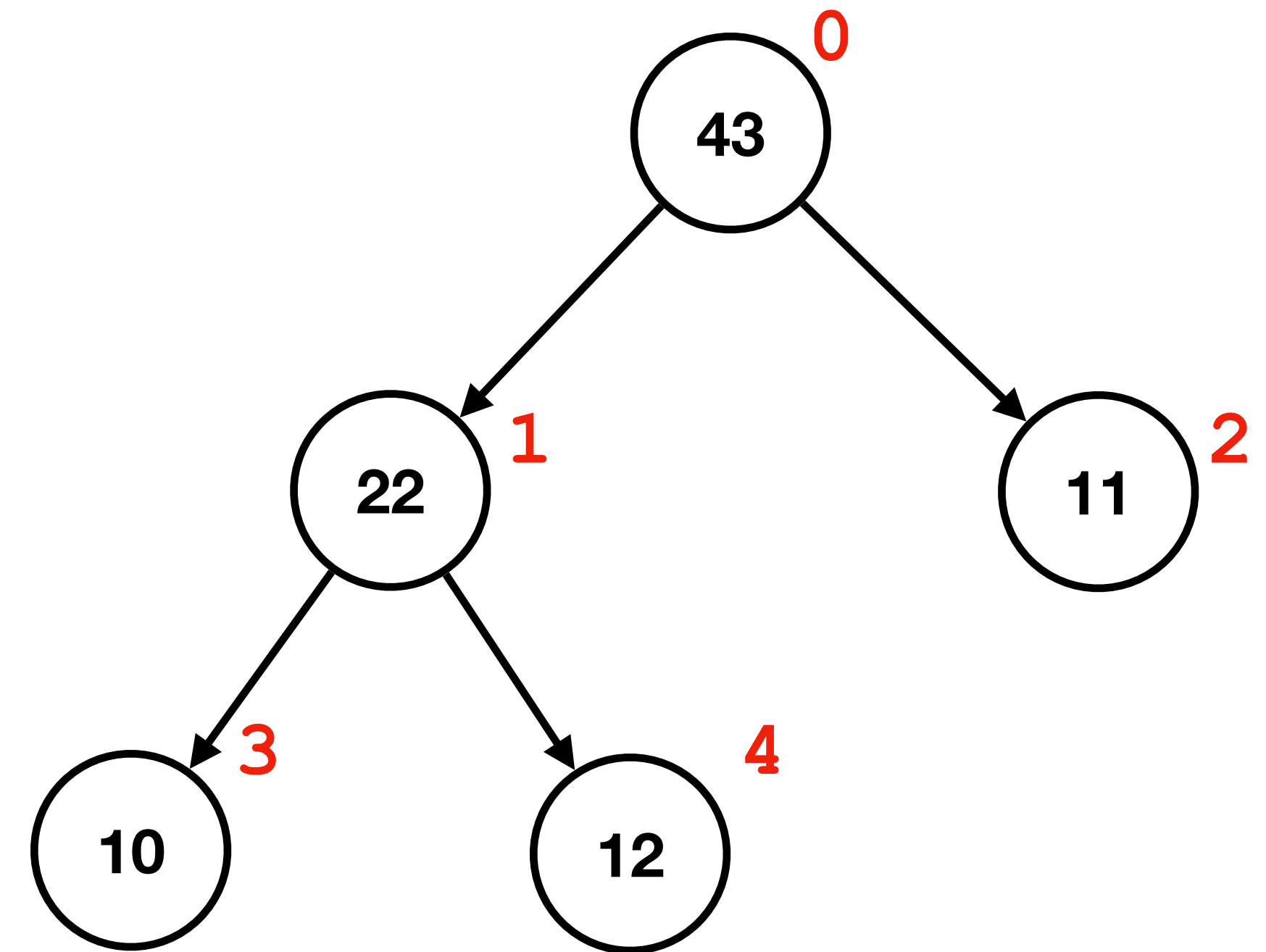
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

What is the best way to store this?

- For an element at position i :
 - left child: $2i + 1$
 - right child: $2i + 2$
 - parent: $\lfloor (i - 1) / 2 \rfloor$

```
int self = heap[i];  
int left_child = heap[2 * i + 1];  
int right_child = heap[2 * i + 2];  
int parent = heap[(i - 1) / 2];
```

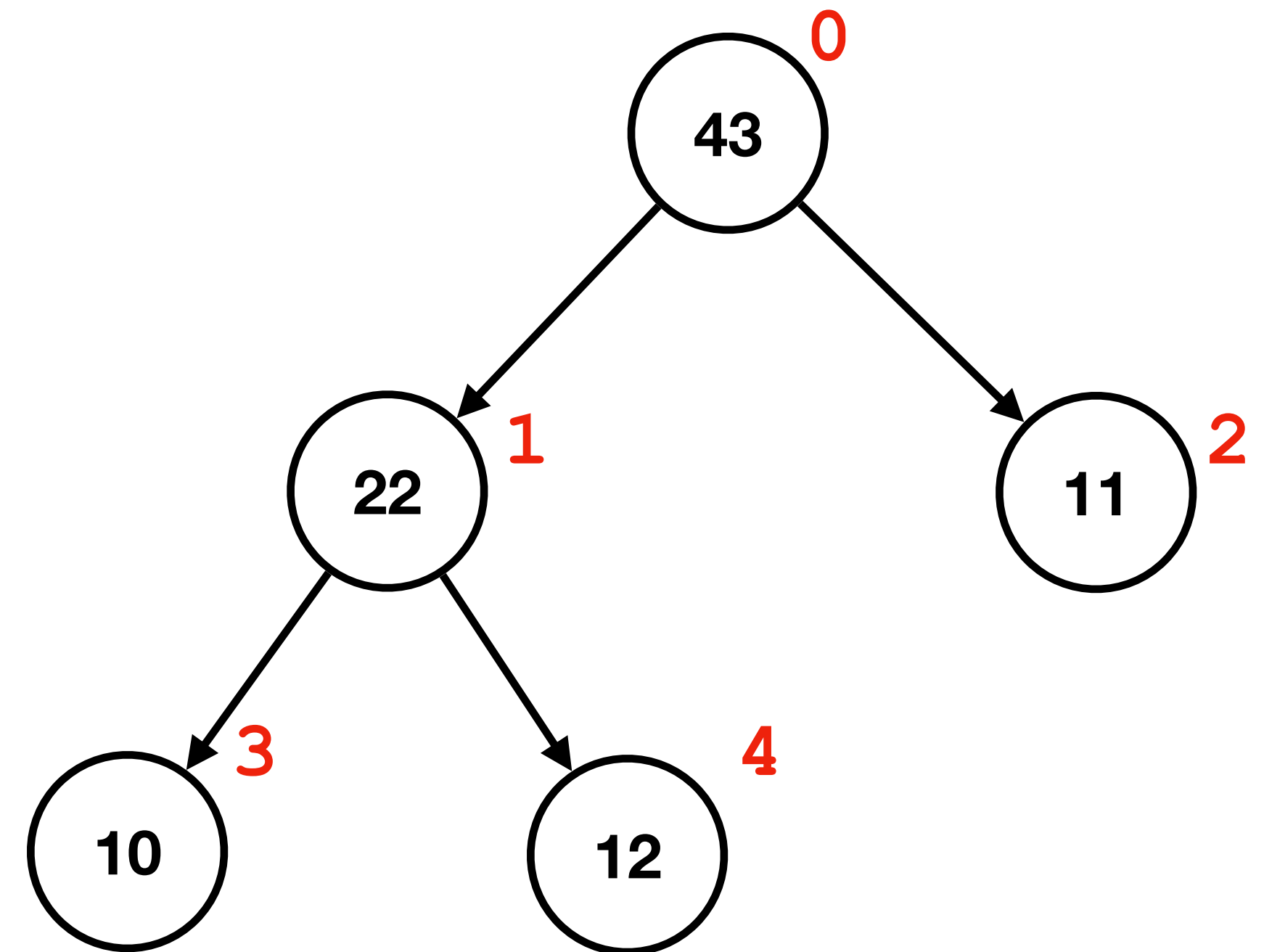


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

What is the best way to store this?

- Possible exam questions:
 - Given tree, write array
 - Given array, draw tree
 - Identify parent, left, right children

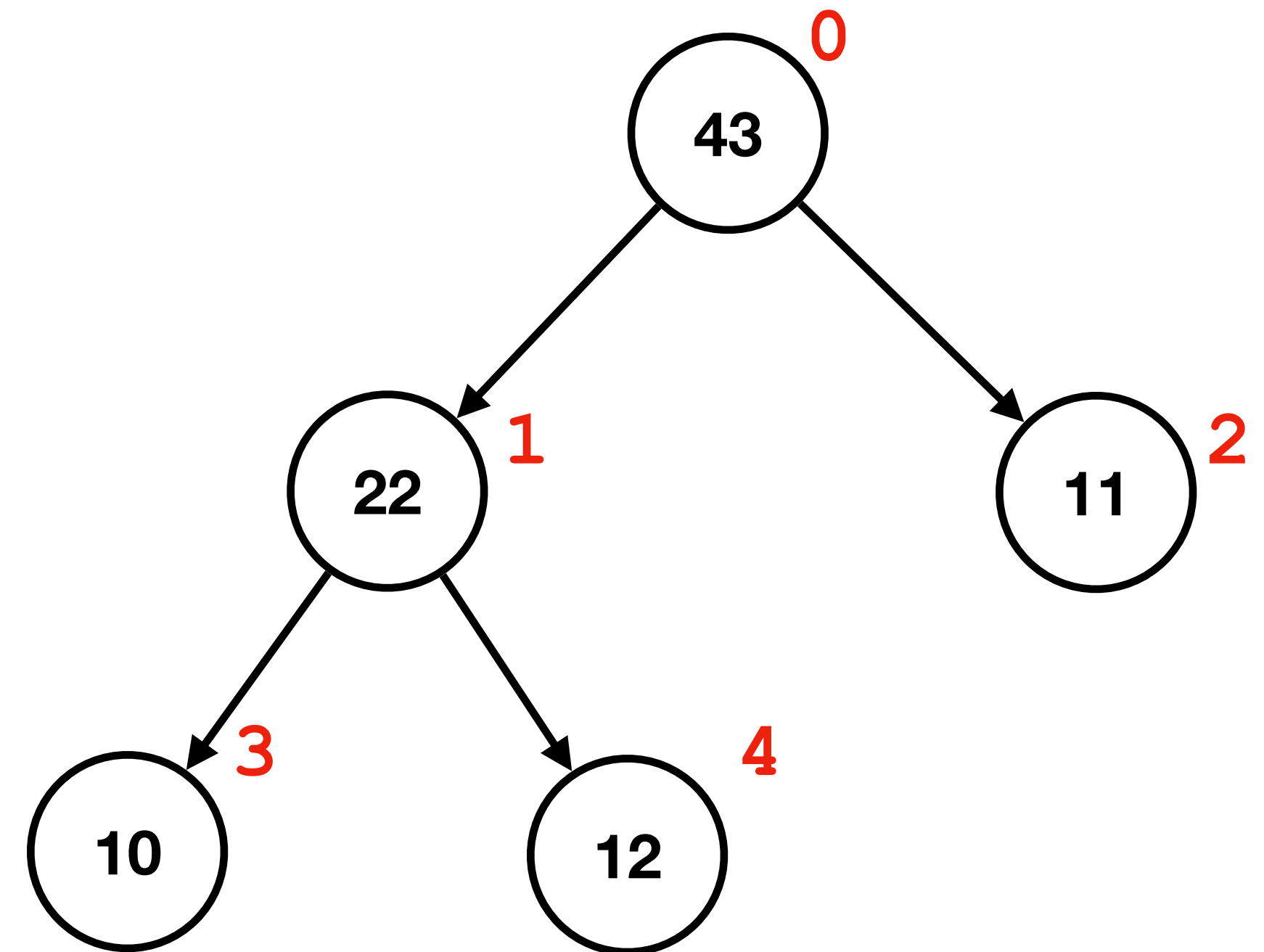


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

- `get_top()`
- $O(1)$ operation: `h[0]`
- How about insertion and removal?

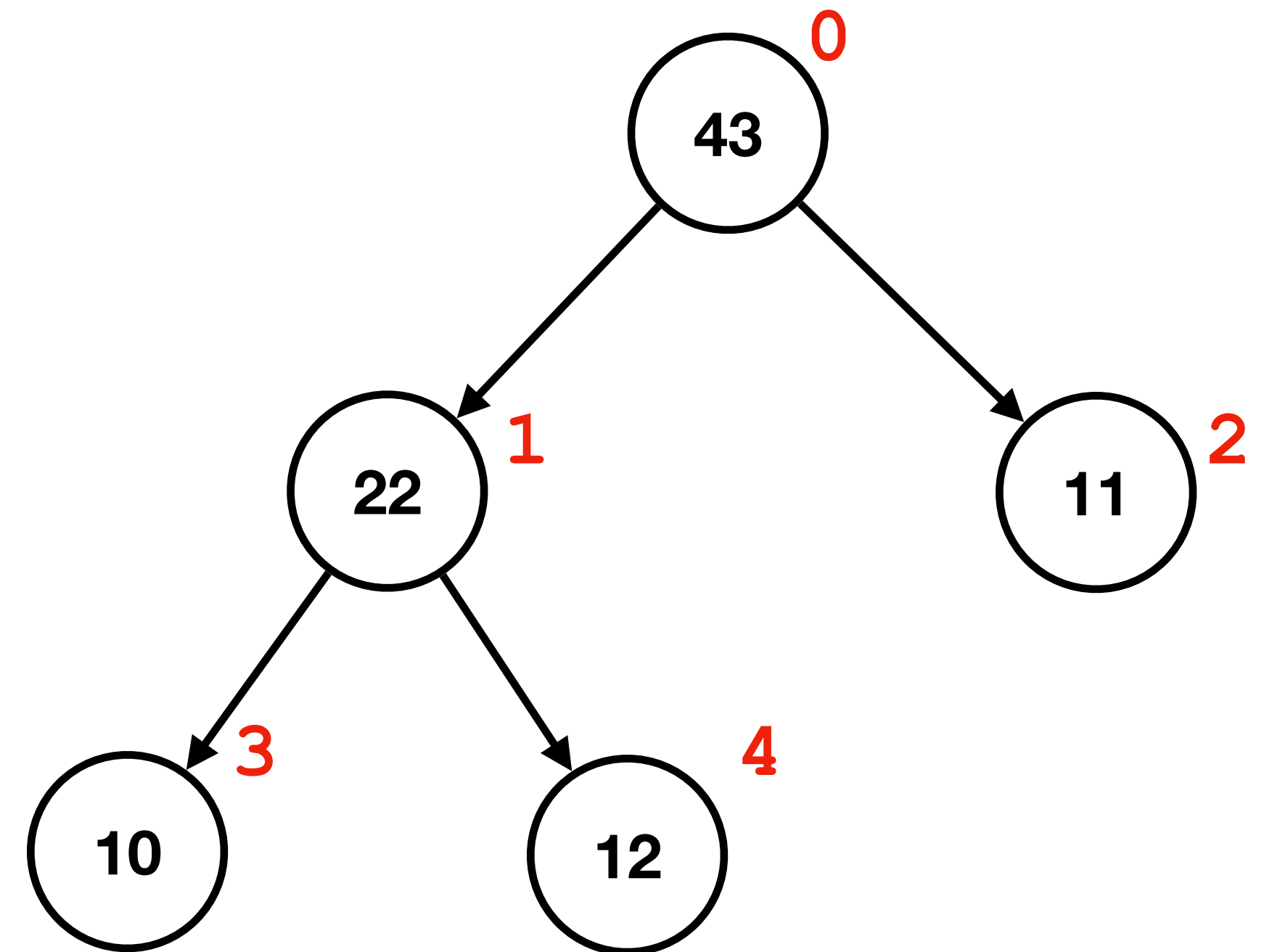


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

- Insertion and removal scheme:
 1. Restore shape property first, ignoring value property
 2. Restore value property without changing the shape

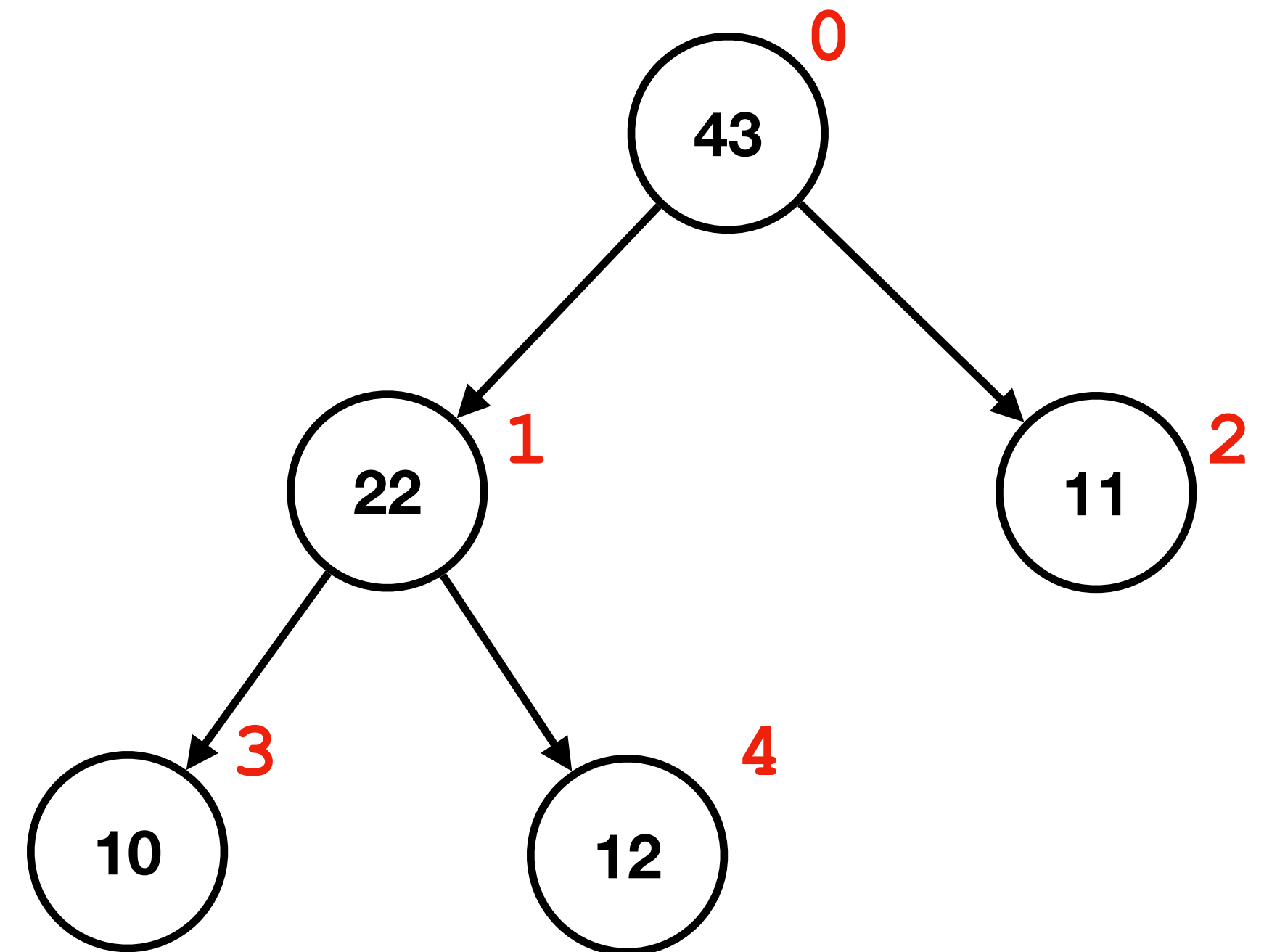


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

- `insert(50)`
- Where should we put the element to maintain the *shape* property?

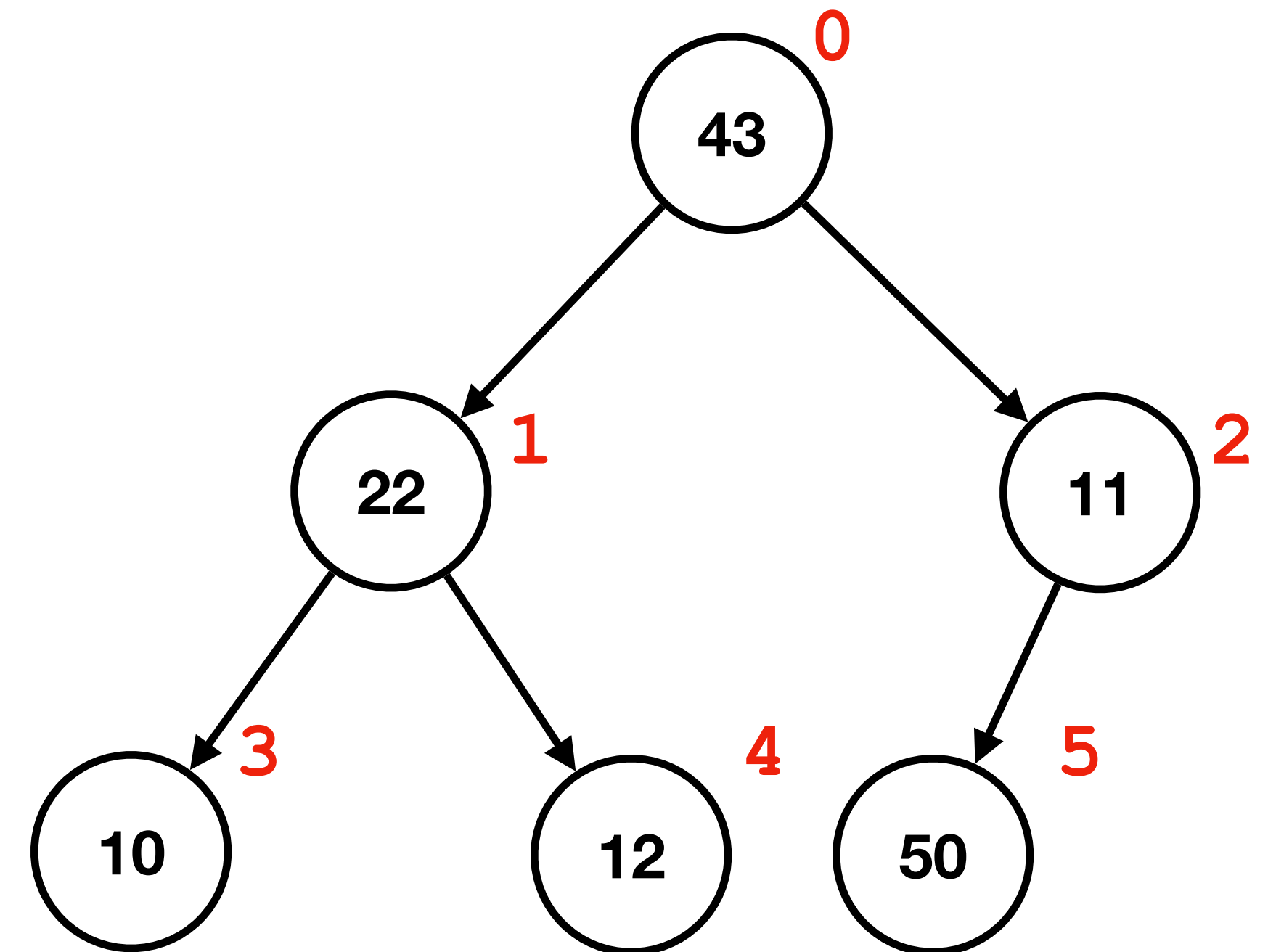


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

- `insert(50)`
 - Where should we put the element to maintain the shape property?
1. Insert item at element `h[heap_size]` (very likely destroying the heap property)
 2. Increment heap size
 3. Bubble up until you get to root:
 - Swap with its parent if in incorrect order

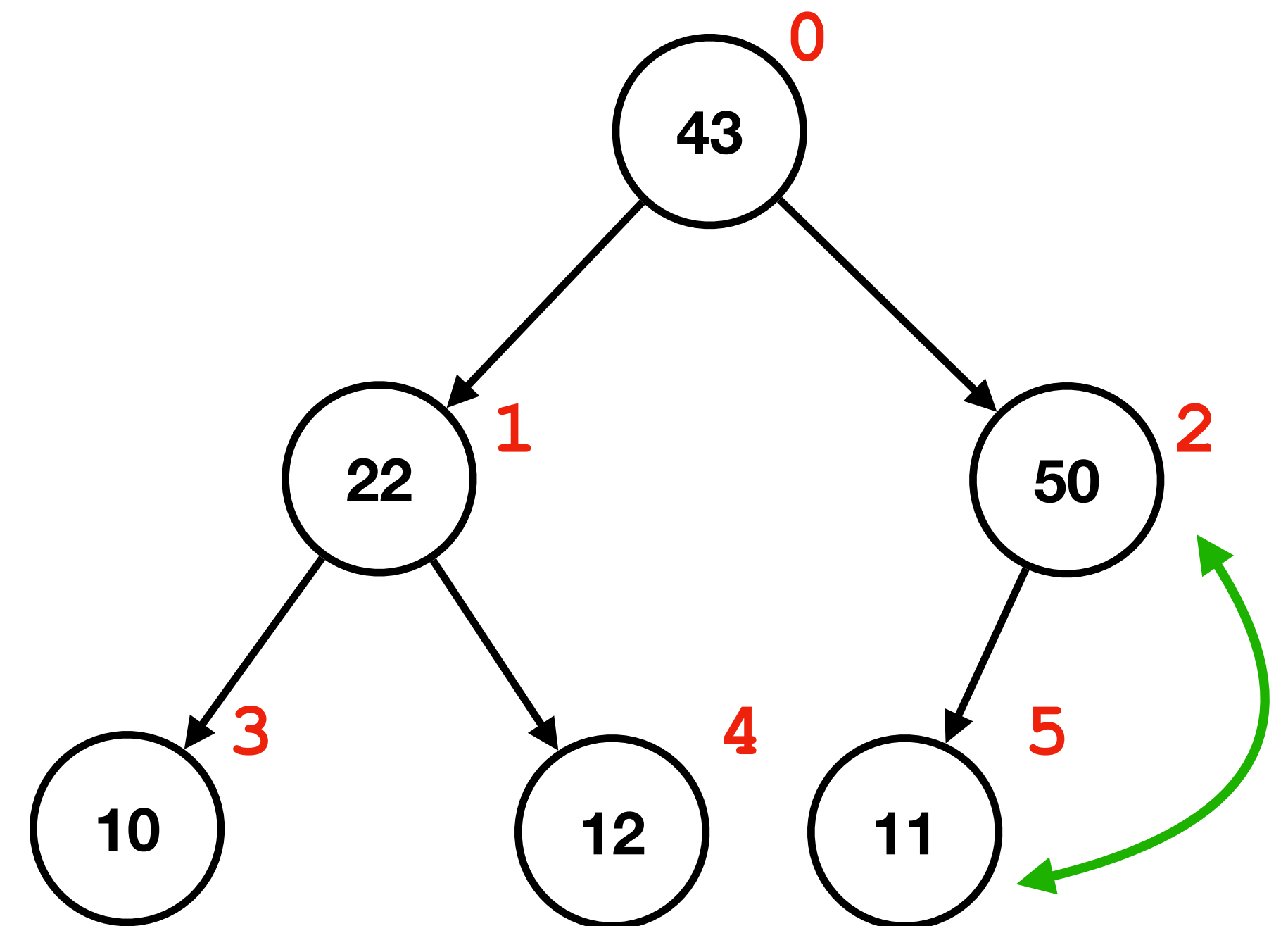


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12	50					

Heap

Operations

- `insert(50)`
 - Where should we put the element to maintain the shape property?
1. Insert item at element `h[heap_size + 1]`
(very likely destroying the heap property)
 2. Increment heap size
 3. Bubble up until you get to root:
 - Swap with its parent if in incorrect order

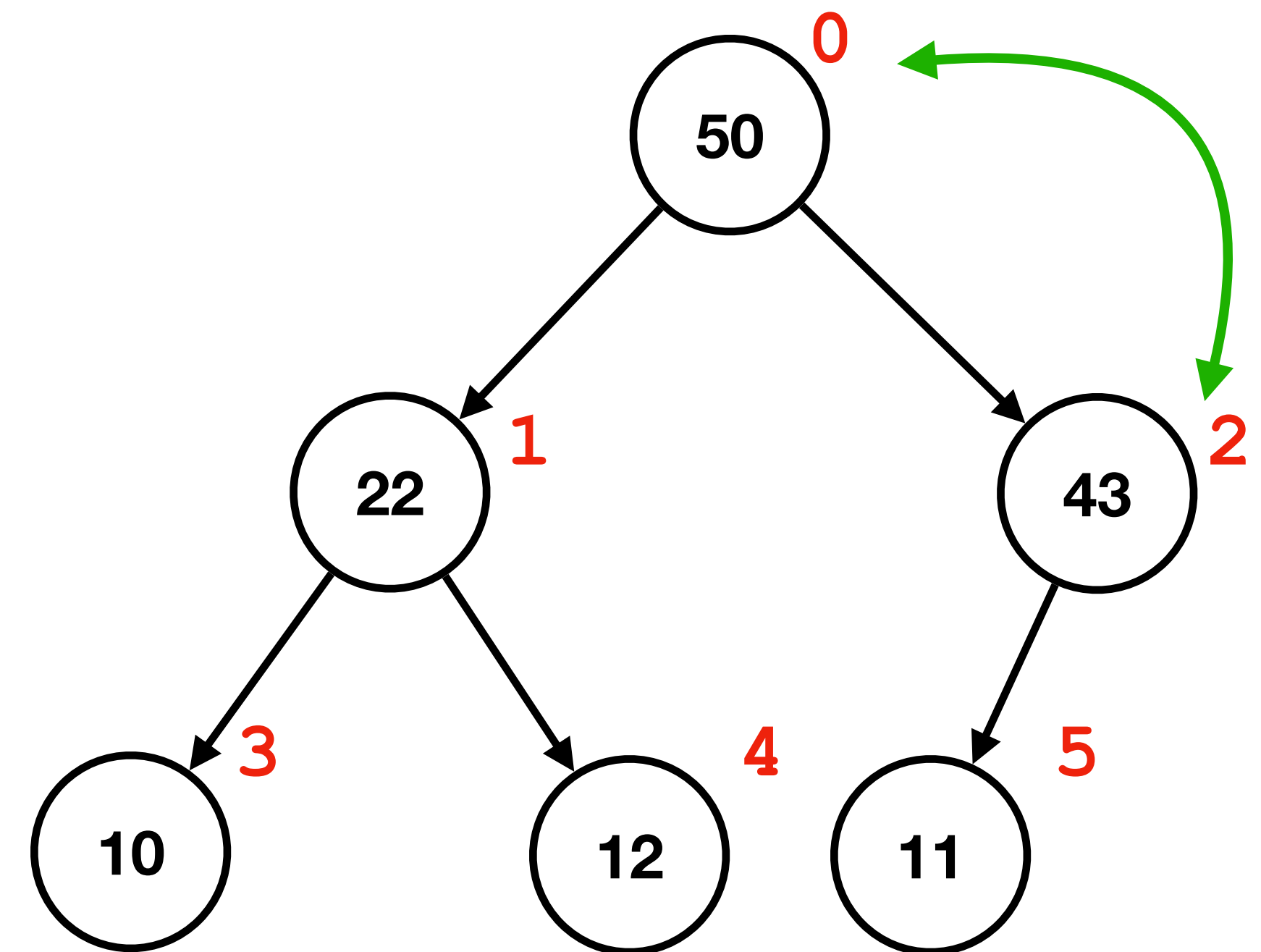


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	50	10	12	11					

Heap

Operations

- `insert(50)`
 - Where should we put the element to maintain the *shape* property?
1. Insert item at element `h[heap_size + 1]`
(very likely destroying the heap property)
 2. Increment heap size
 3. Bubble up until you get to root:
 - Swap with its parent if in incorrect order

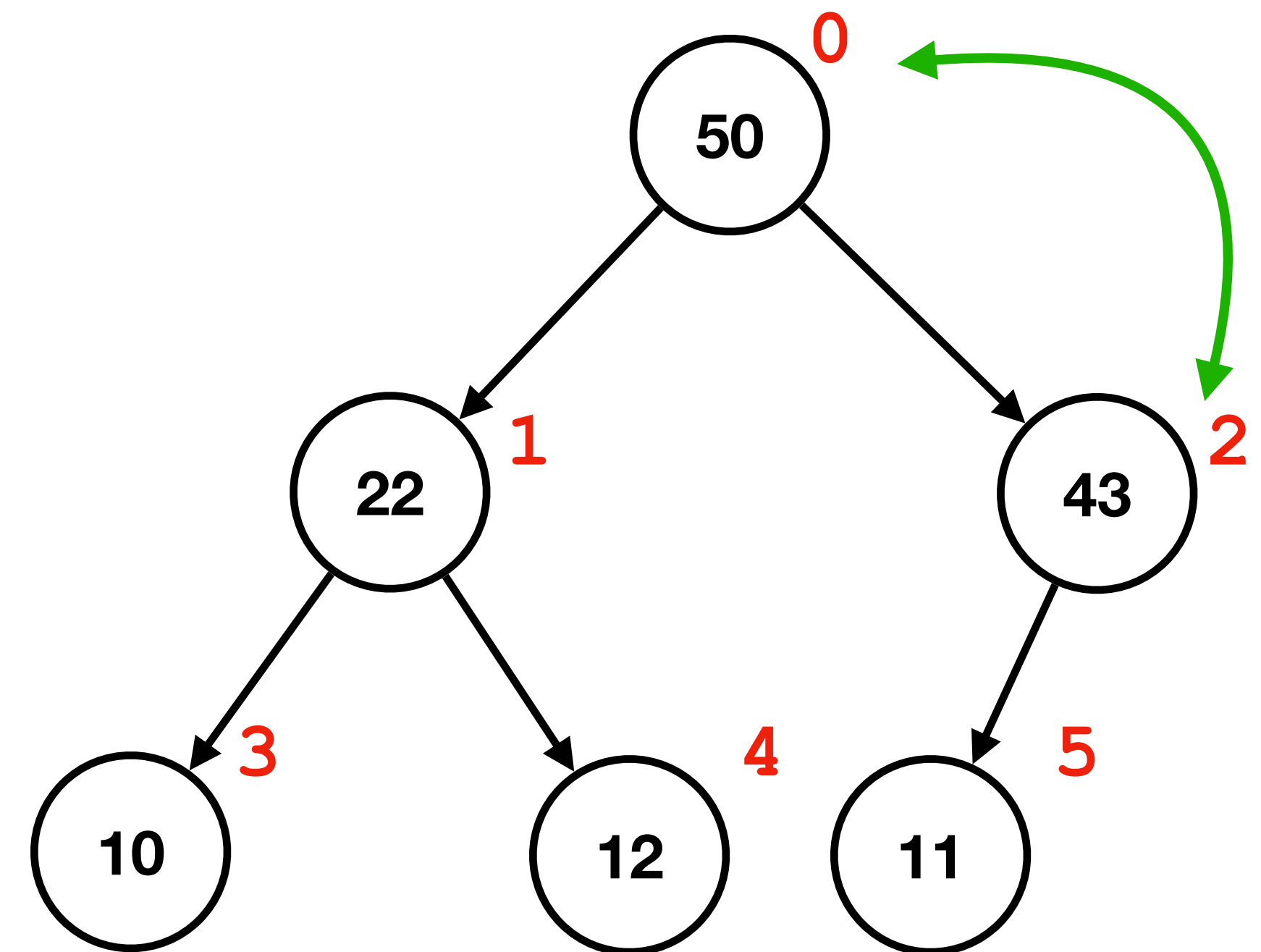


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
50	22	43	10	12	11					

Heap

Operations

- `insert(50)`
 - Worst case complexity: $O(\log n)$
1. Insert item at element `h[heap_size]` (very likely destroying the heap property)
 2. Increment heap size
 3. Bubble up until you get to root:
 - Swap with its parent if in incorrect order



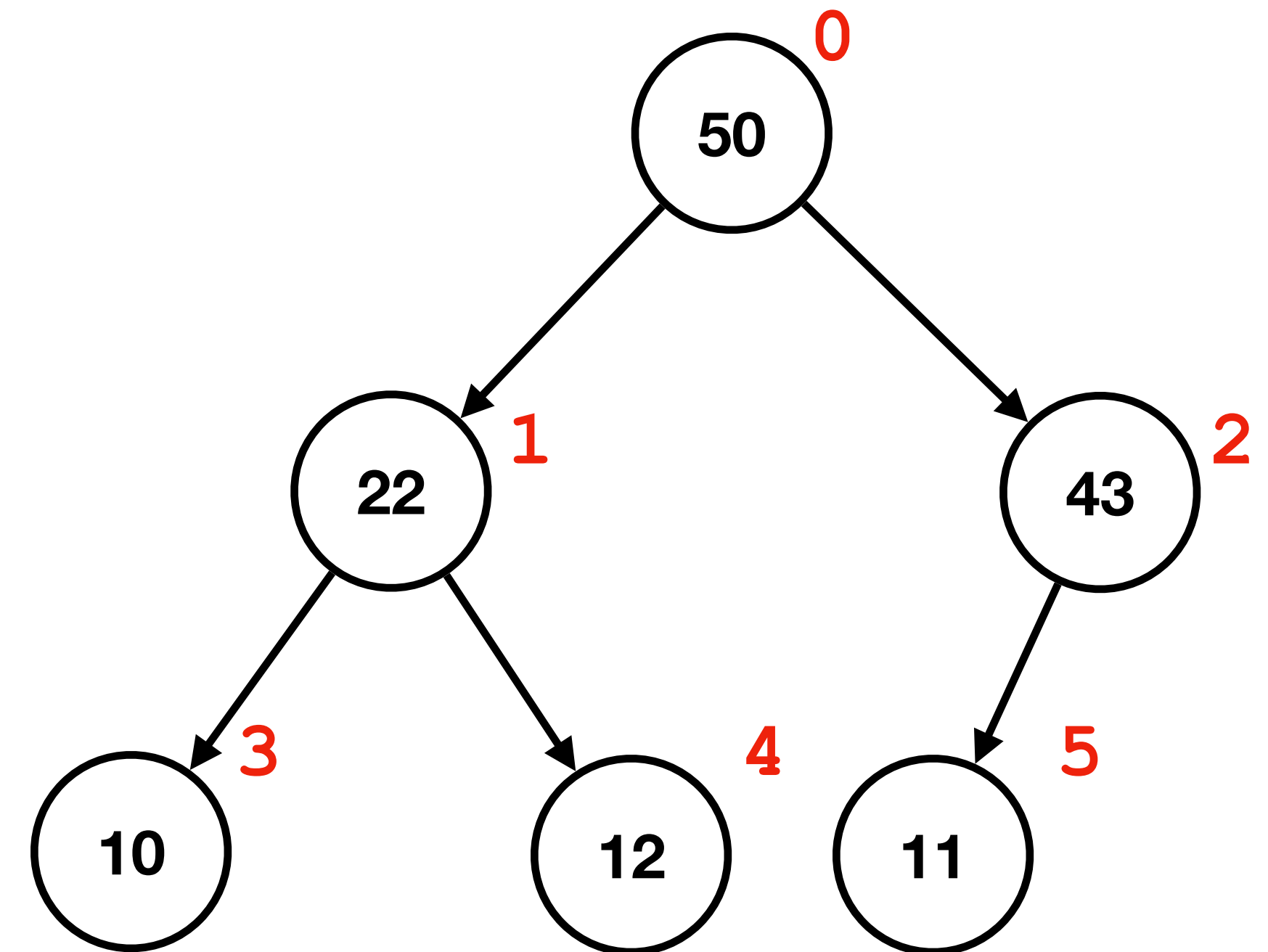
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
50	22	43	10	12	11					

Heap

Operations

- `remove_top()`

1. Remove root (save for later return)



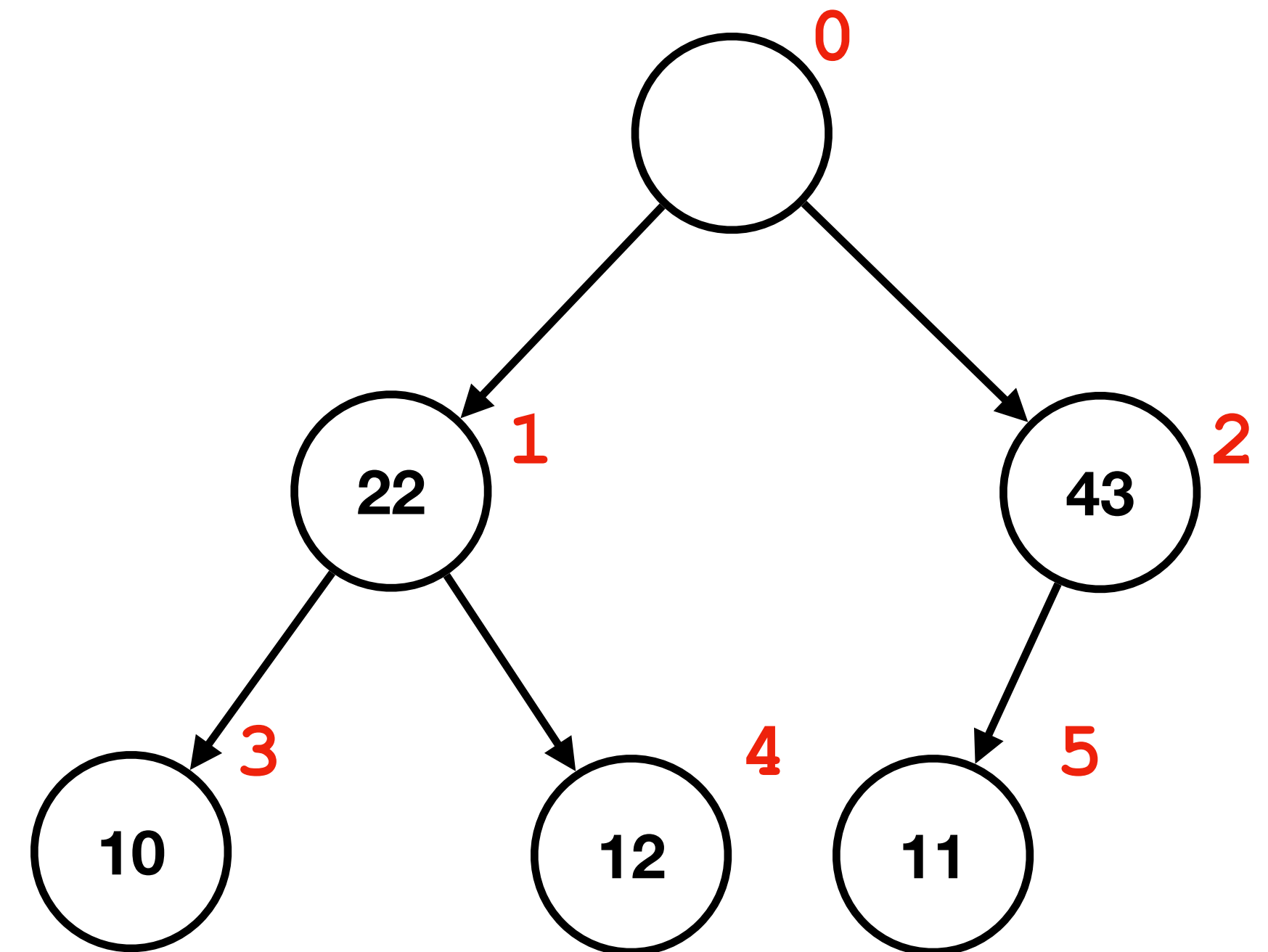
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
50	22	43	10	12	11					

Heap

Operations

- `remove_top()`

1. Remove root (save for later return)

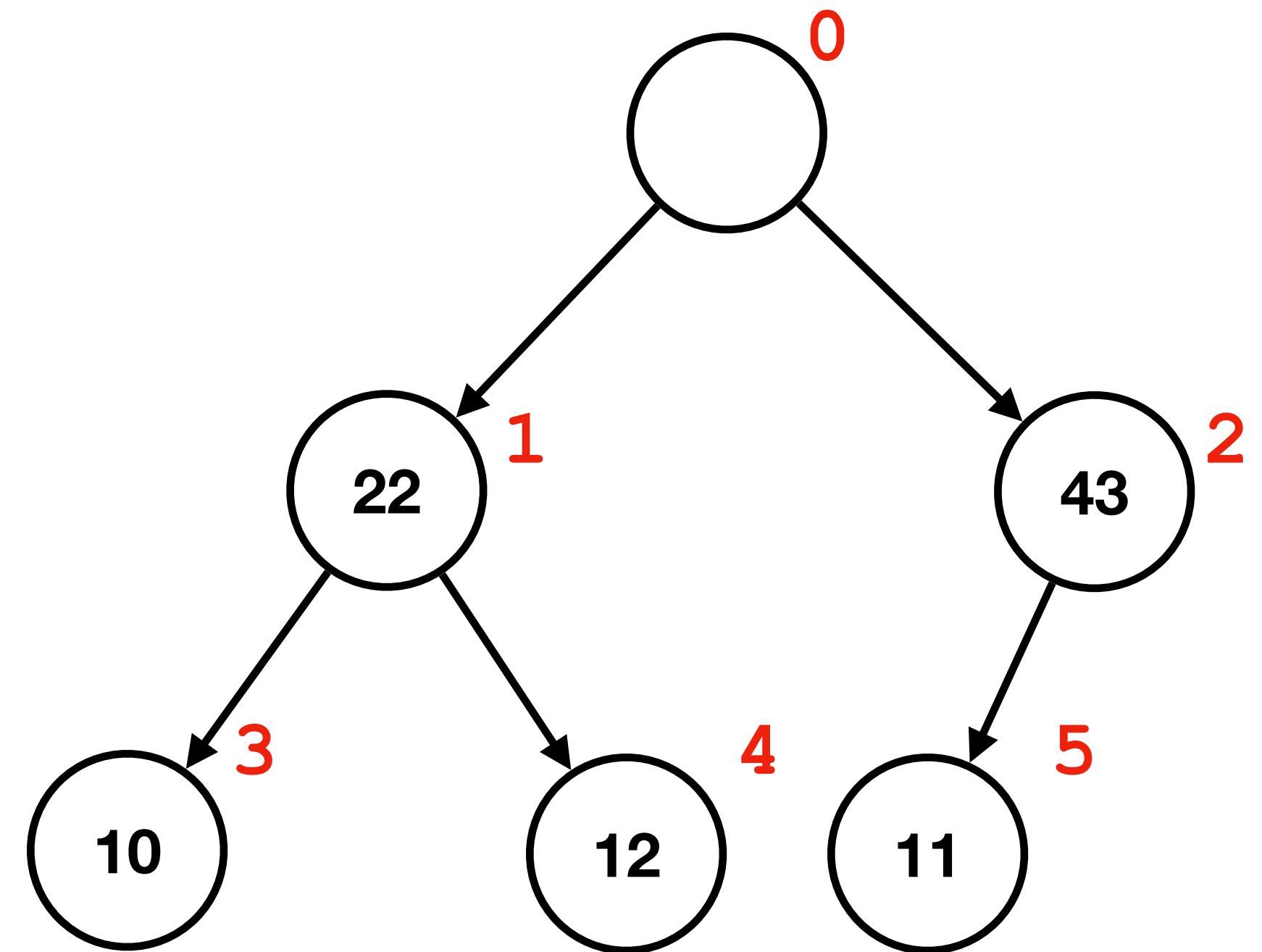


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	22	43	10	12	11					

Heap

Operations

- `remove_top()`
1. Remove root (save for later return)
 2. Maintain shape: replace root with last element in the array

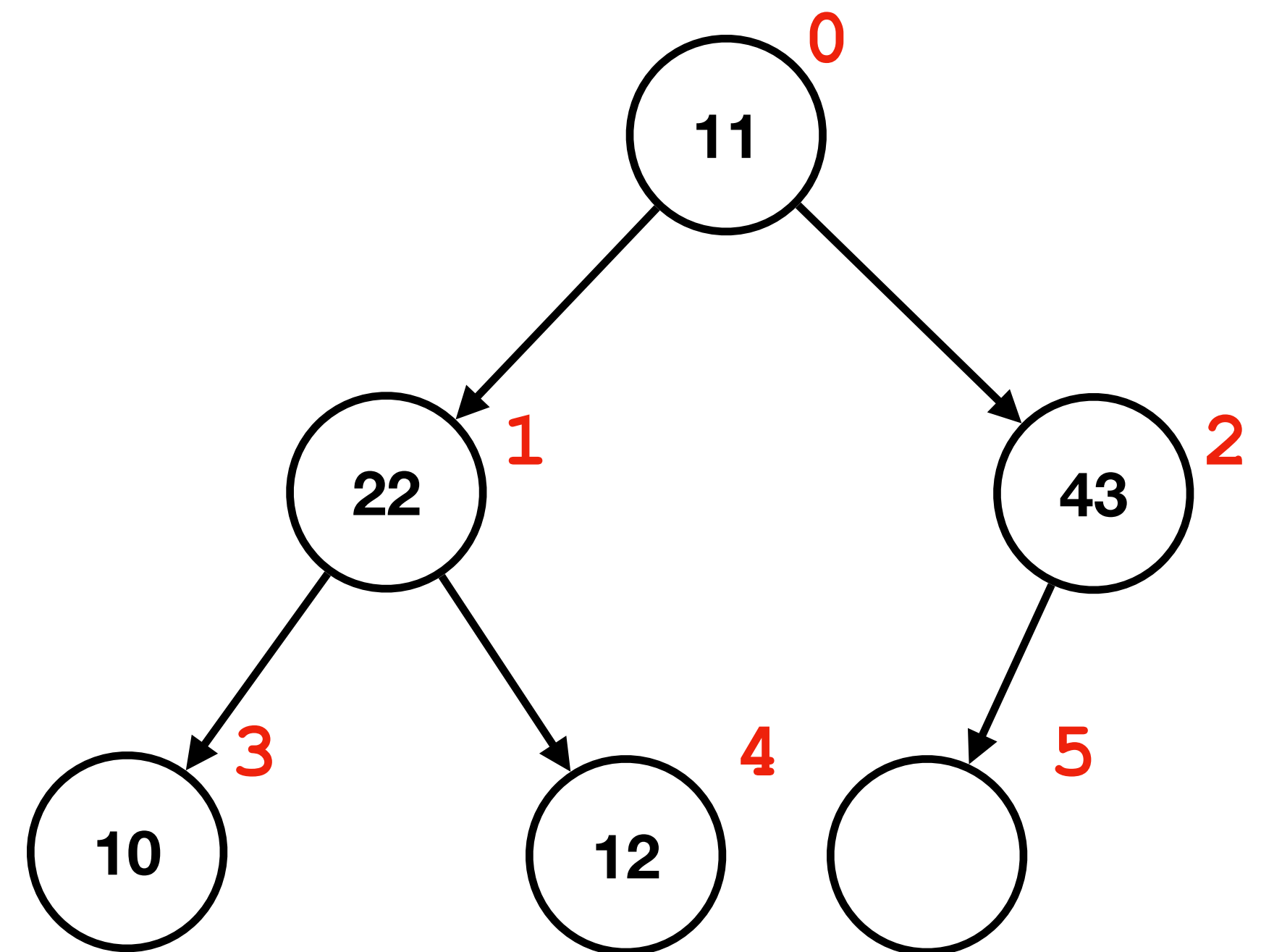


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
	22	43	10	12	11					

Heap

Operations

- `remove_top()`
1. Remove root (save for later return)
 2. Maintain shape: replace root with last element in the array
(Decrement `heap_size`)

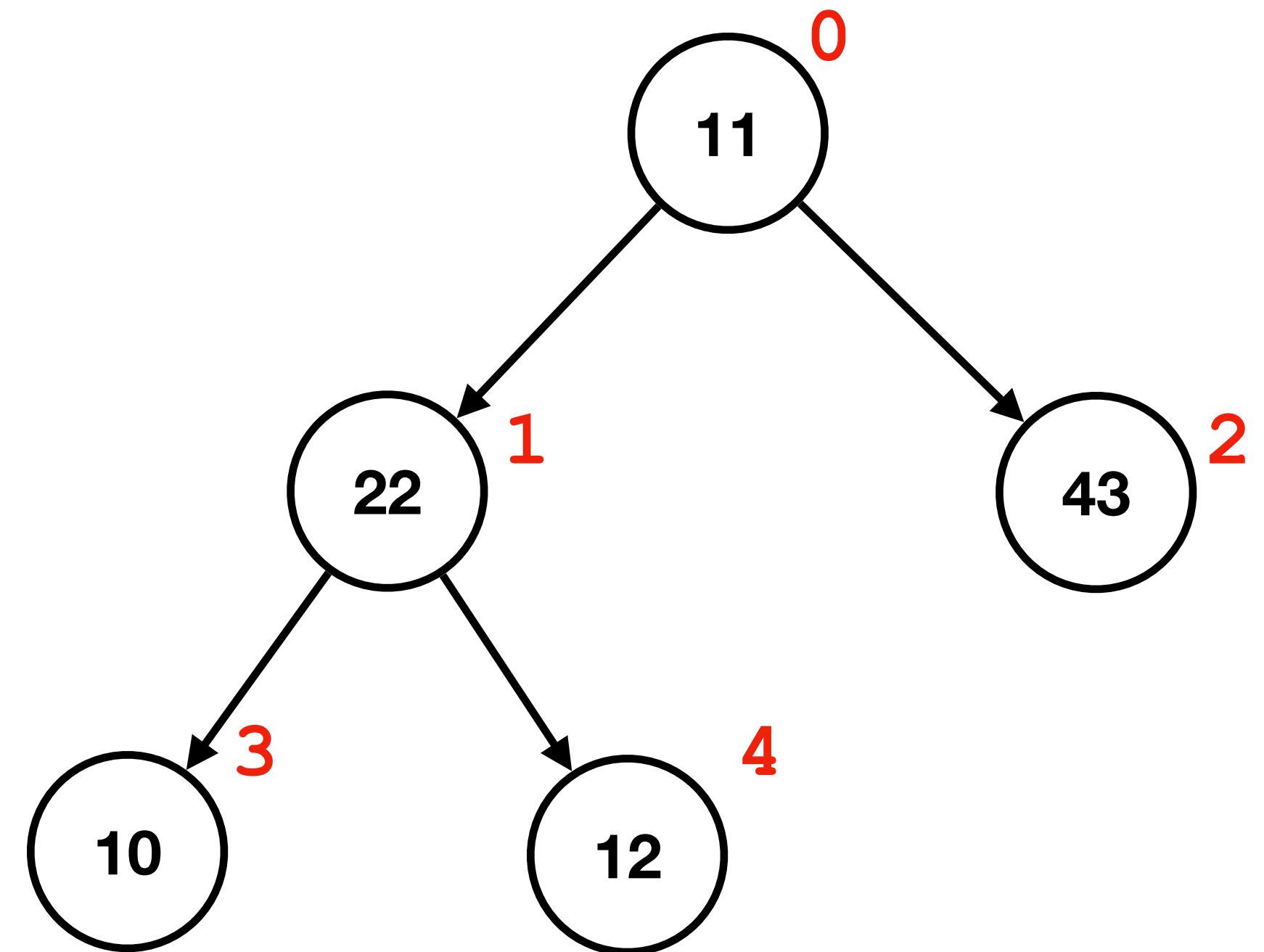


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
11	22	43	10	12						

Heap

Operations

- `remove_top()`
1. Remove root (save for later return)
 2. Maintain shape: replace root with last element in the array
(Decrement `heap_size`)

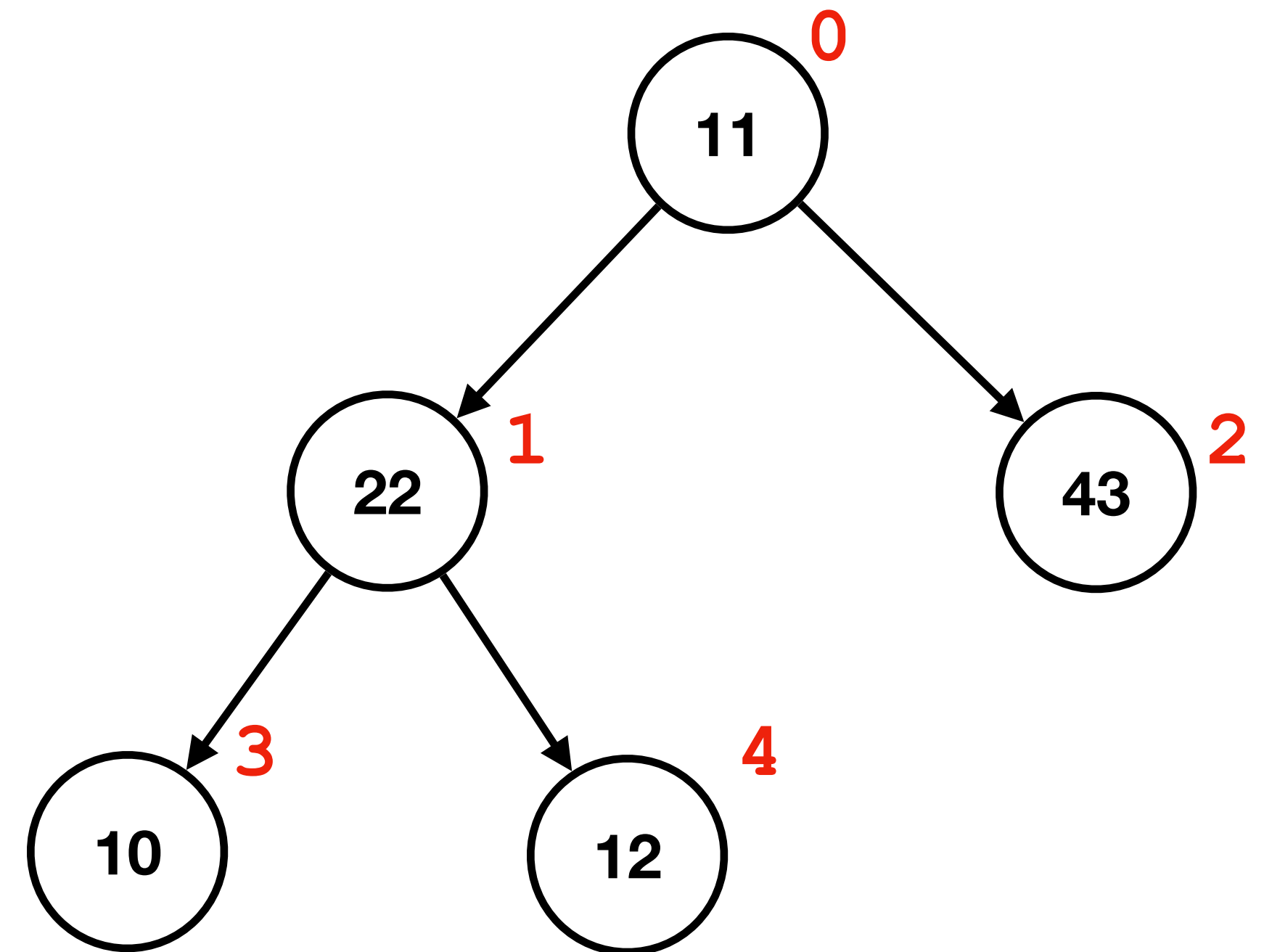


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
11	22	43	10	12						

Heap

Operations

- `remove_top()`
1. Remove root (save for later return)
 2. Maintain shape: replace root with last element in the array
(Decrement `heap_size`)
 3. Sink down:
 1. If \geq both of its children, if correct stop
 2. If not, swapped with the larger

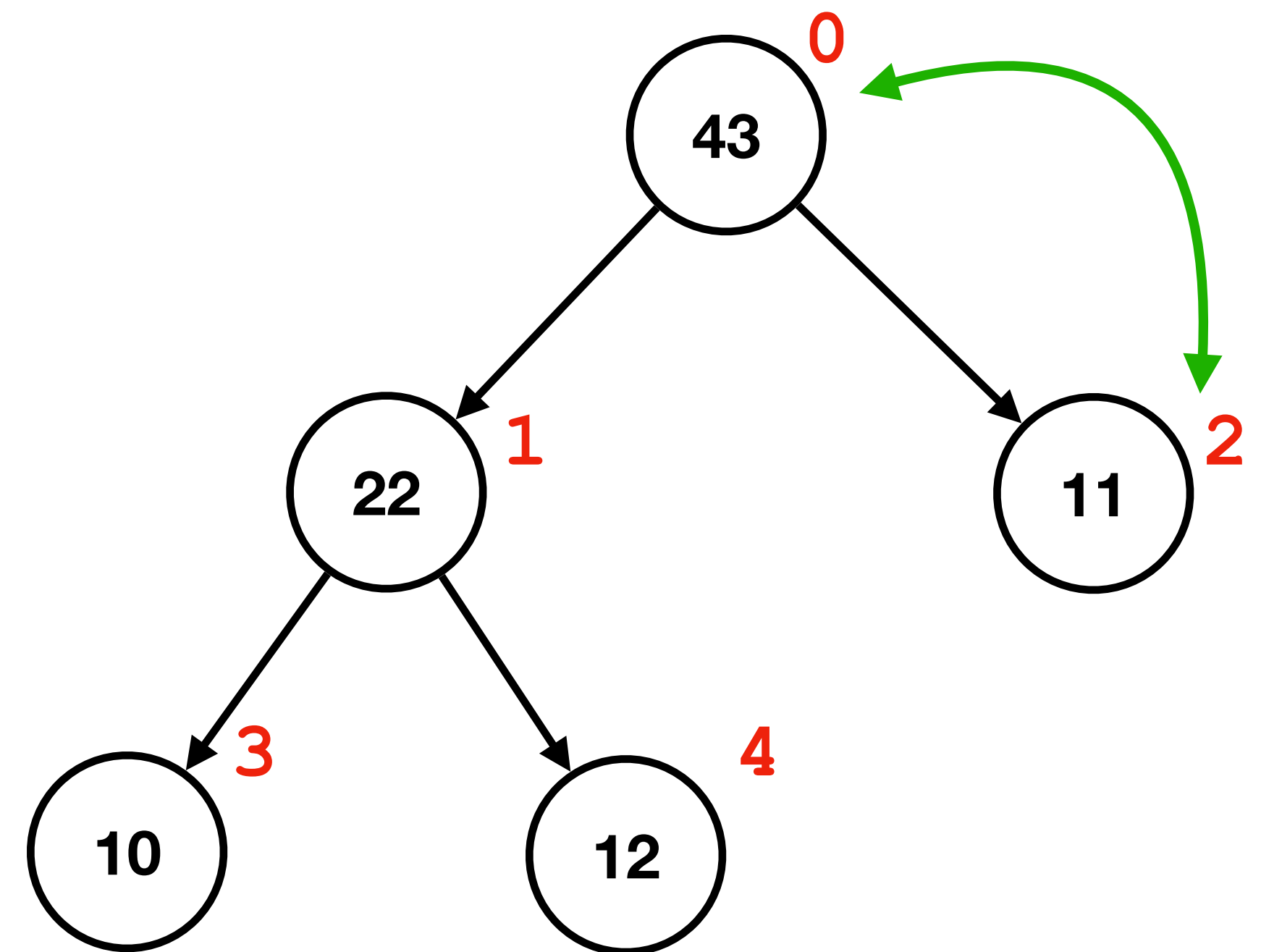


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
11	22	43	10	12						

Heap

Operations

- `remove_top()`
1. Remove root (save for later return)
 2. Maintain shape: replace root with last element in the array
(Decrement `heap_size`)
 3. Sink down:
 1. If \geq both of its children, if correct stop
 2. If not, swapped with the larger



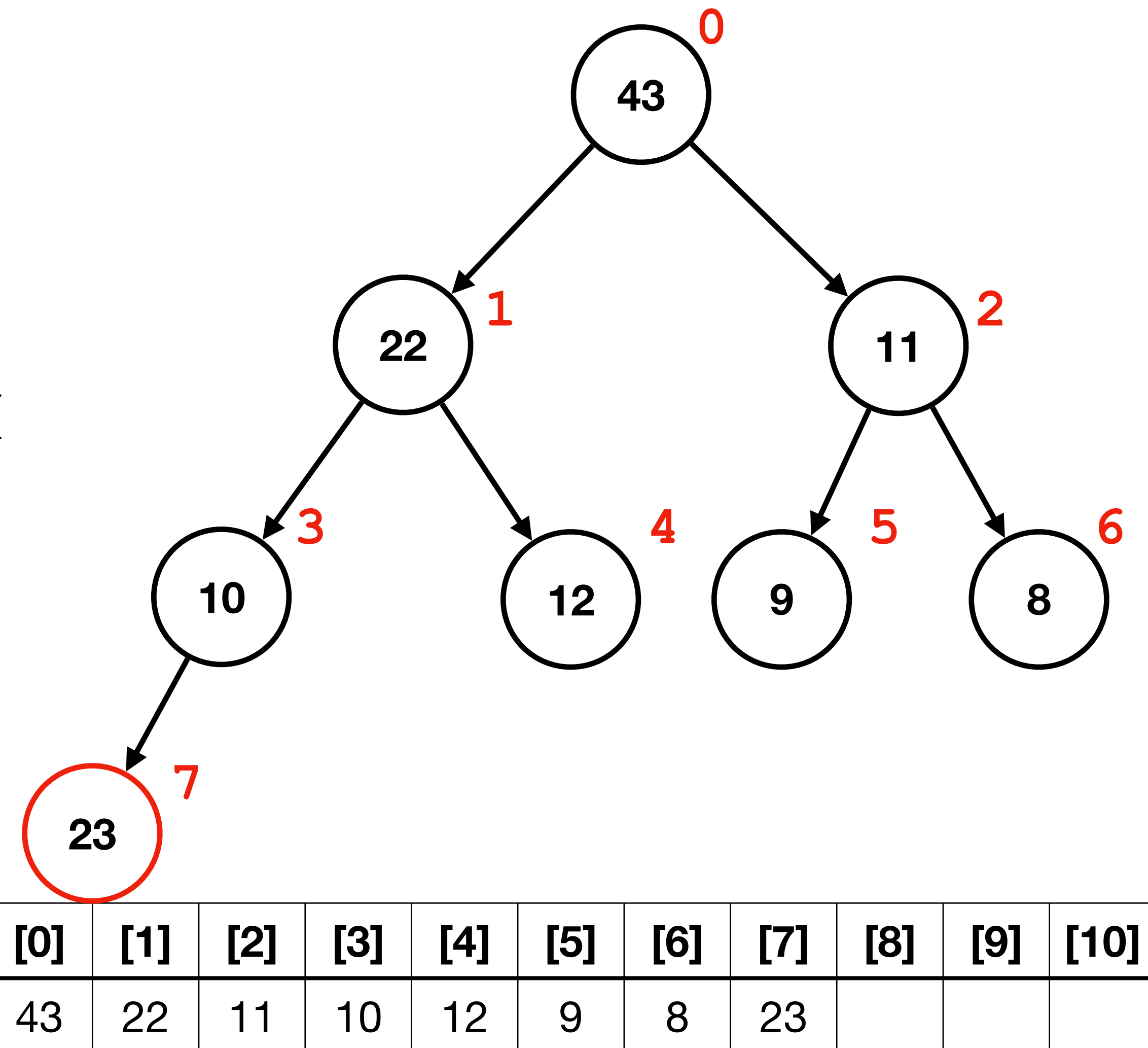
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

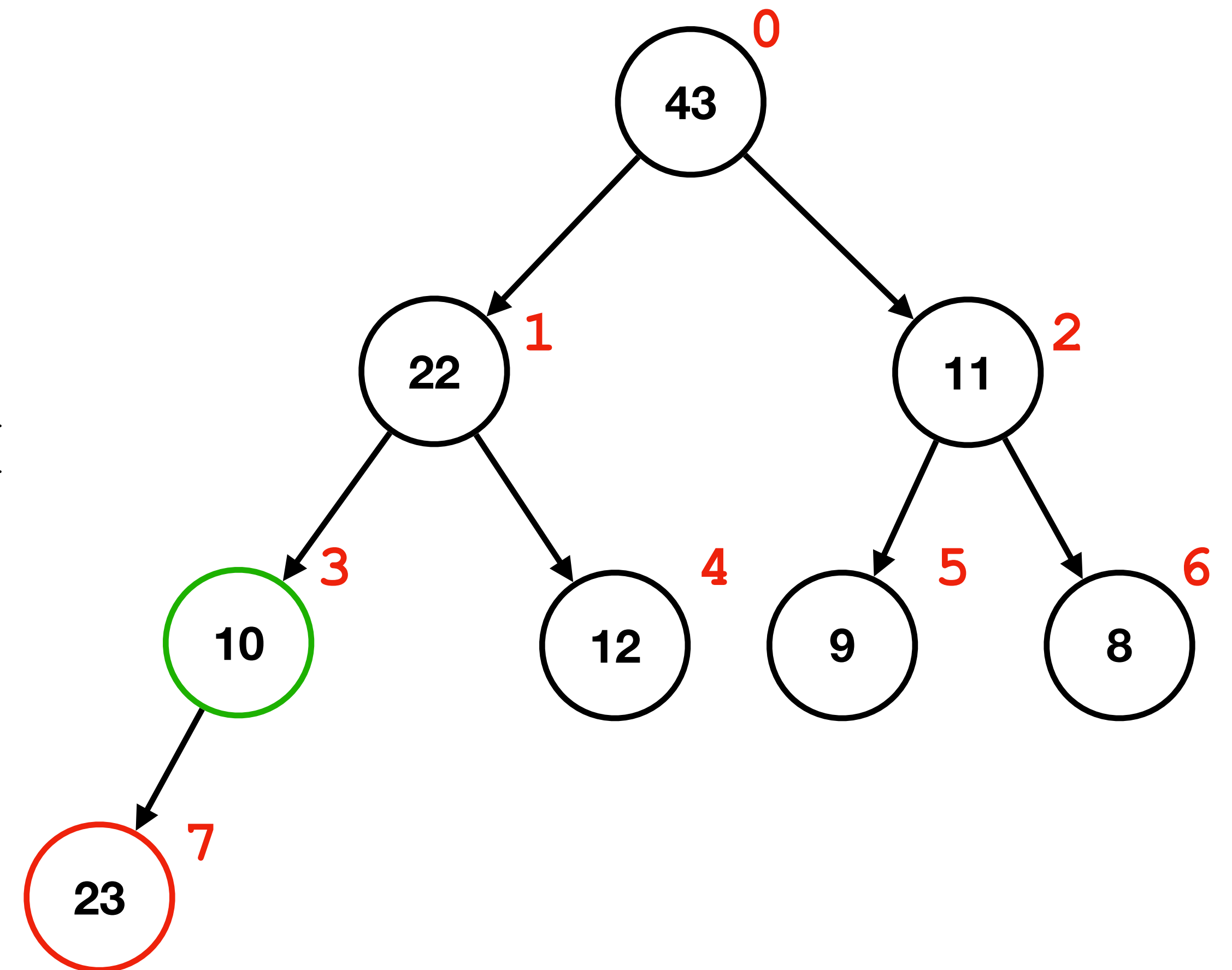


Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        ➡ int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```



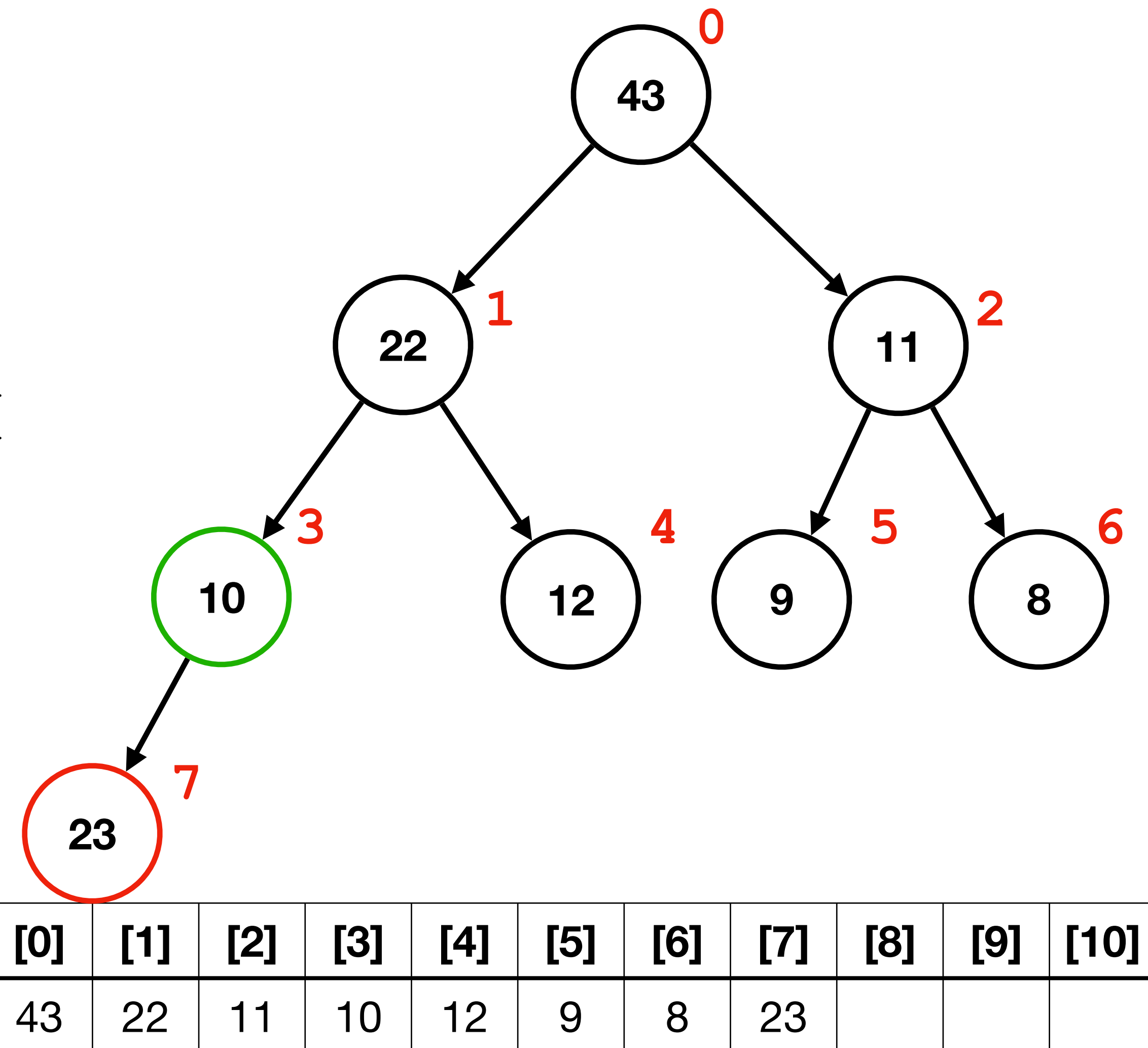
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12	9	8	23			

Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        ➡ if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```



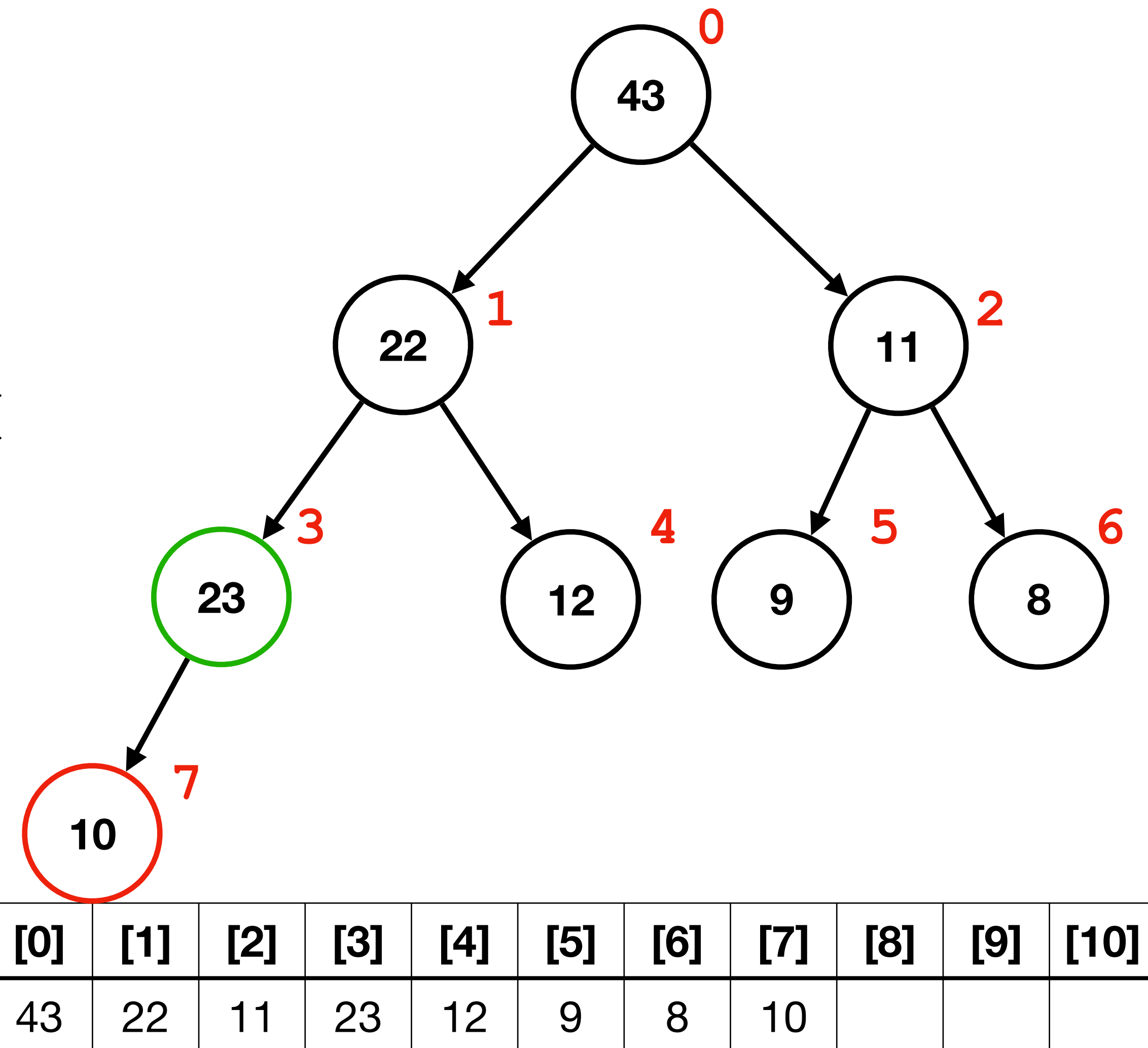
Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

→



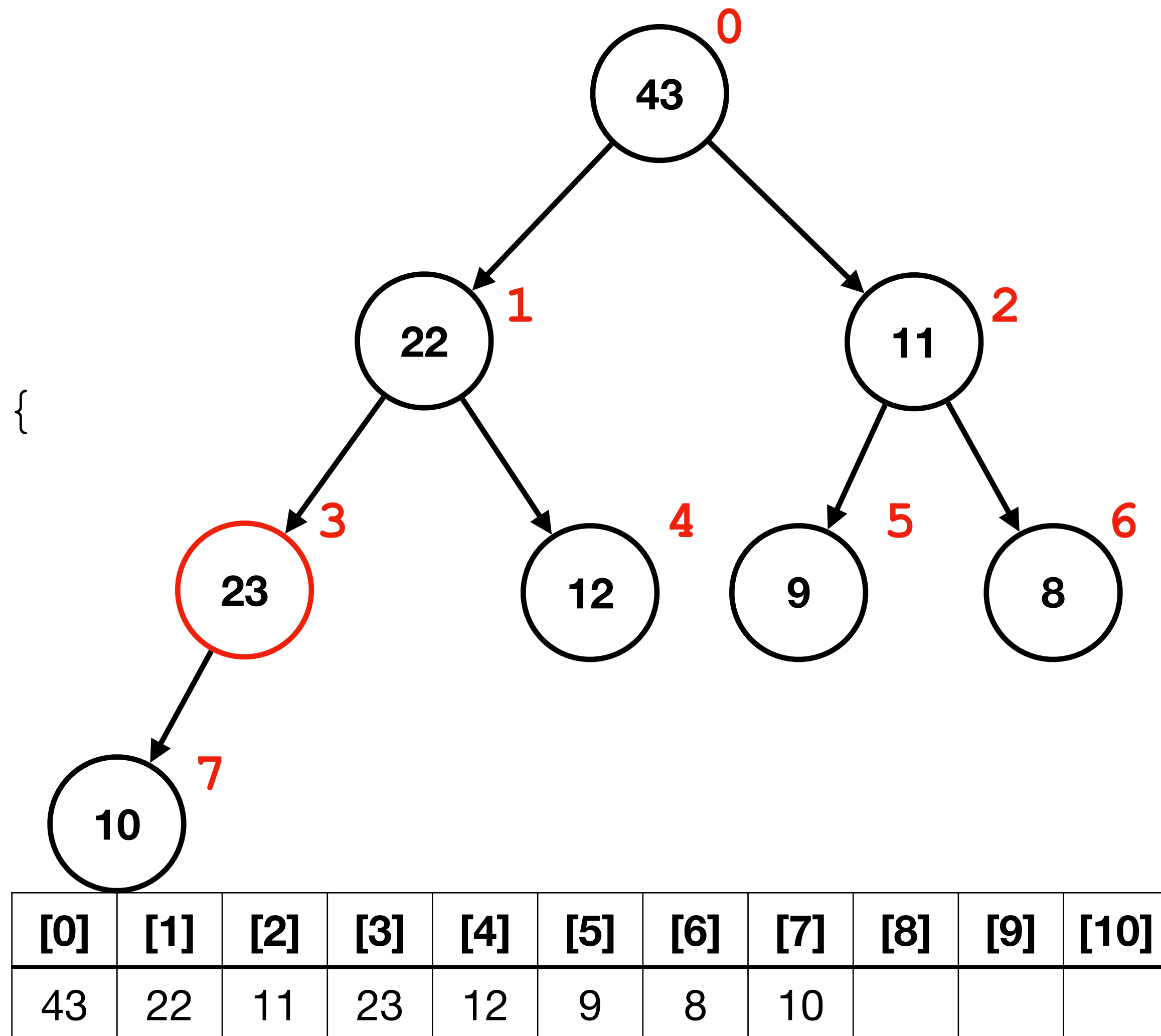
Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

→

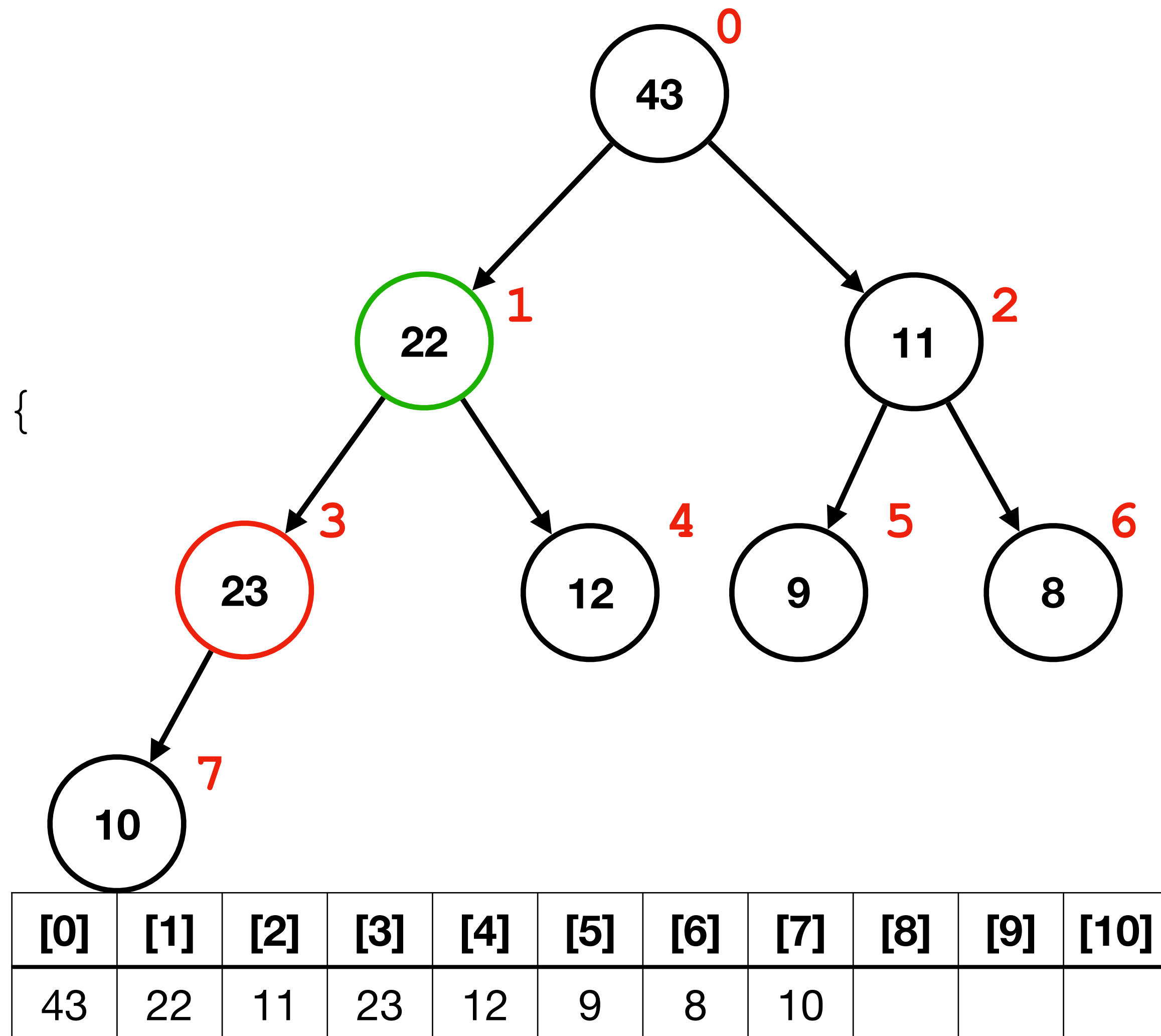


Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        ➡ int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

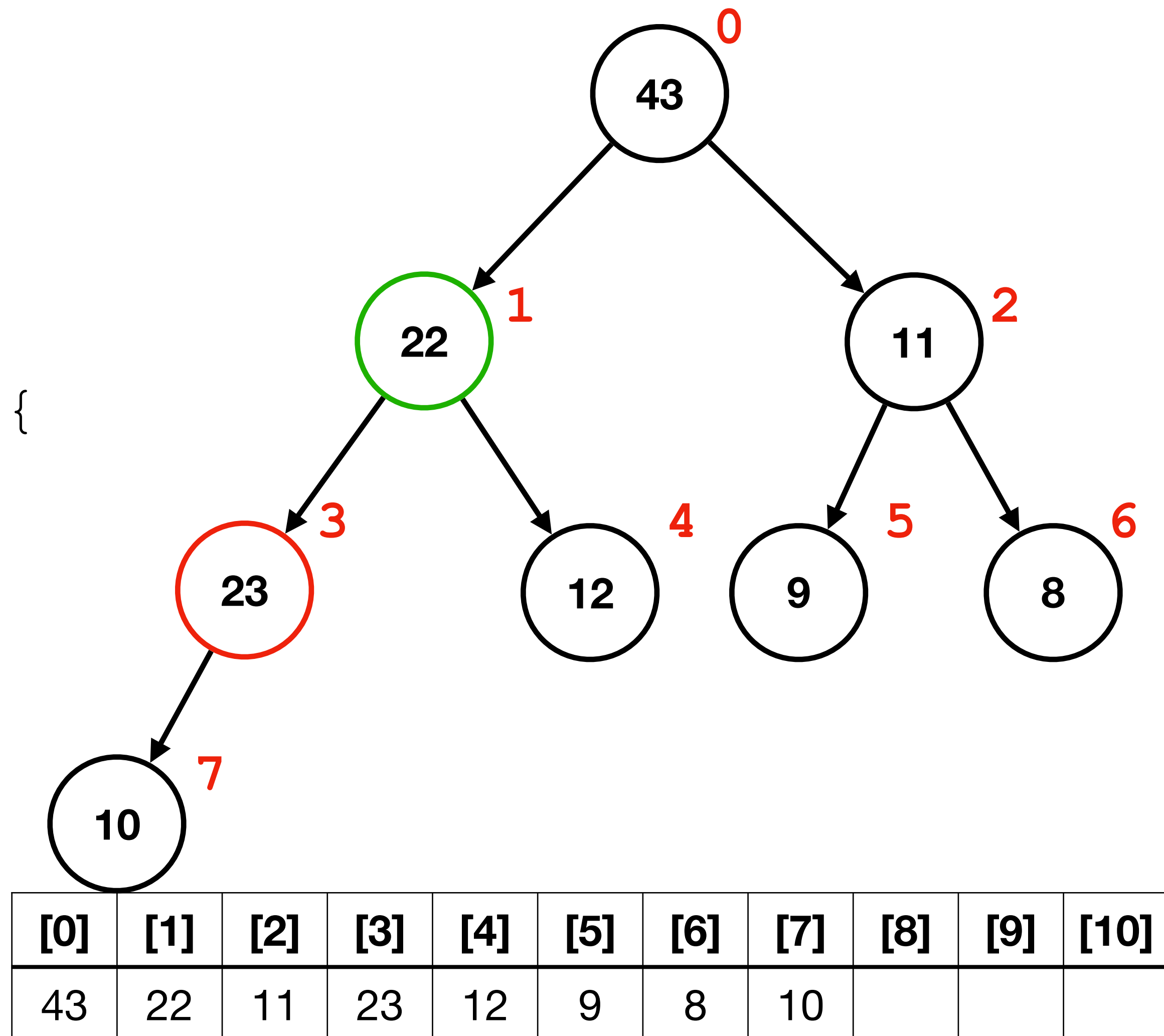


Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        ➔ if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```



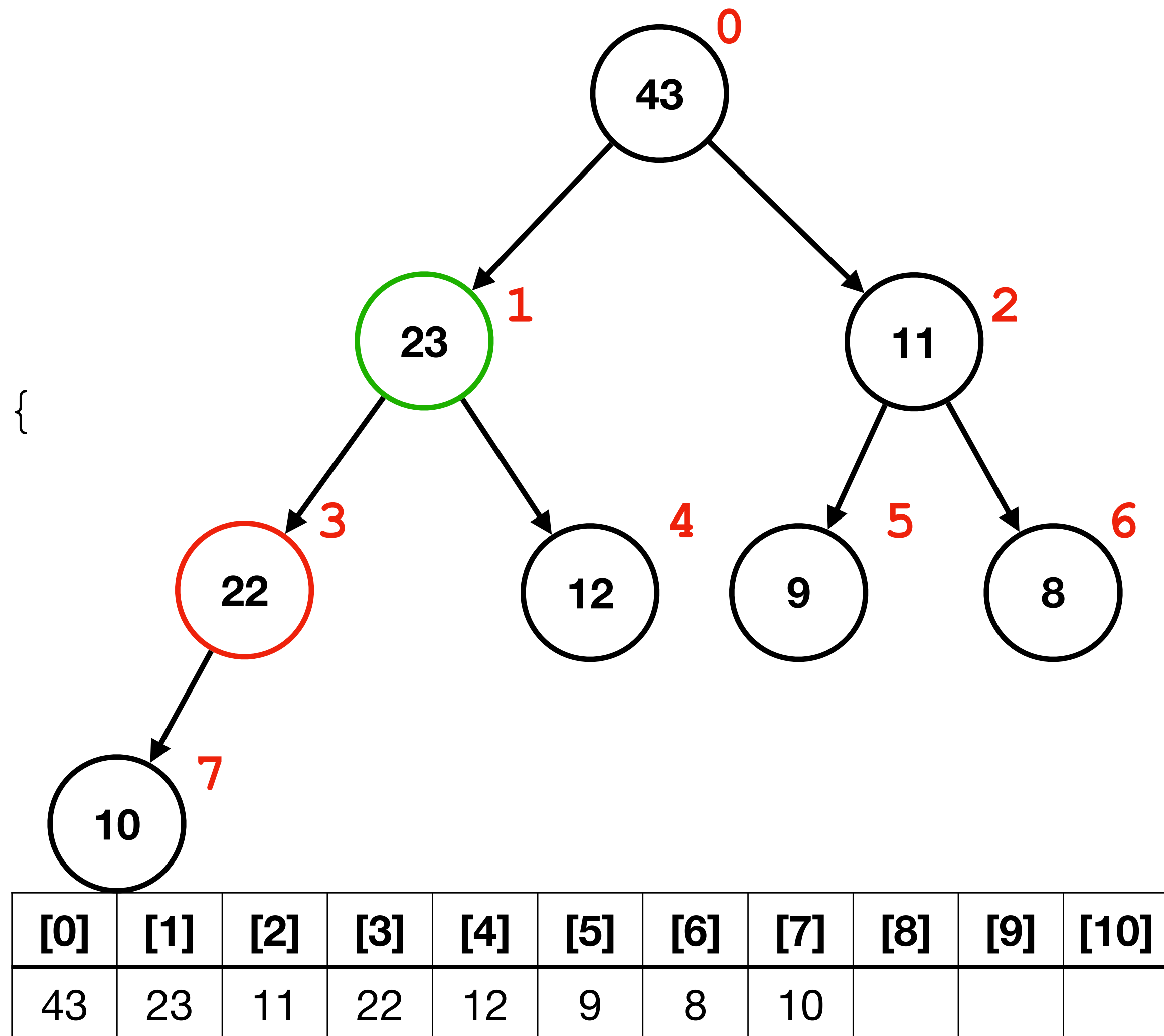
Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

→



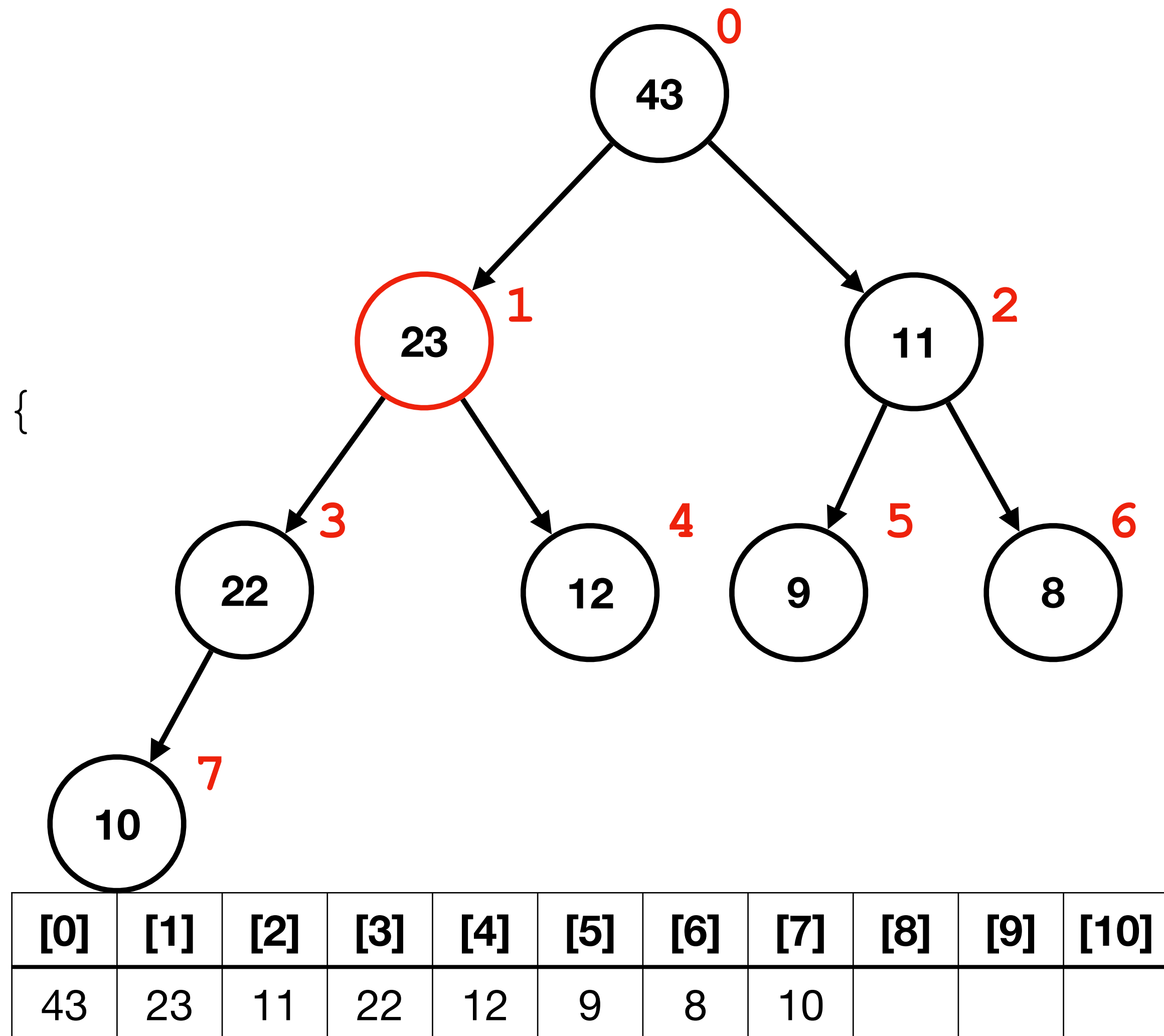
Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

➔

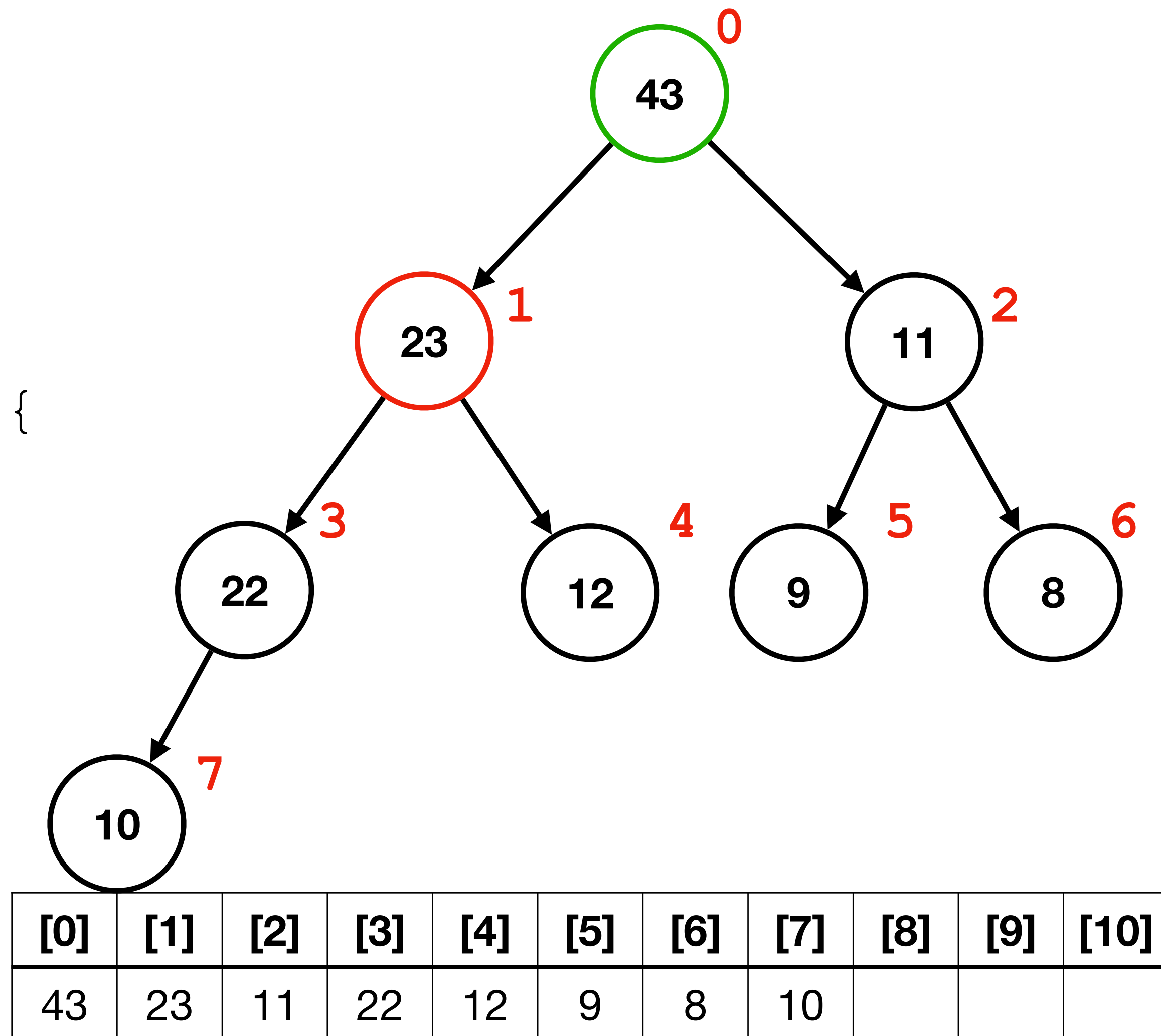


Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        ➡ int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

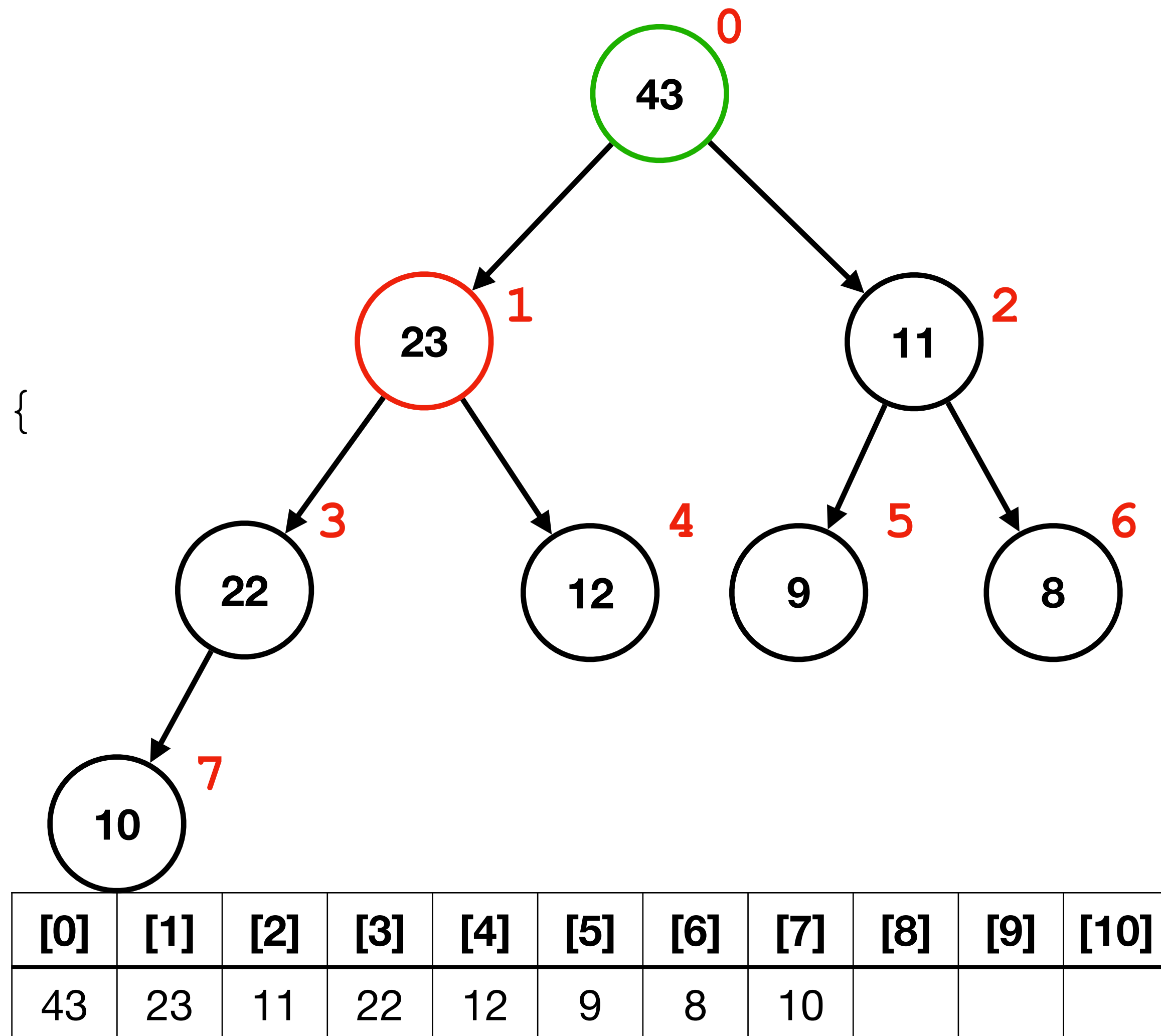



Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```

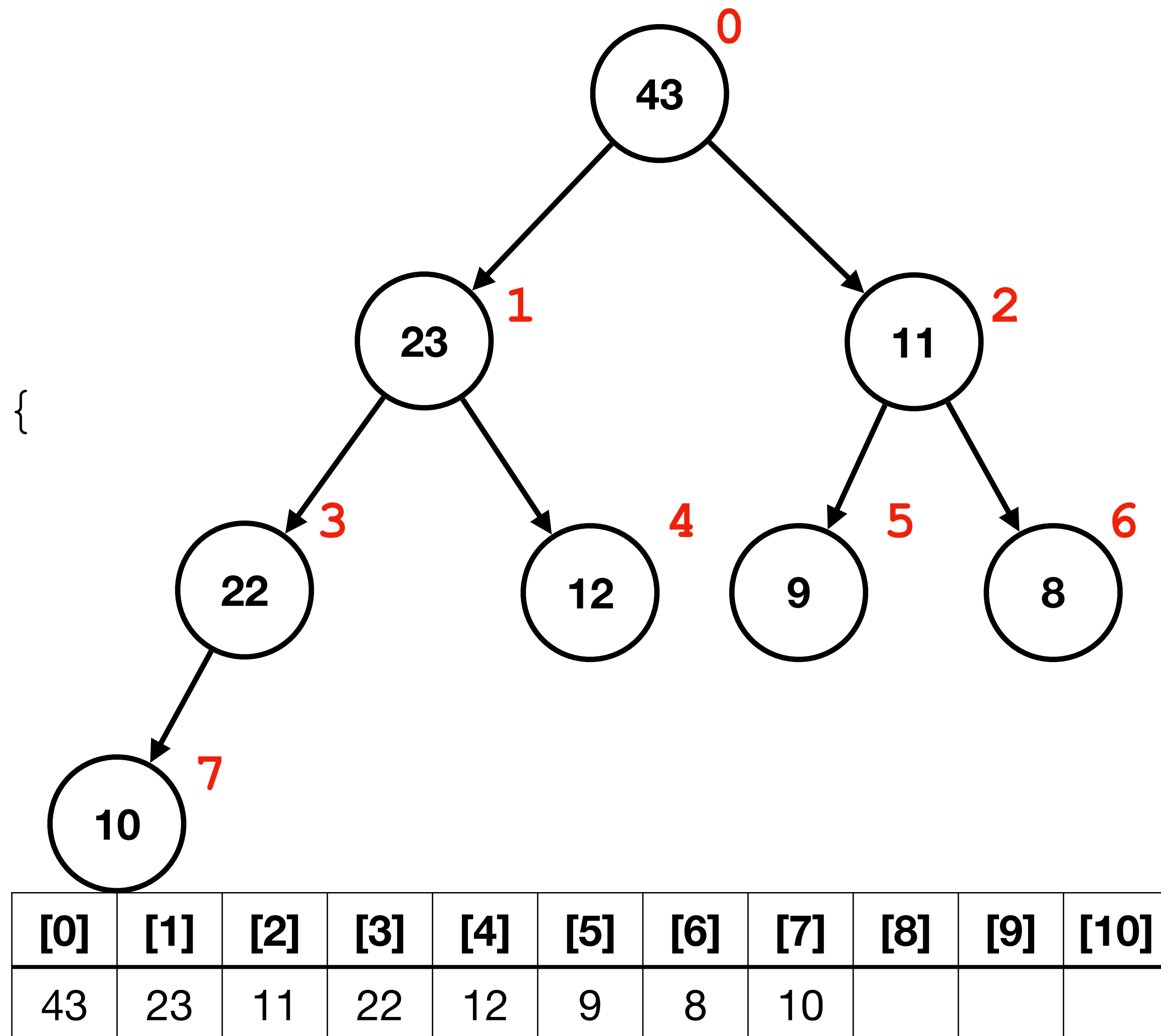


Heap

Bubble Up

```
void bubble_up(int *arr, int i)
{
    while (i > 0) {
        int parent = (i - 1) / 2;

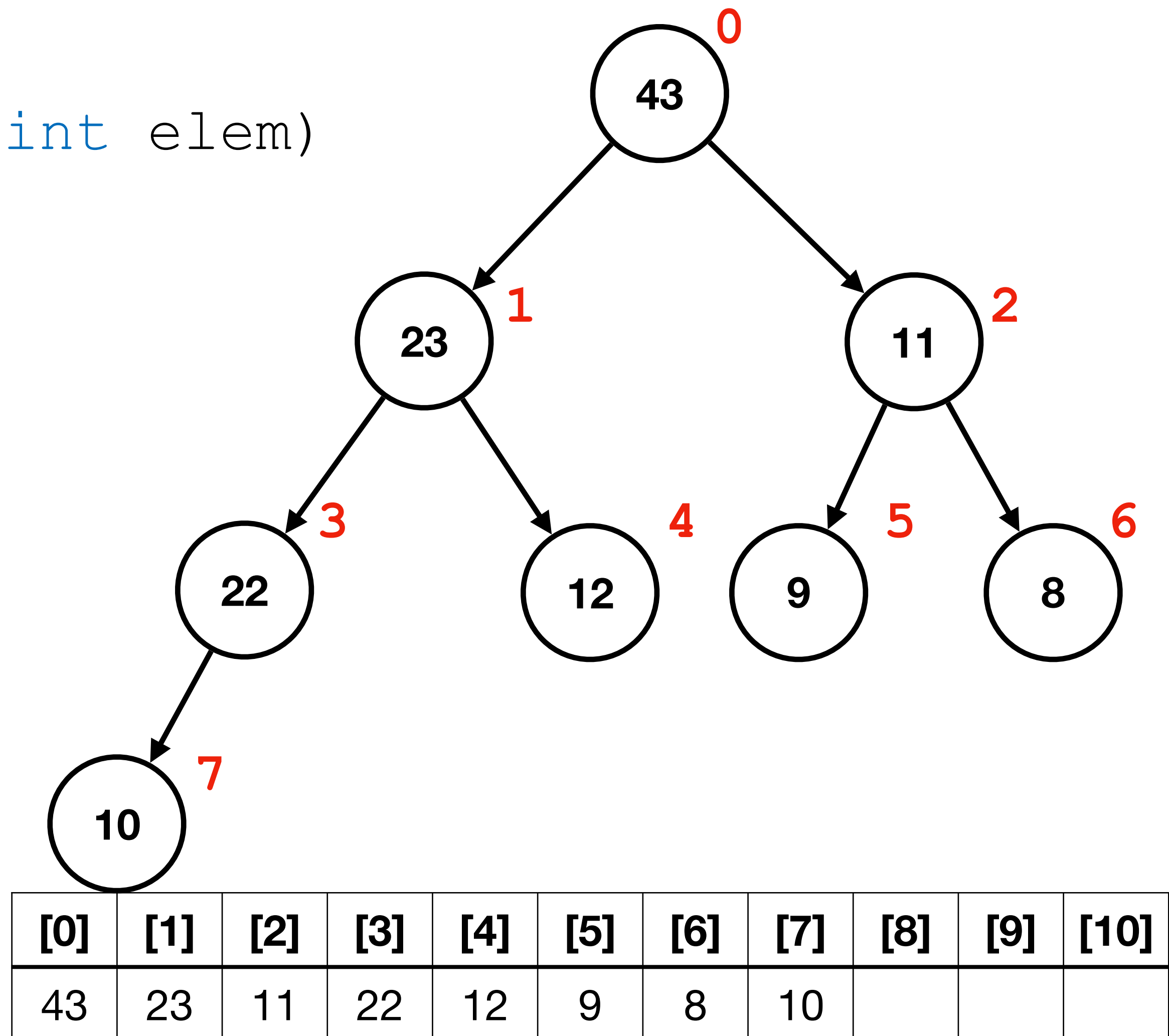
        if (arr[parent] < arr[i]) {
            swap(arr, parent, i);
            i = parent;
        } else {
            break;
        }
    }
}
```



Heap

Bubble Up

```
void push_heap(int *arr, int len, int elem)
{
    arr[len] = elem;
    bubble_up(arr, len);
}
```

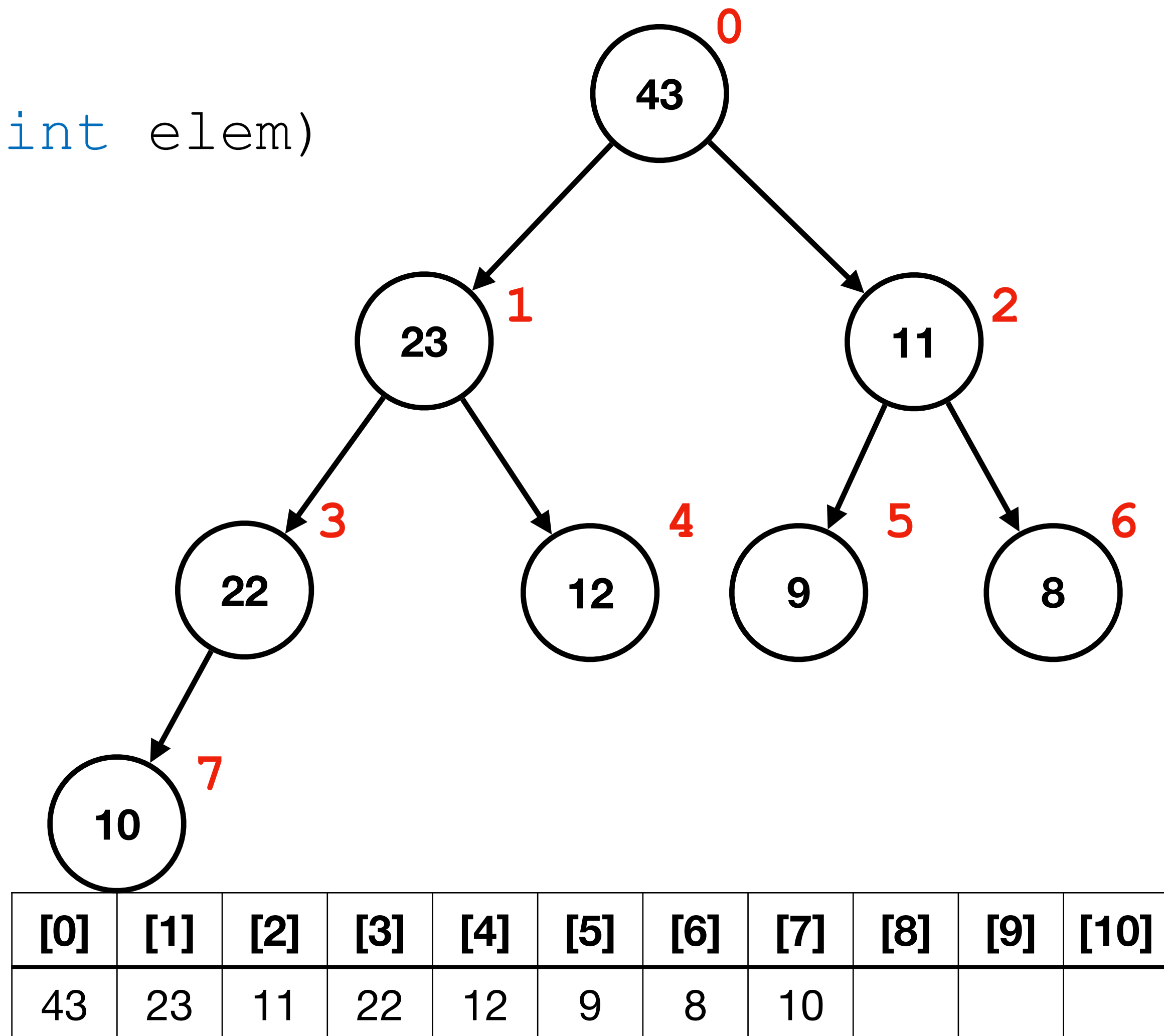


Heap

Bubble Up

```
void push_heap(int *arr, int len, int elem)
{
    arr[len] = elem;
    bubble_up(arr, len);
}
```

```
push_heap(arr, 8, 40);
```

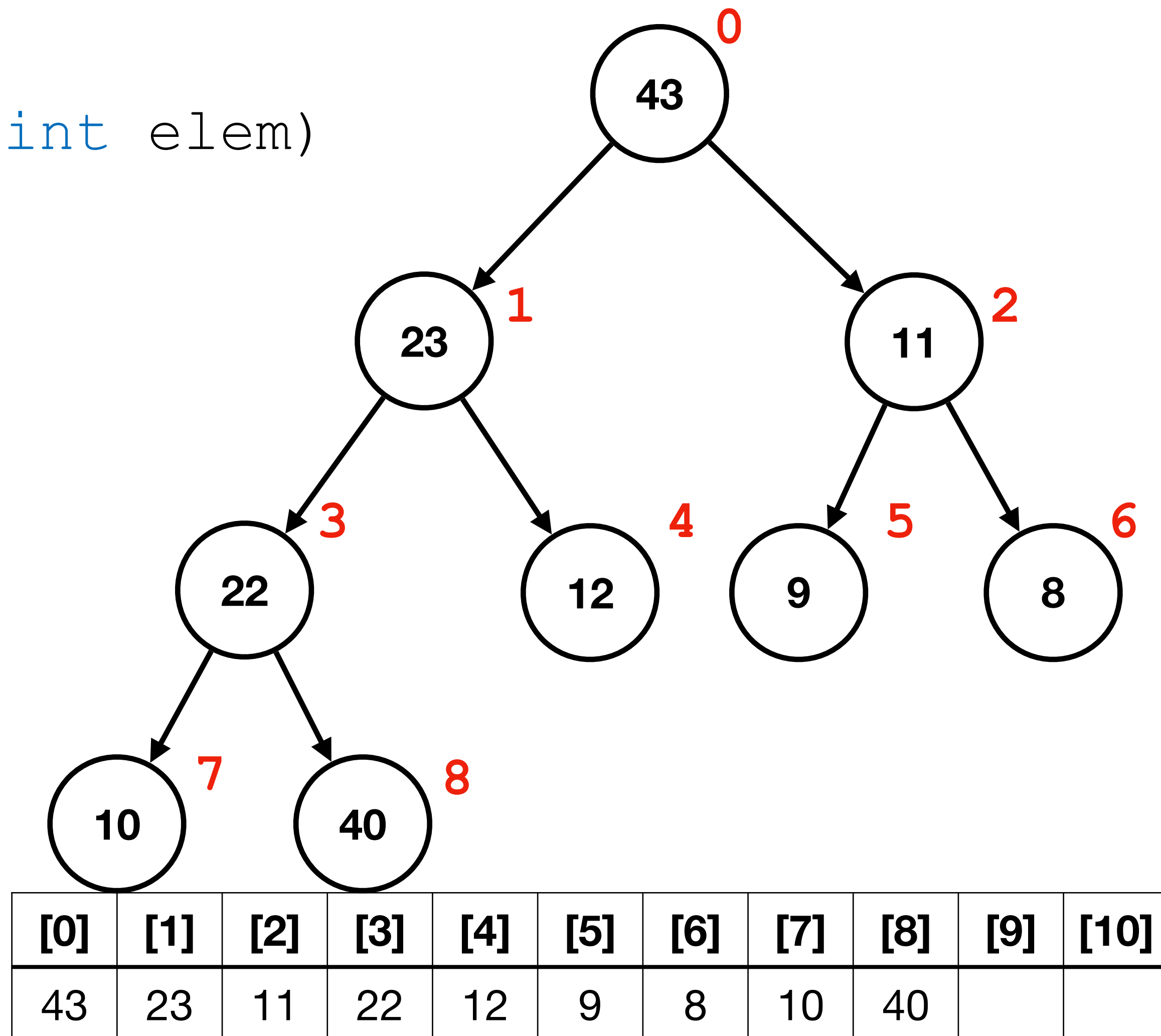


Heap

Bubble Up

```
void push_heap(int *arr, int len, int elem)
{
    arr[len] = elem;
    bubble_up(arr, len);
}
```

```
push_heap(arr, 8, 40);
```

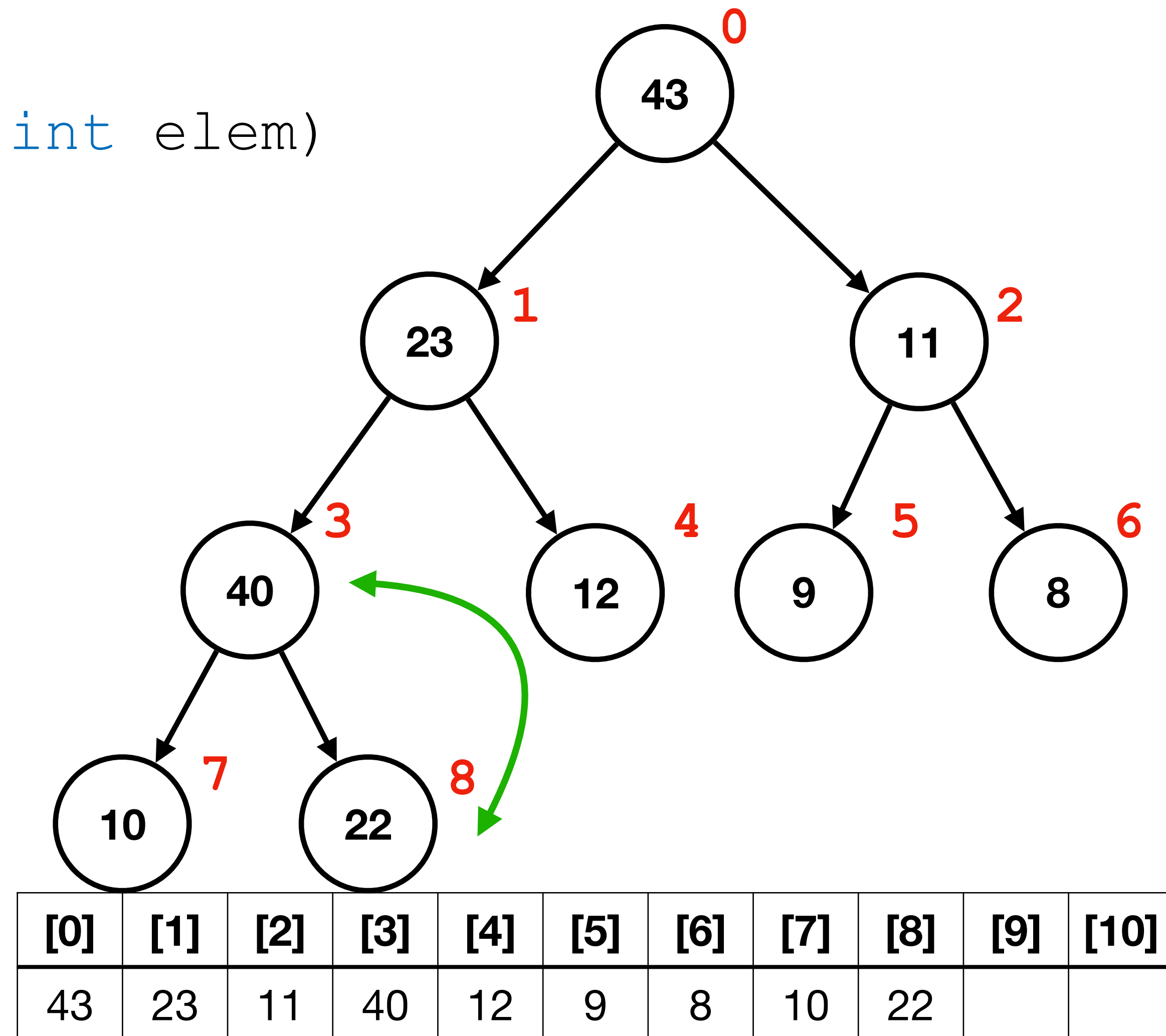


Heap

Bubble Up

```
void push_heap(int *arr, int len, int elem)
{
    arr[len] = elem;
    bubble_up(arr, len);
}
```

```
push_heap(arr, 8, 40);
```

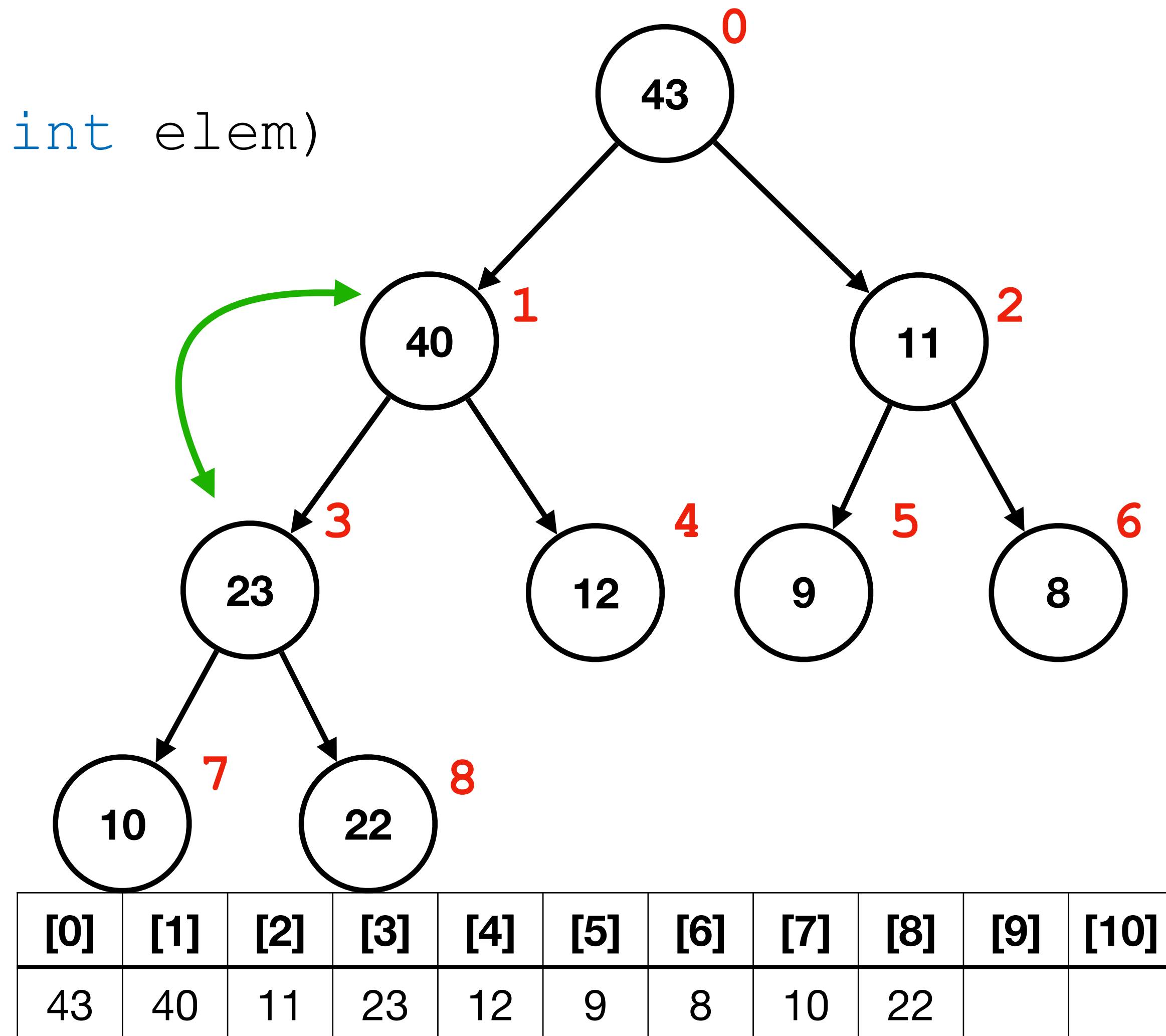


Heap

Bubble Up

```
void push_heap(int *arr, int len, int elem)
{
    arr[len] = elem;
    bubble_up(arr, len);
}
```

```
push_heap(arr, 8, 40);
```



Heap

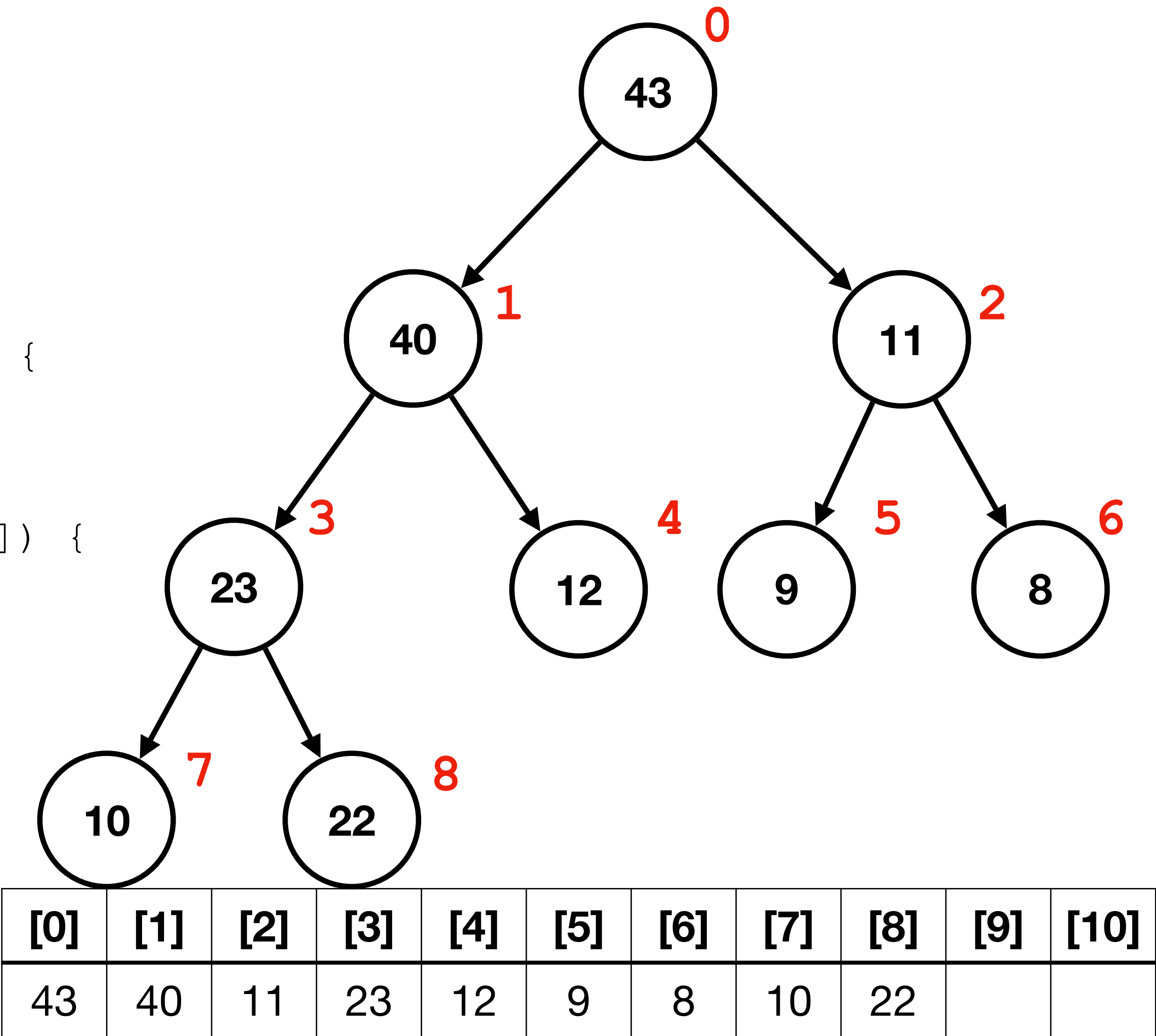
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

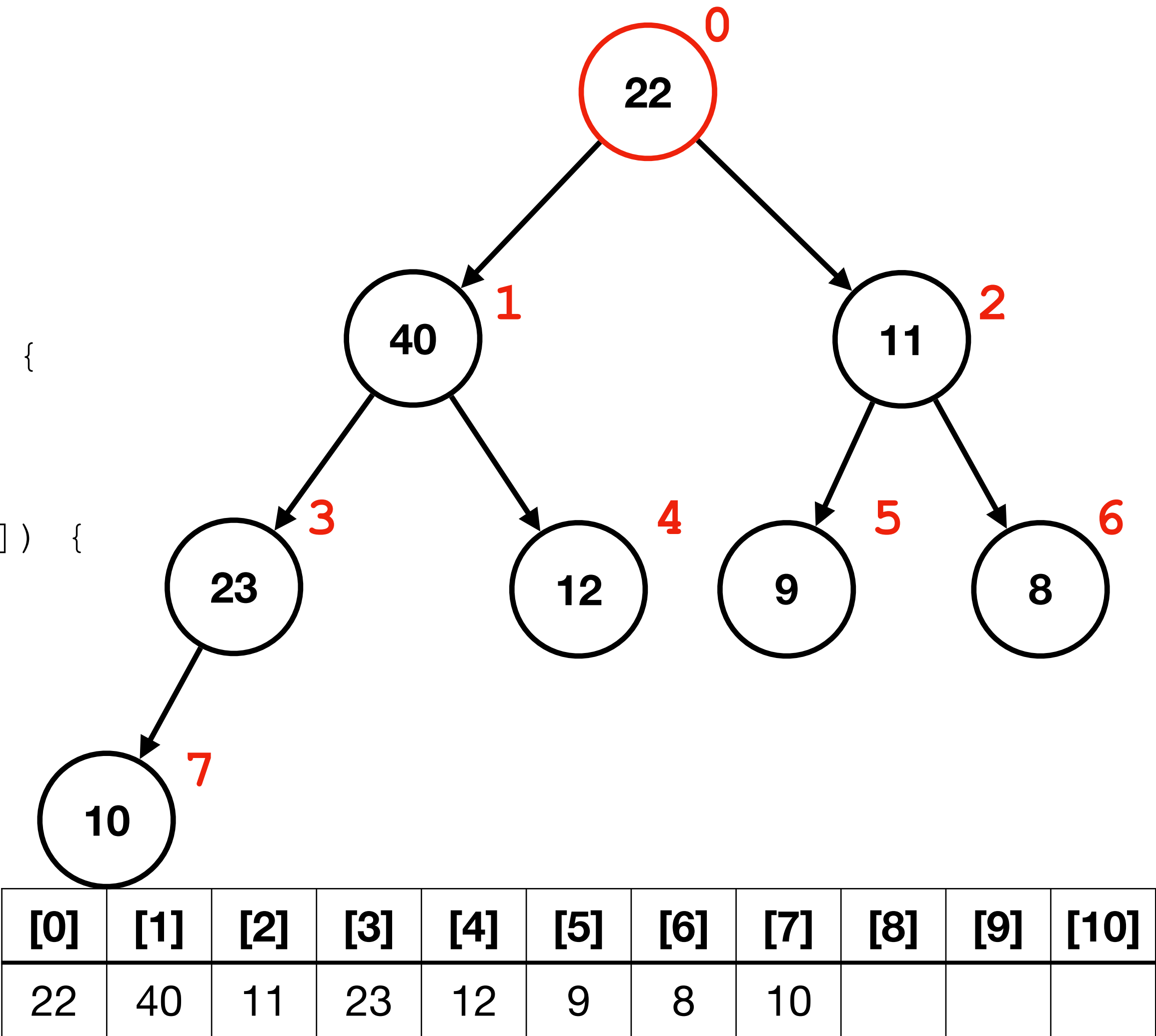
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

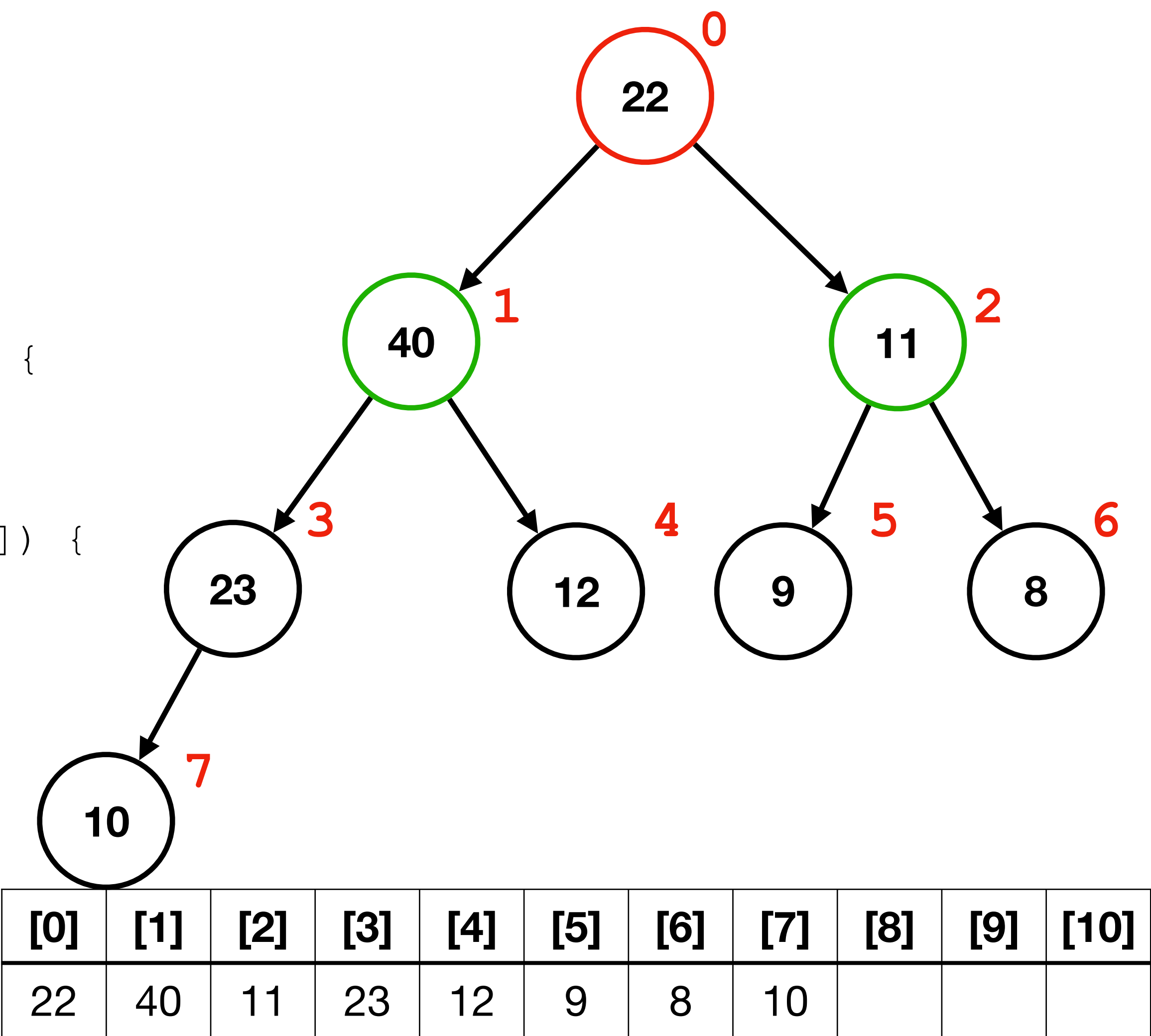
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

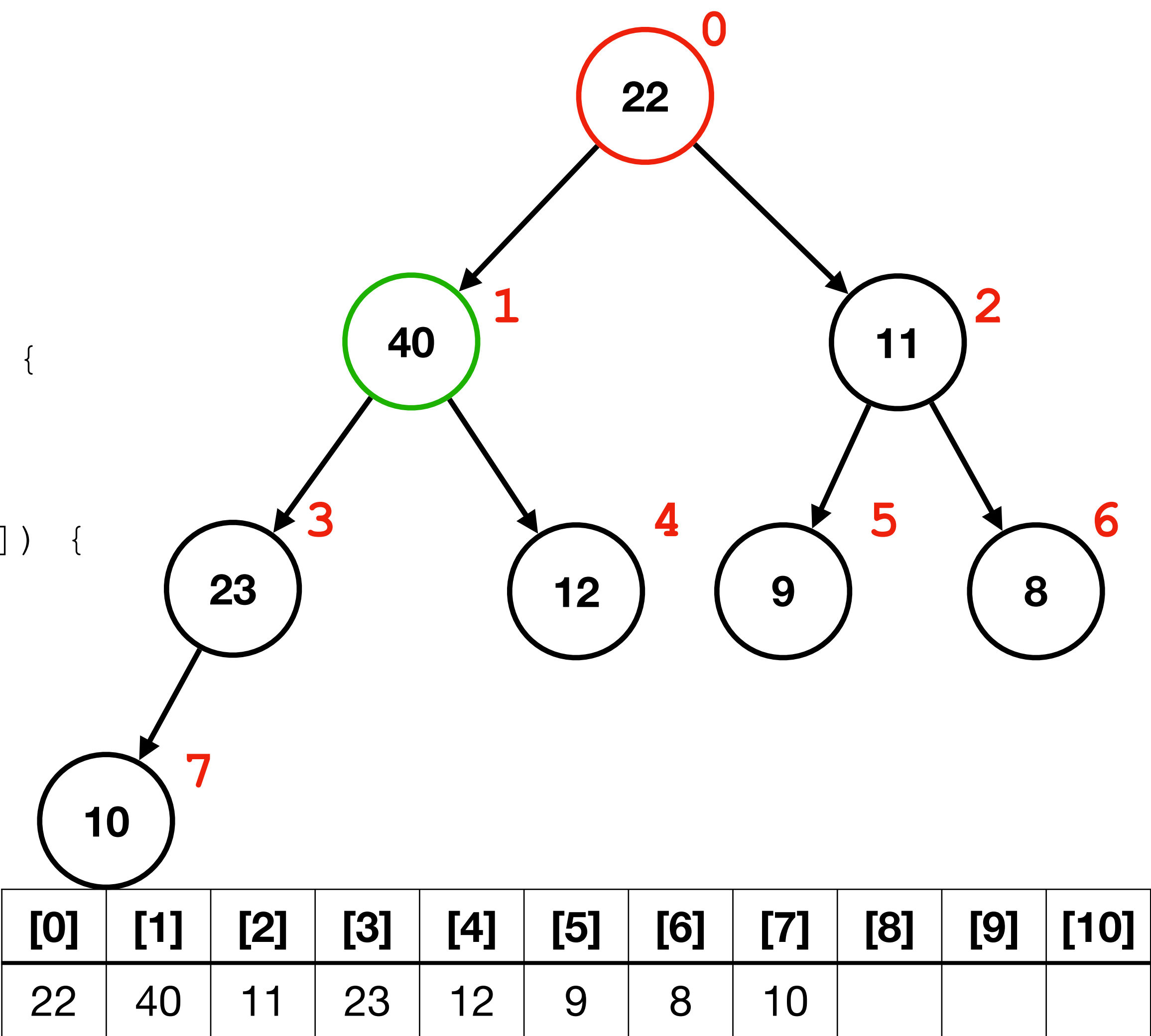
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

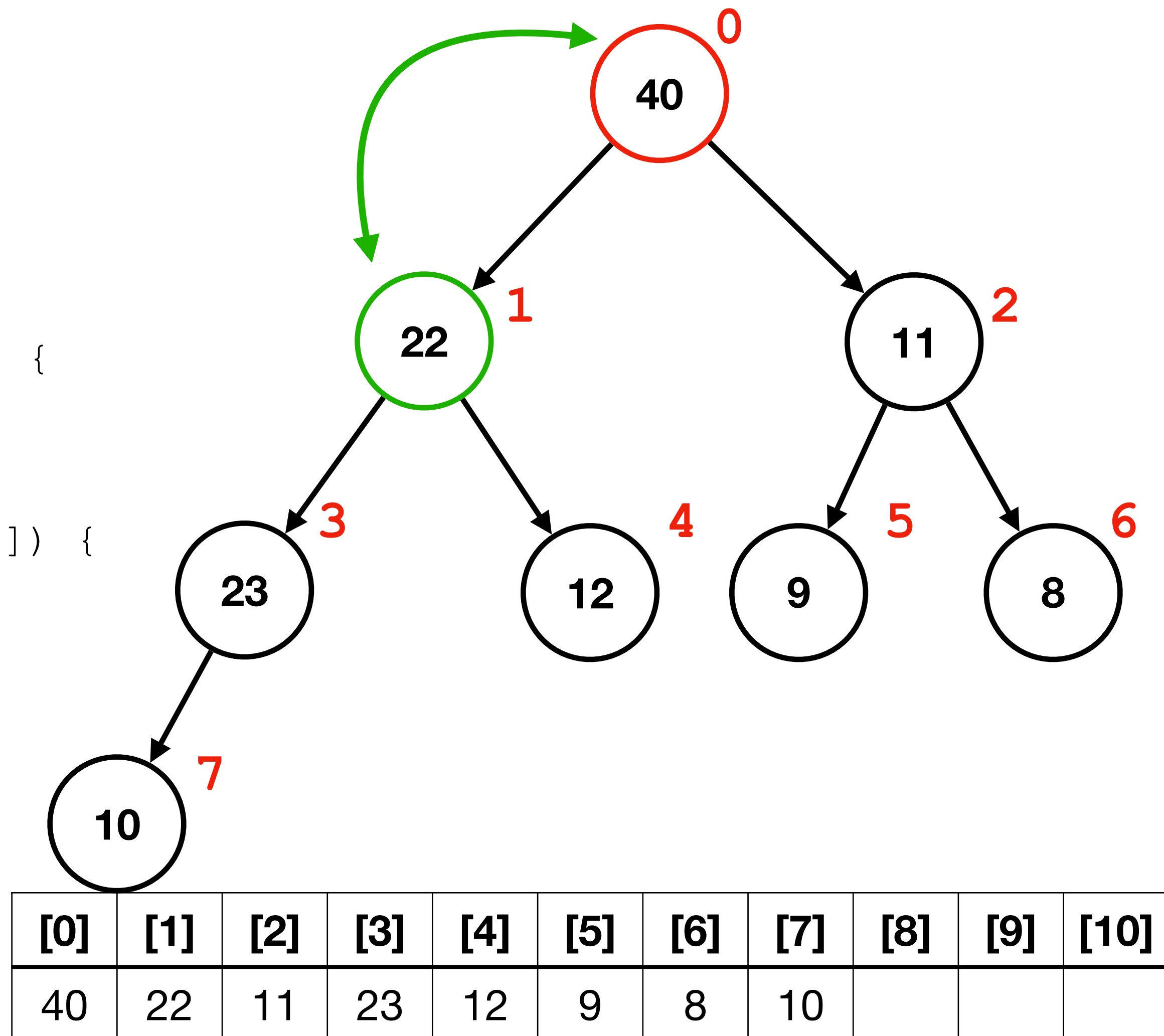
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

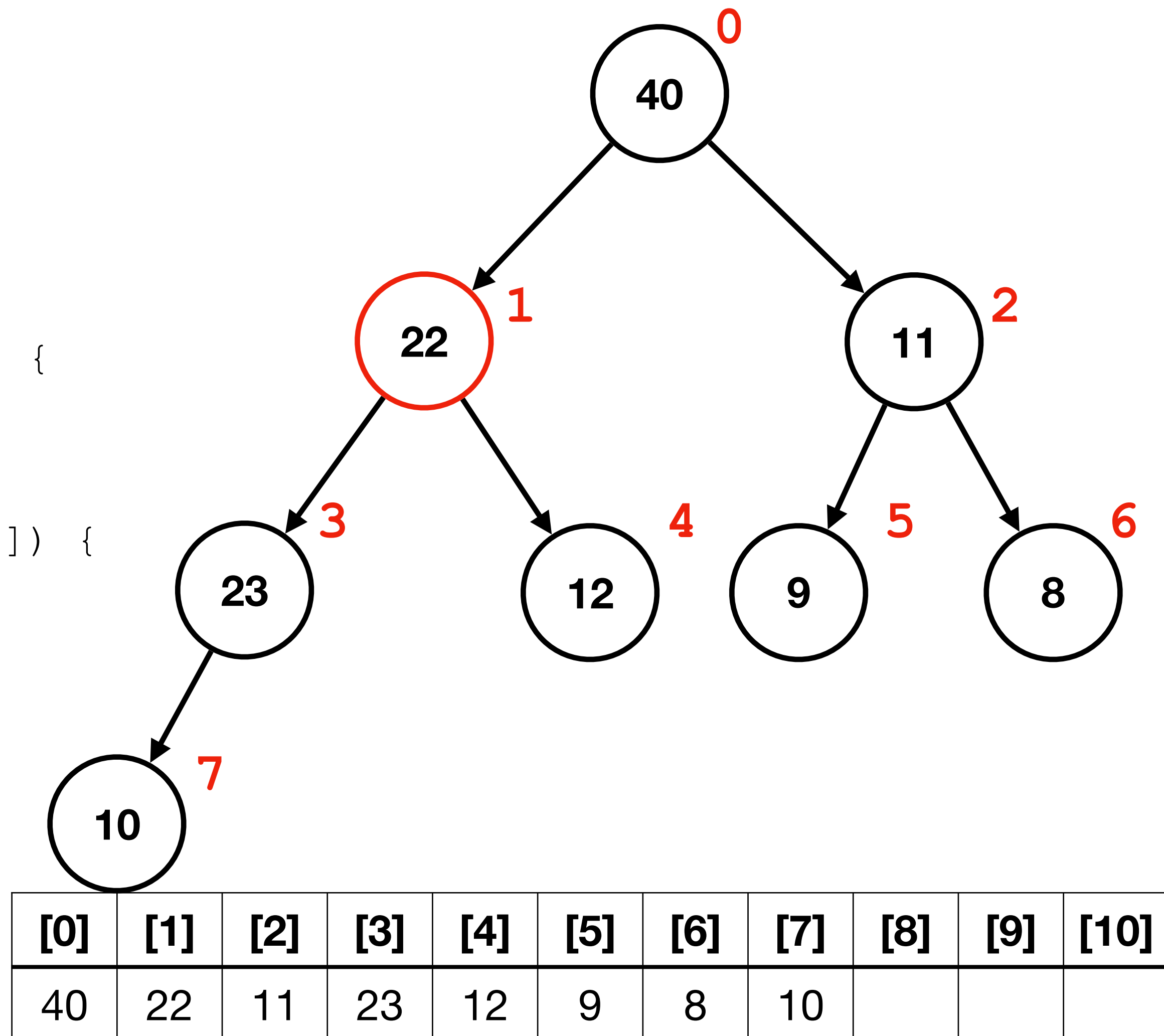
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

Bubble Down

```
void bubble_down(int *arr, int len, int i)
```

```
{
```

```
    for (;;) {
```

```
        int largest = i;
```

```
        int left = 2 * i + 1;
```

```
        int right = 2 * i + 2;
```



```
        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }
```

```
        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }
```

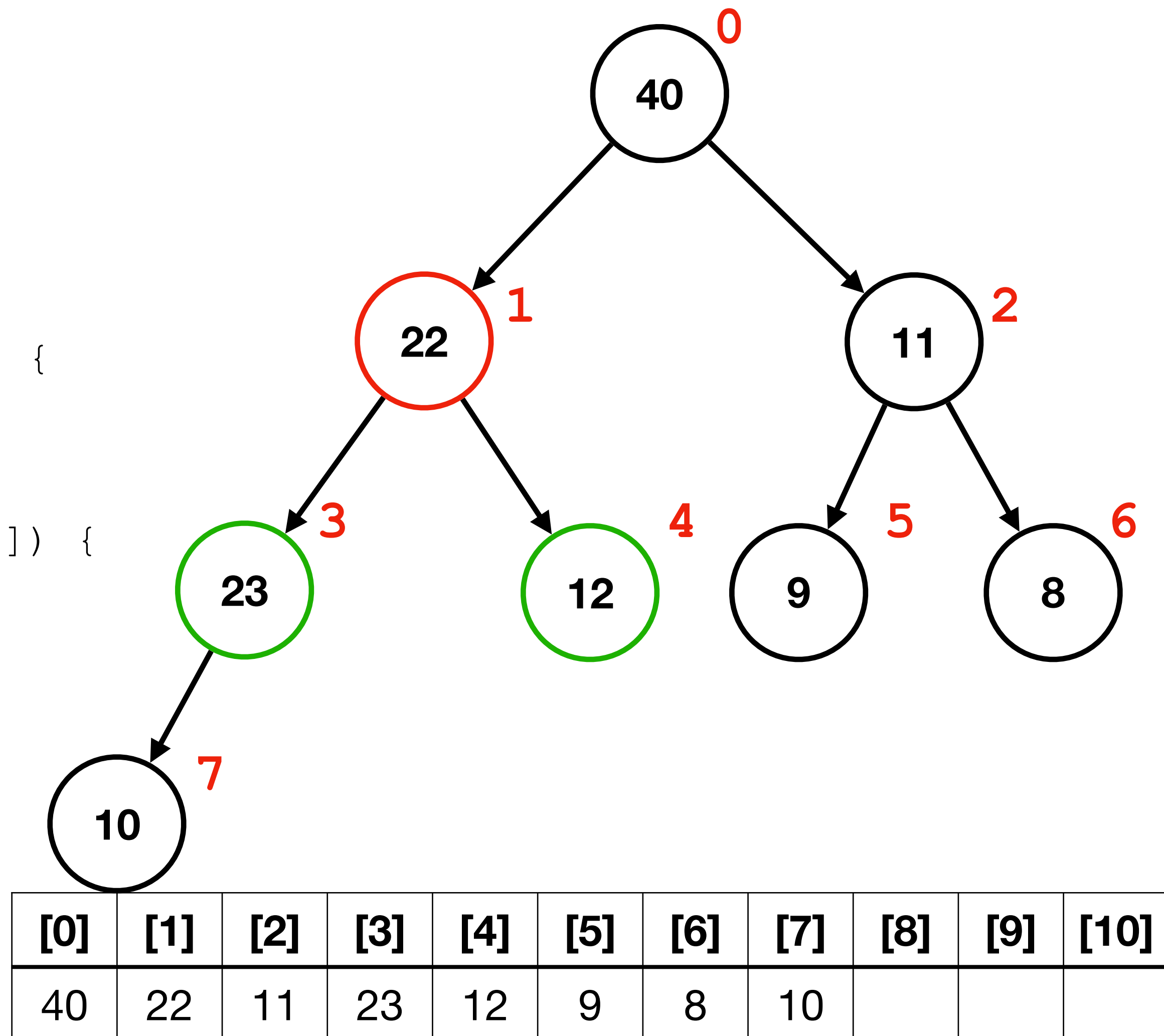
```
        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```



Heap

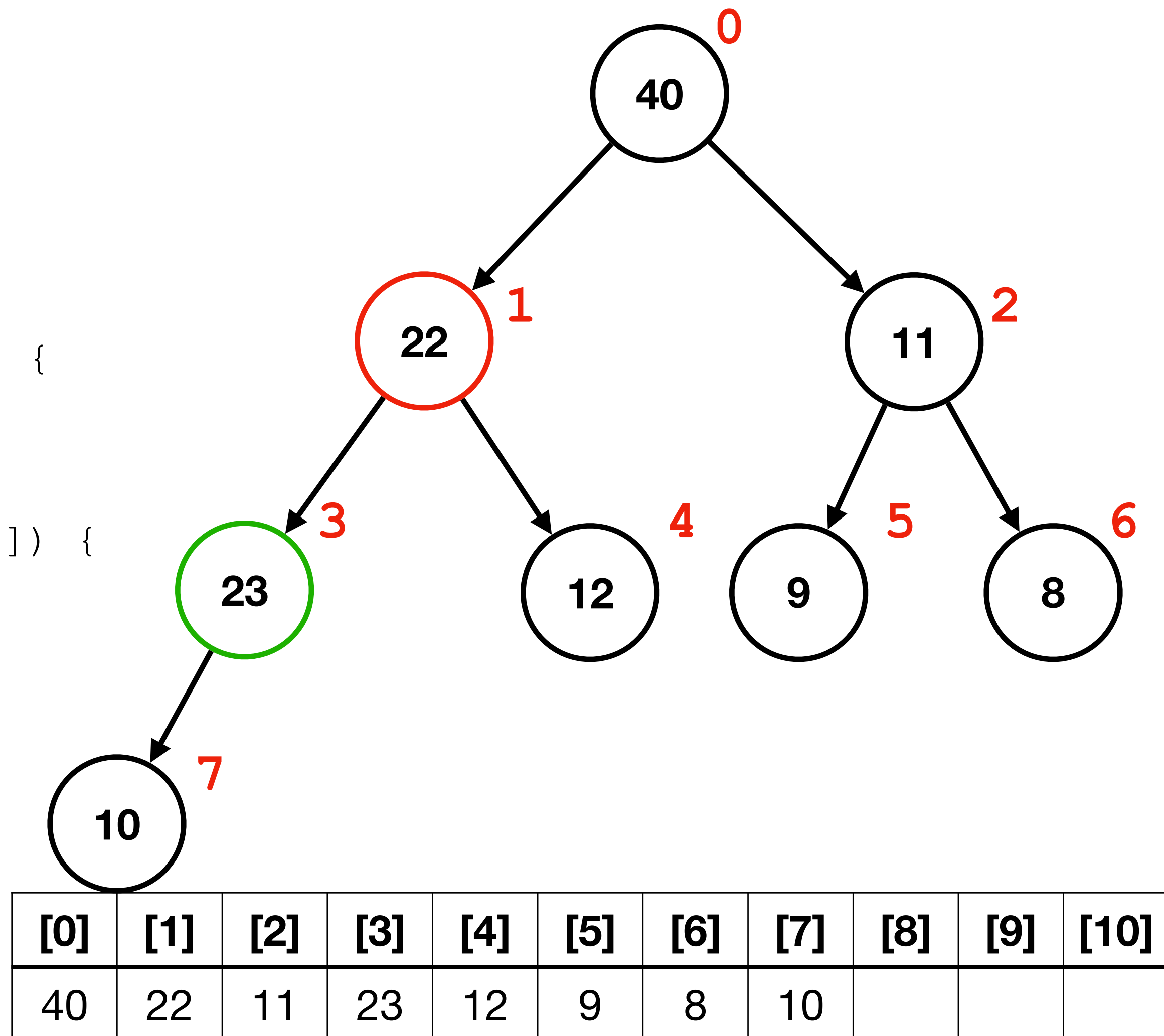
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        → if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

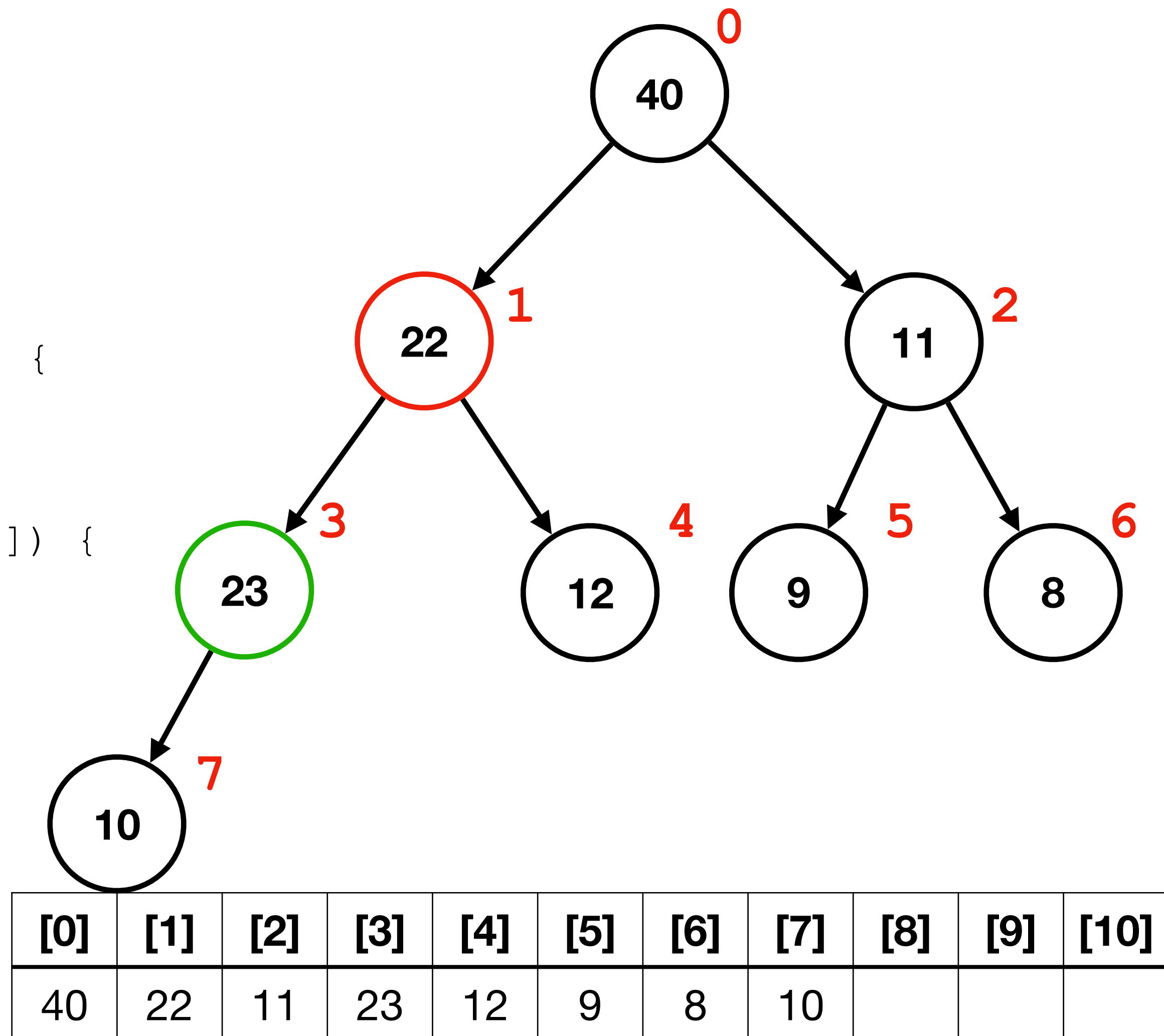
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

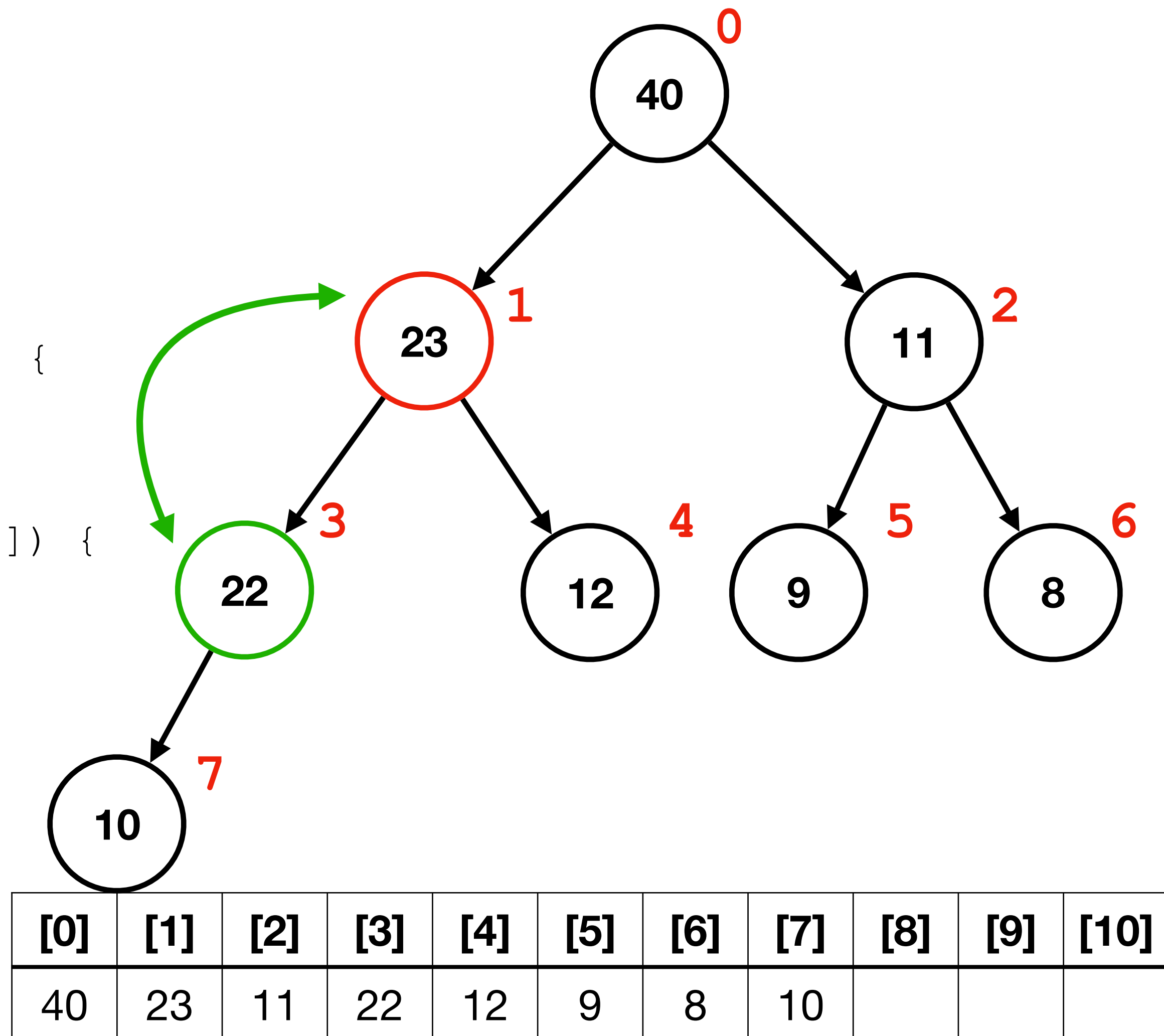
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

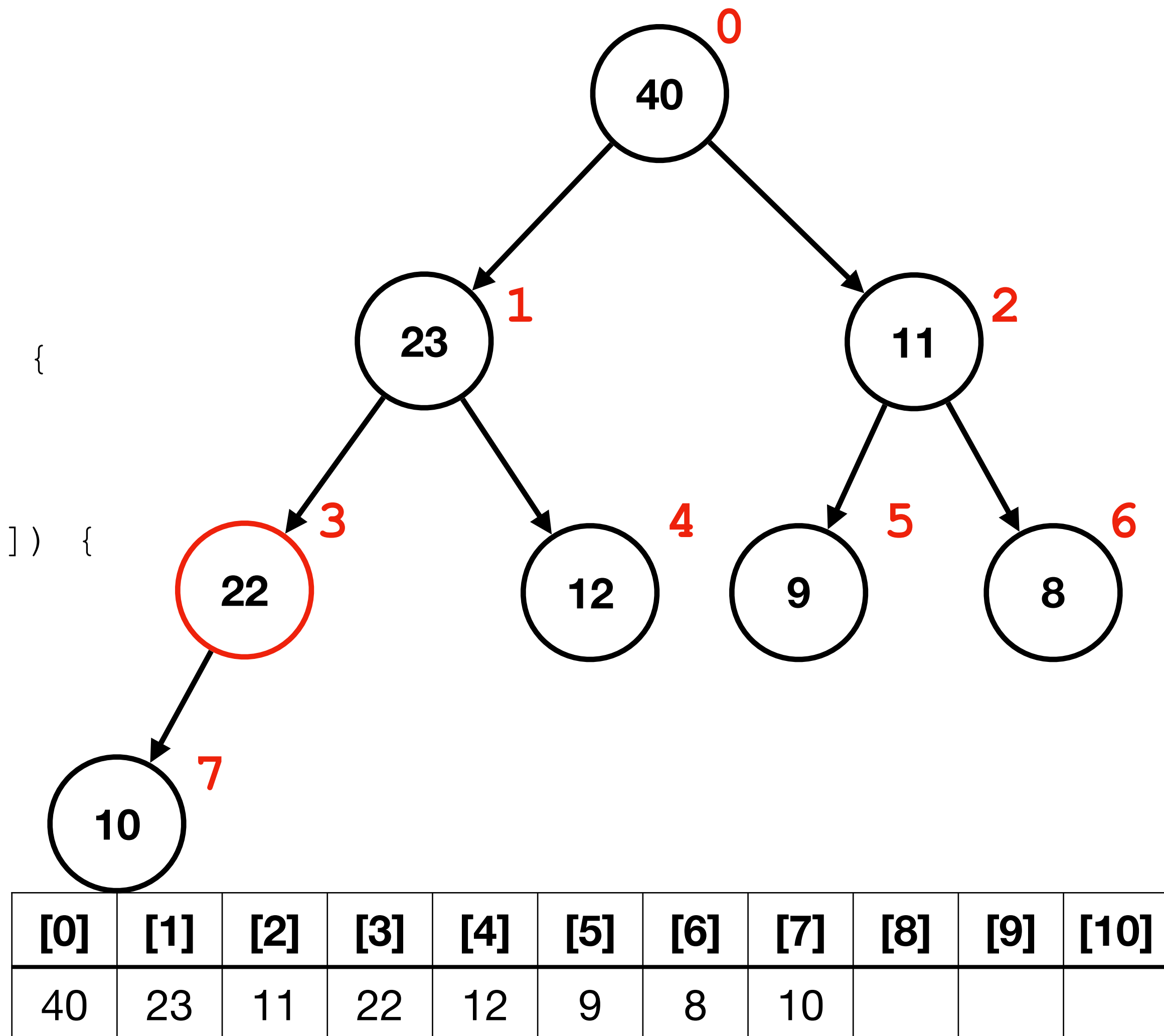
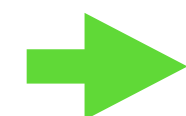
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

Bubble Down

```
void bubble_down(int *arr, int len, int i)
```

```
{
```

```
    for (;;) {
```

```
        int largest = i;
```

```
        int left = 2 * i + 1;
```

```
        int right = 2 * i + 2;
```



```
        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }
```

```
        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }
```

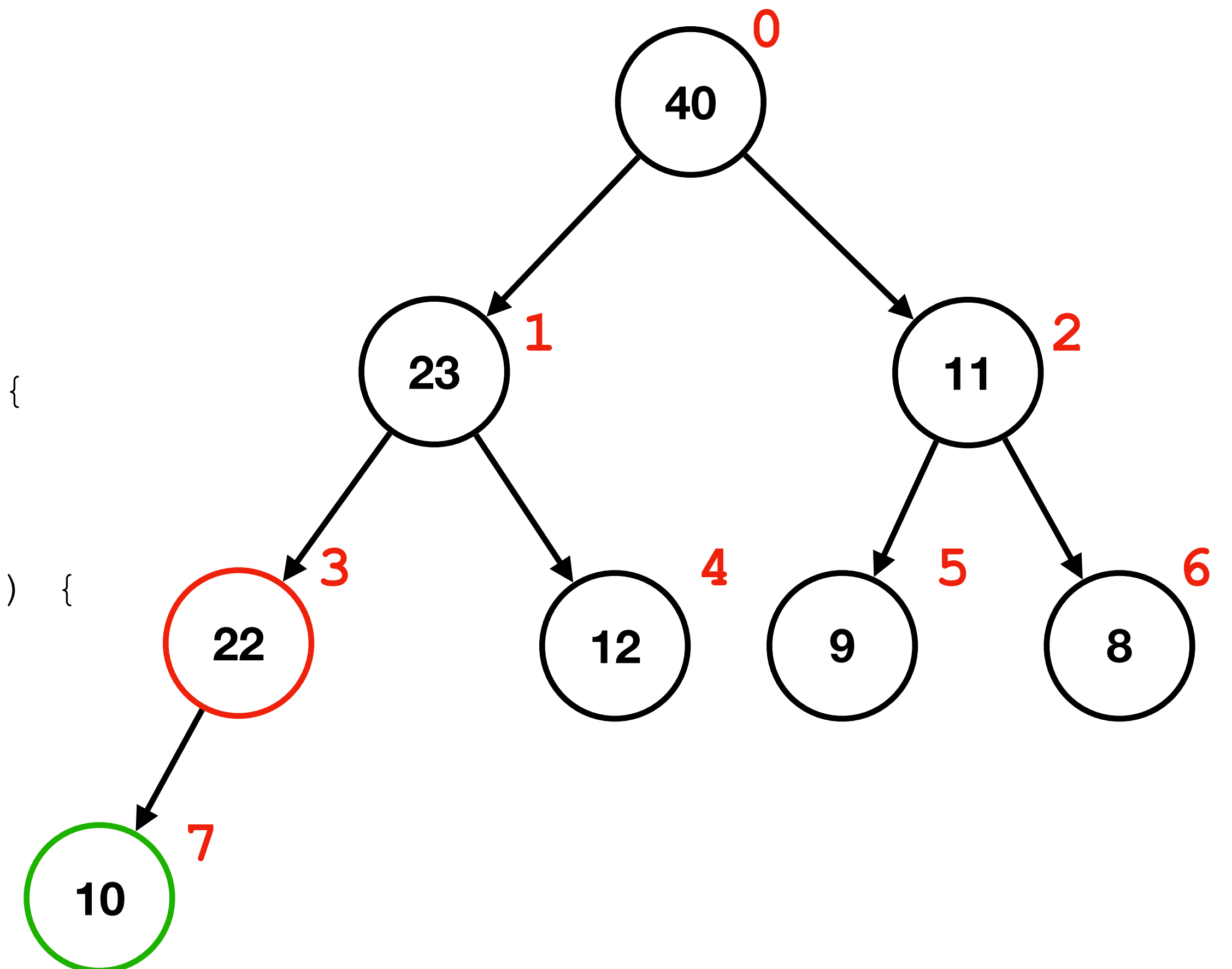
```
        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
40	23	11	22	12	9	8	10			

Heap

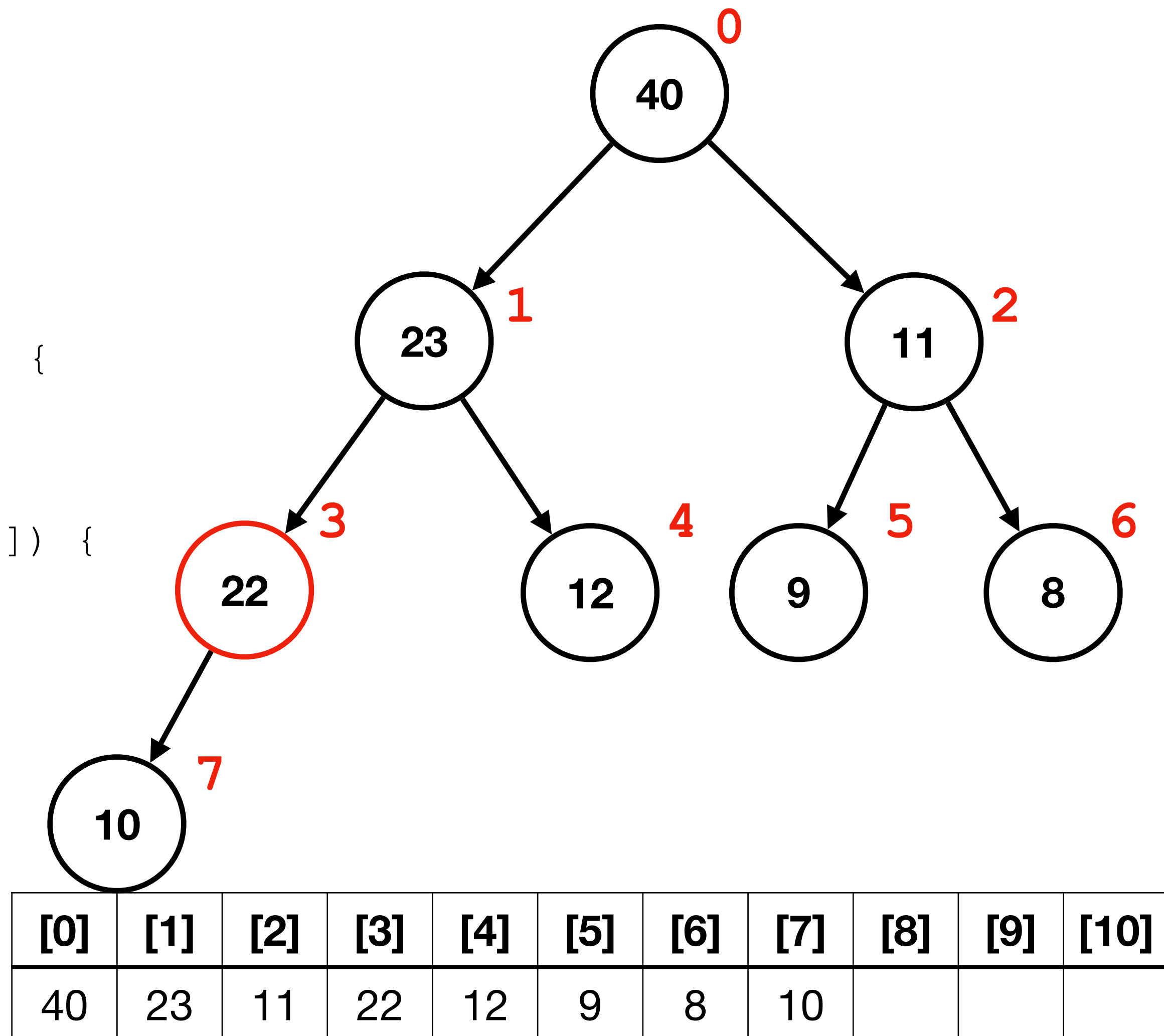
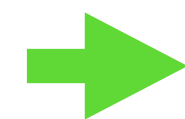
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```



Heap

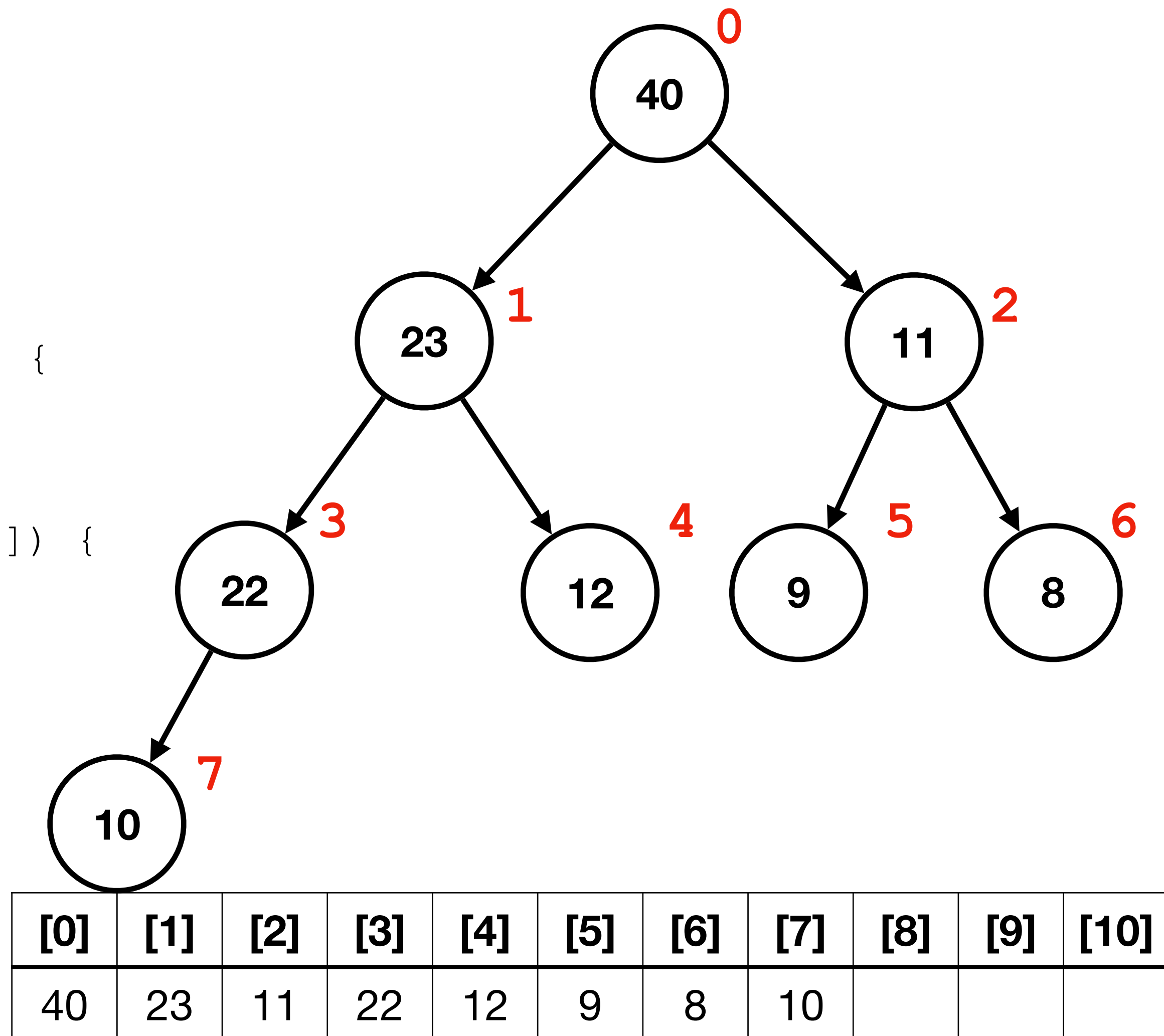
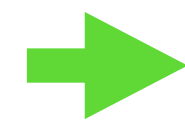
Bubble Down

```
void bubble_down(int *arr, int len, int i)
{
    for (;;) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        if (left < len && arr[left] > arr[largest]) {
            largest = left;
        }

        if (right < len && arr[right] > arr[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(arr, i, largest);
            i = largest;
        } else {
            break;
        }
    }
}
```

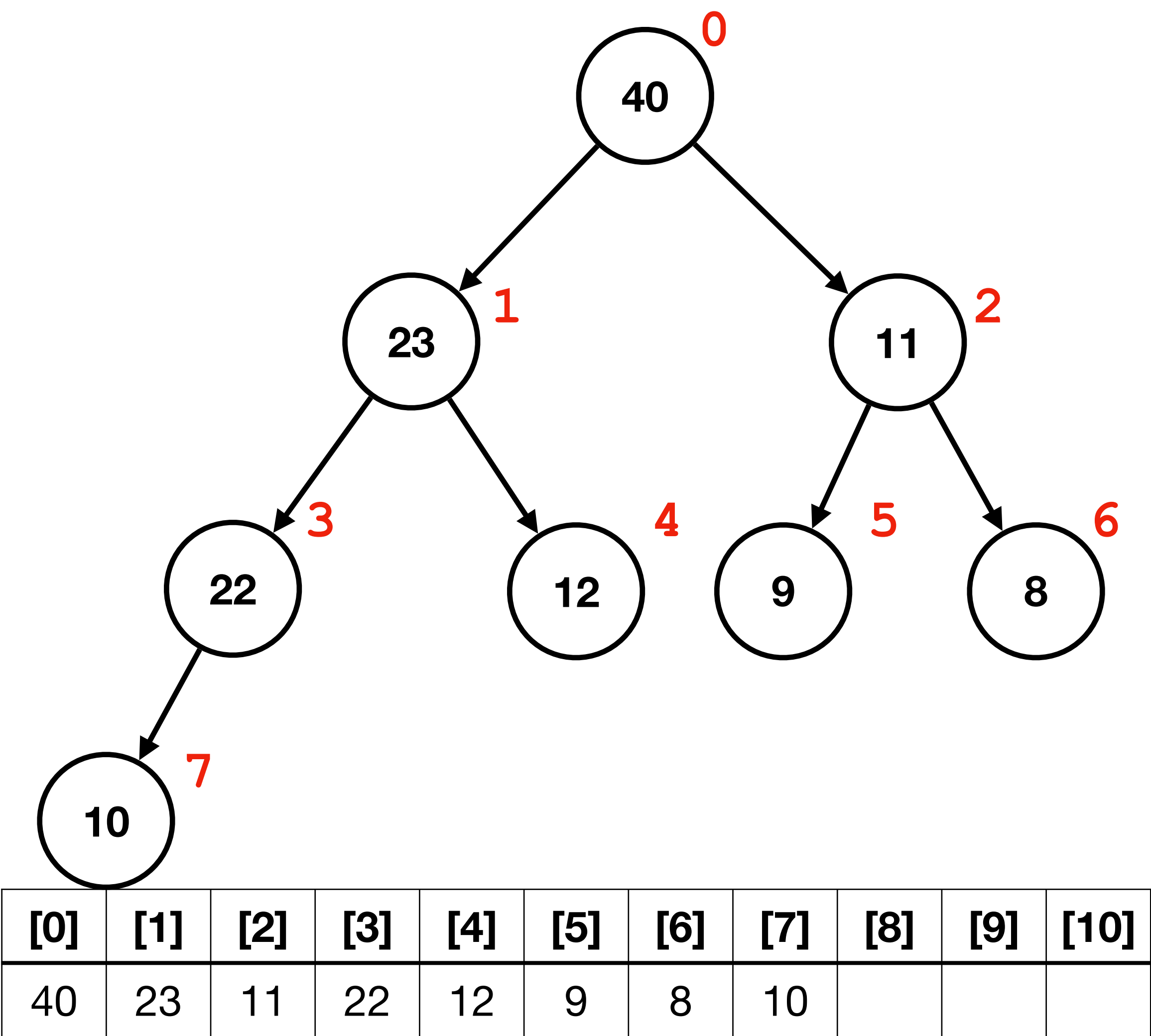


Heap

Bubble Down

```
int pop_heap(int *arr, int len)
{
    int top = arr[0];
    arr[0] = arr[len - 1];

    bubble_down(arr, len - 1, 0);
}
```



Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap			

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap	$O(\log n)$		

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap	$O(\log n)$	$O(1)$	

Priority Queue

Implementations

	<code>insert</code>	<code>get_top</code>	<code>remove_top</code>
ArrayList	$O(1)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(1)$	$O(1)$
Sorted Linked List	$O(n)$	$O(1)$	$O(1)$
General BST	$O(n)$	$O(n)$	$O(n)$
Balanced BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap	$O(\log n)$	$O(1)$	$O(\log n)$

Heap

Heap Sort

Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

- We could insert each in turn.

Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

- We could insert each in turn.
- Insertion takes $O(\log n)$, doing it n times -- total complexity $O(n \log n)$.

Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

- We could insert each in turn.
- Insertion takes $O(\log n)$, doing it n times -- total complexity $O(n \log n)$.
- It's not bad, but we can do better!

Heap
Heapify

Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

Heap

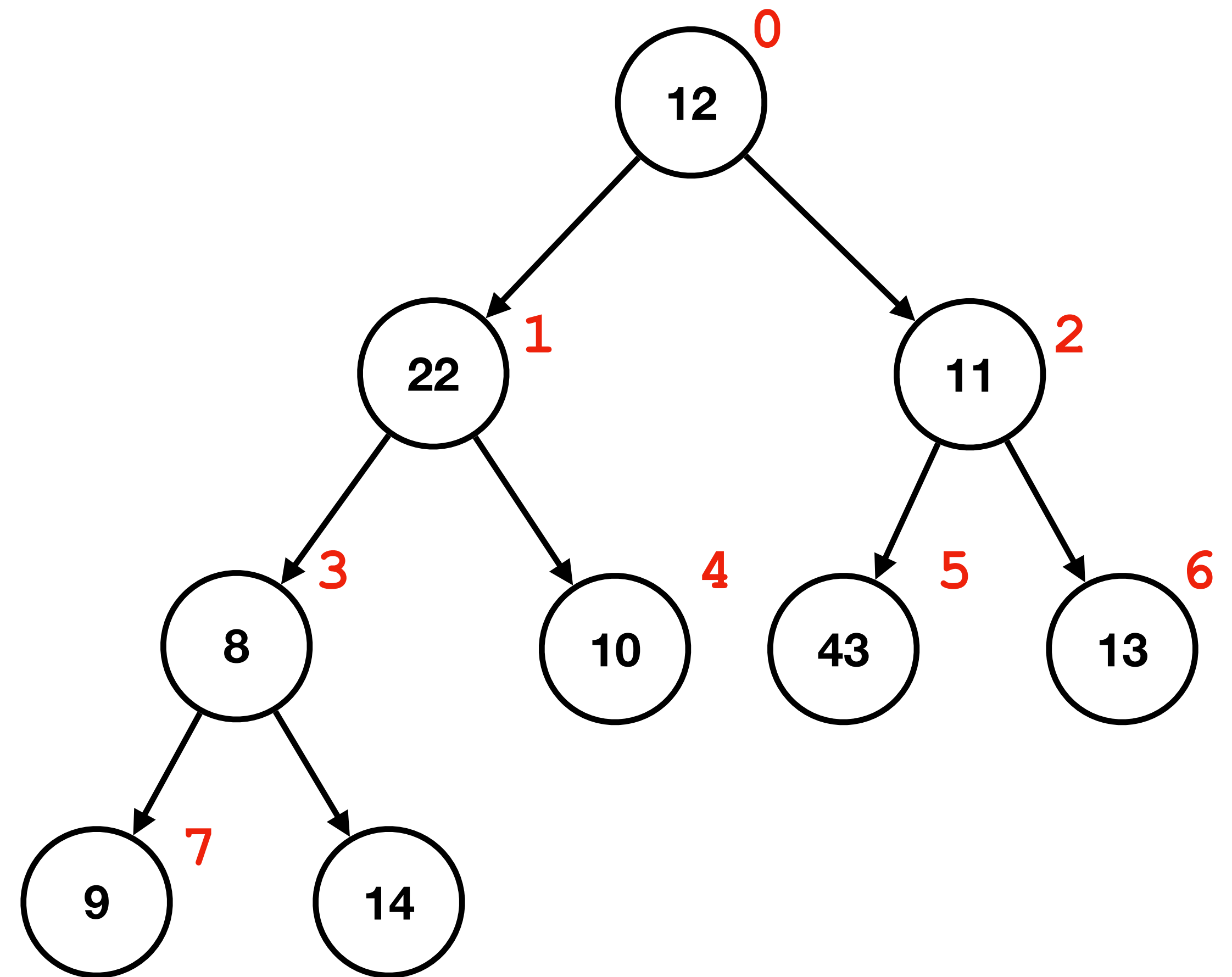
Heapify

- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?

Heap

Heapify

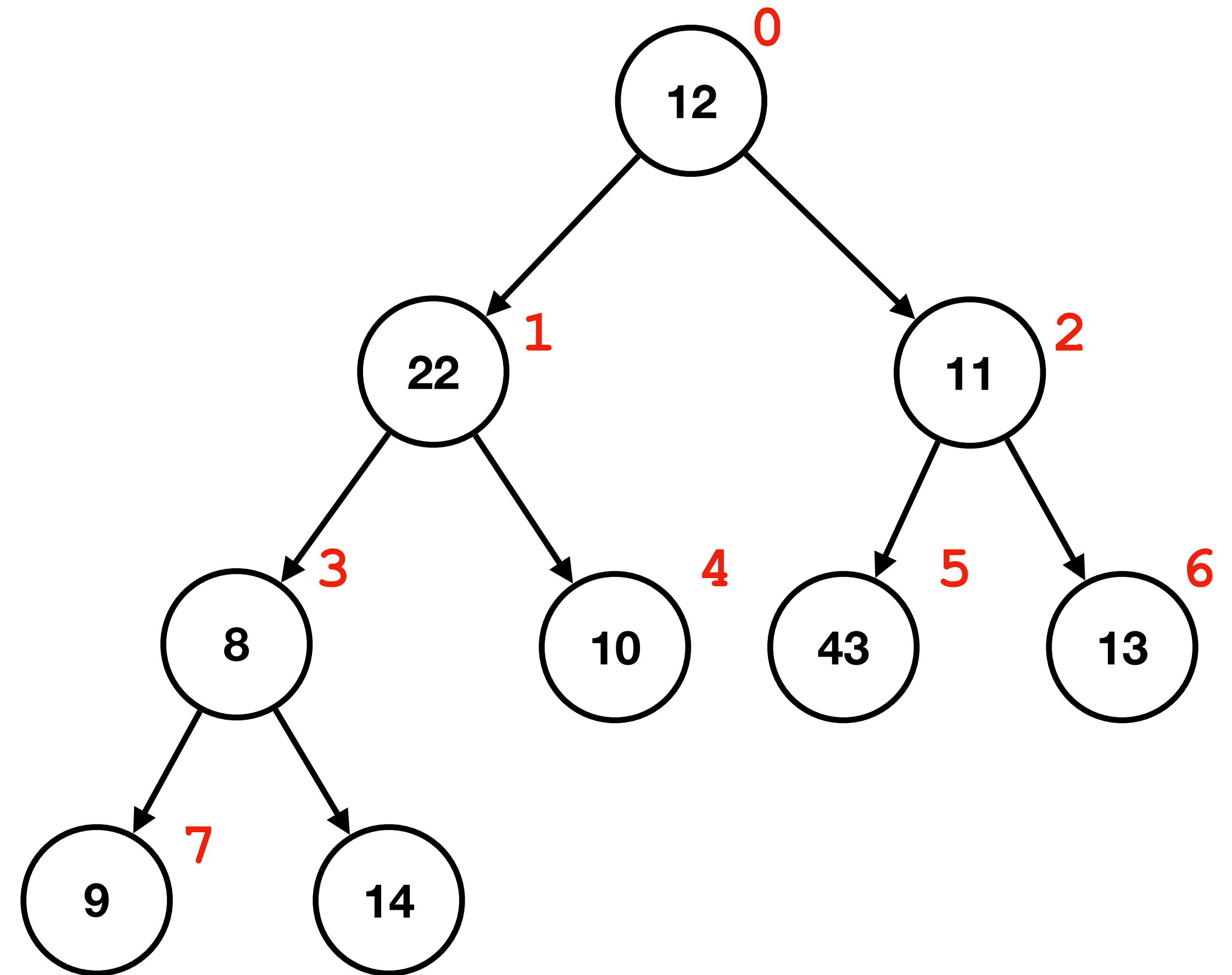
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?



Heap

Heapify

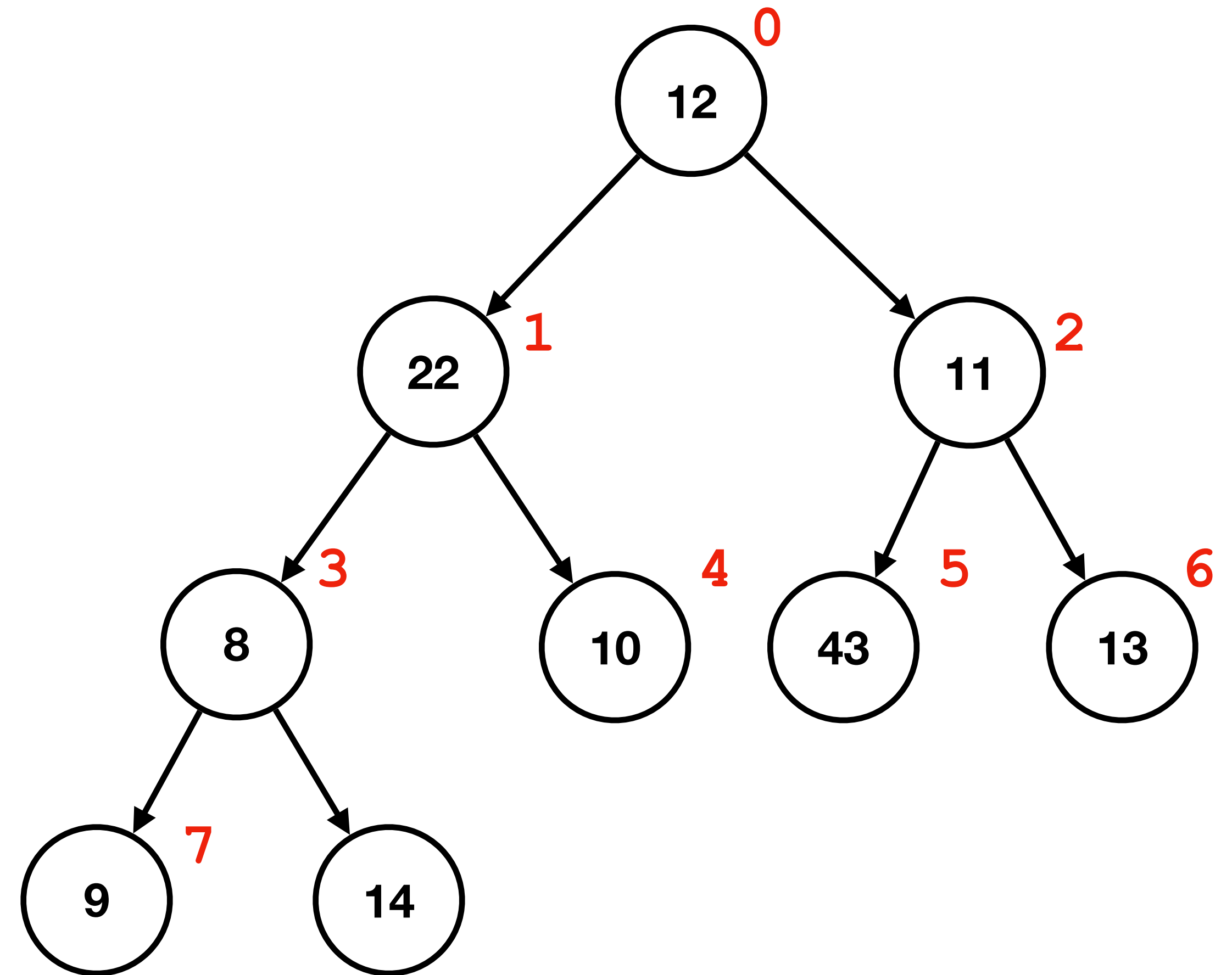
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?
 - Shape property is satisfied



Heap

Heapify

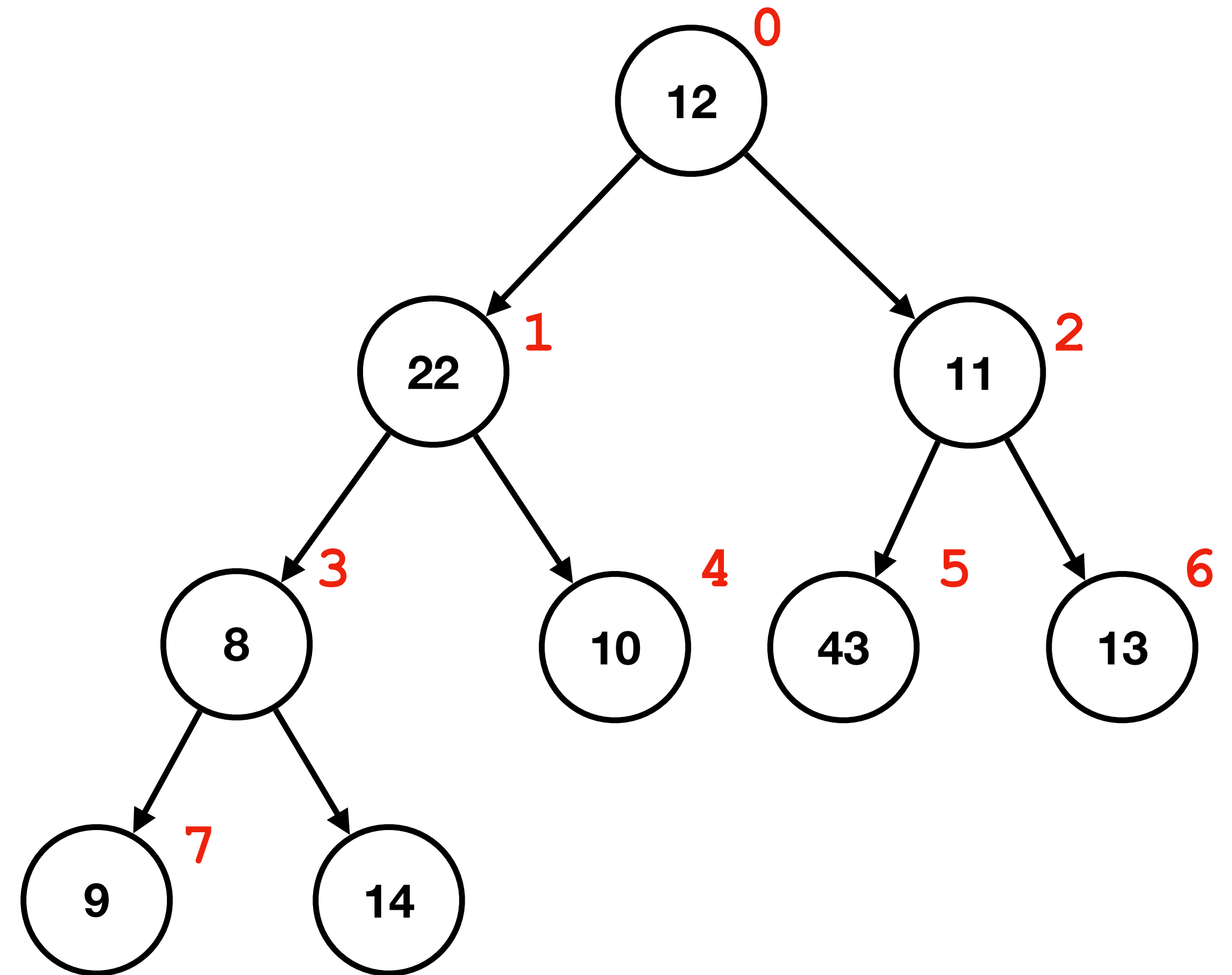
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?
 - Shape property is satisfied
 - Value property is not, but...



Heap

Heapify

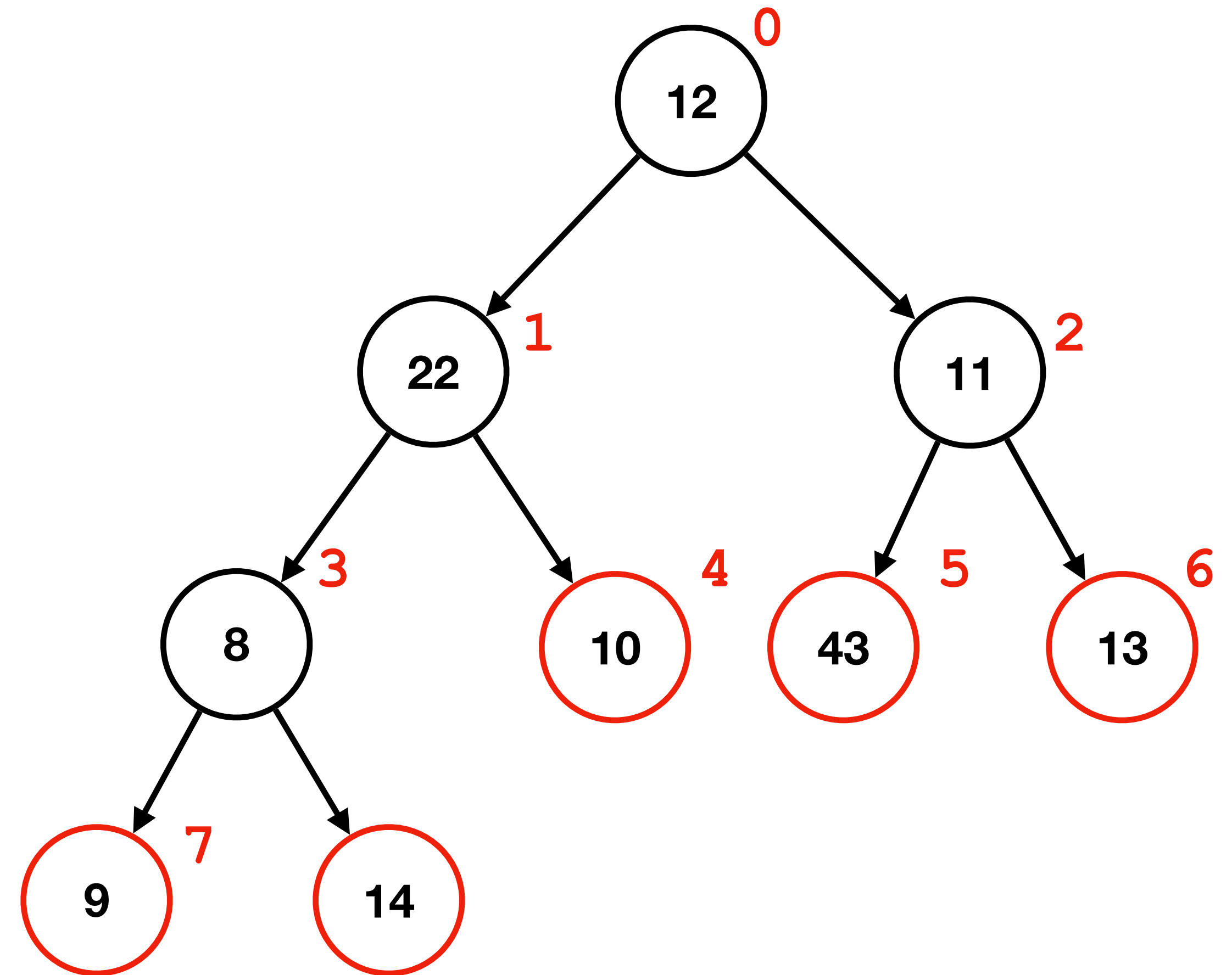
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?
 - Shape property is satisfied
 - Value property is not, but...
 - All leaves satisfy the value property (they have no children)



Heap

Heapify

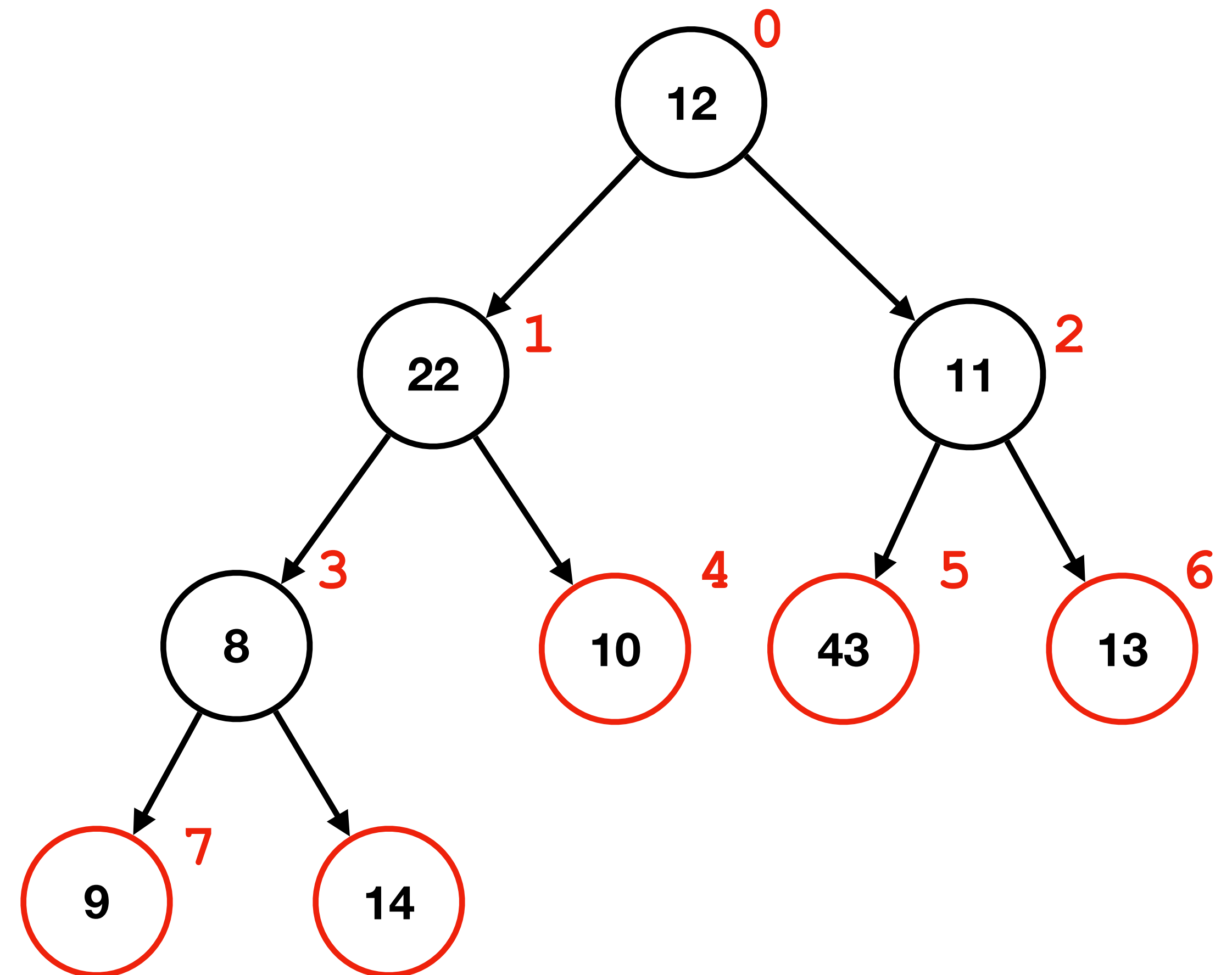
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?
 - Shape property is satisfied
 - Value property is not, but...
 - All leaves satisfy the value property (they have no children)



Heap

Heapify

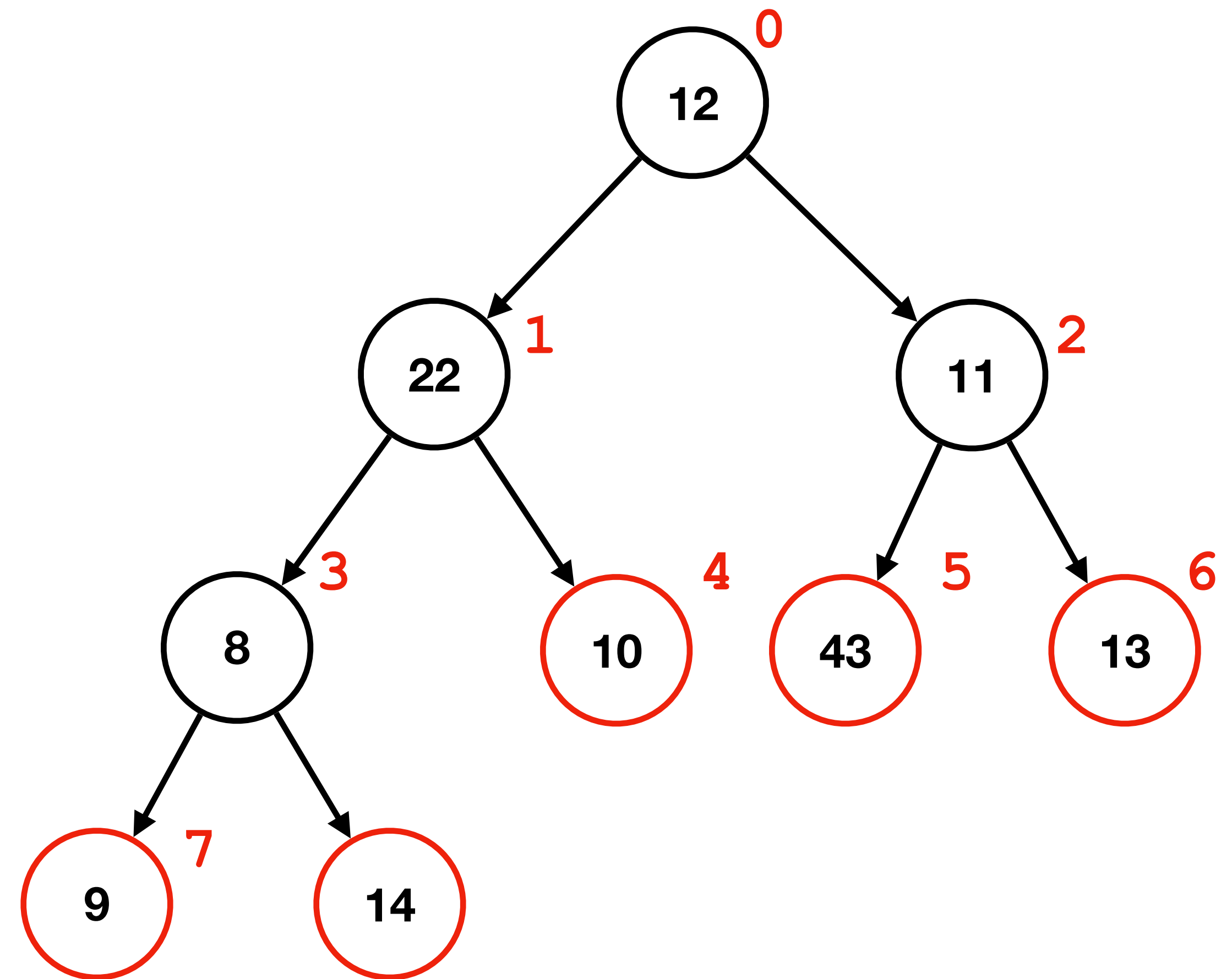
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?



Heap

Heapify

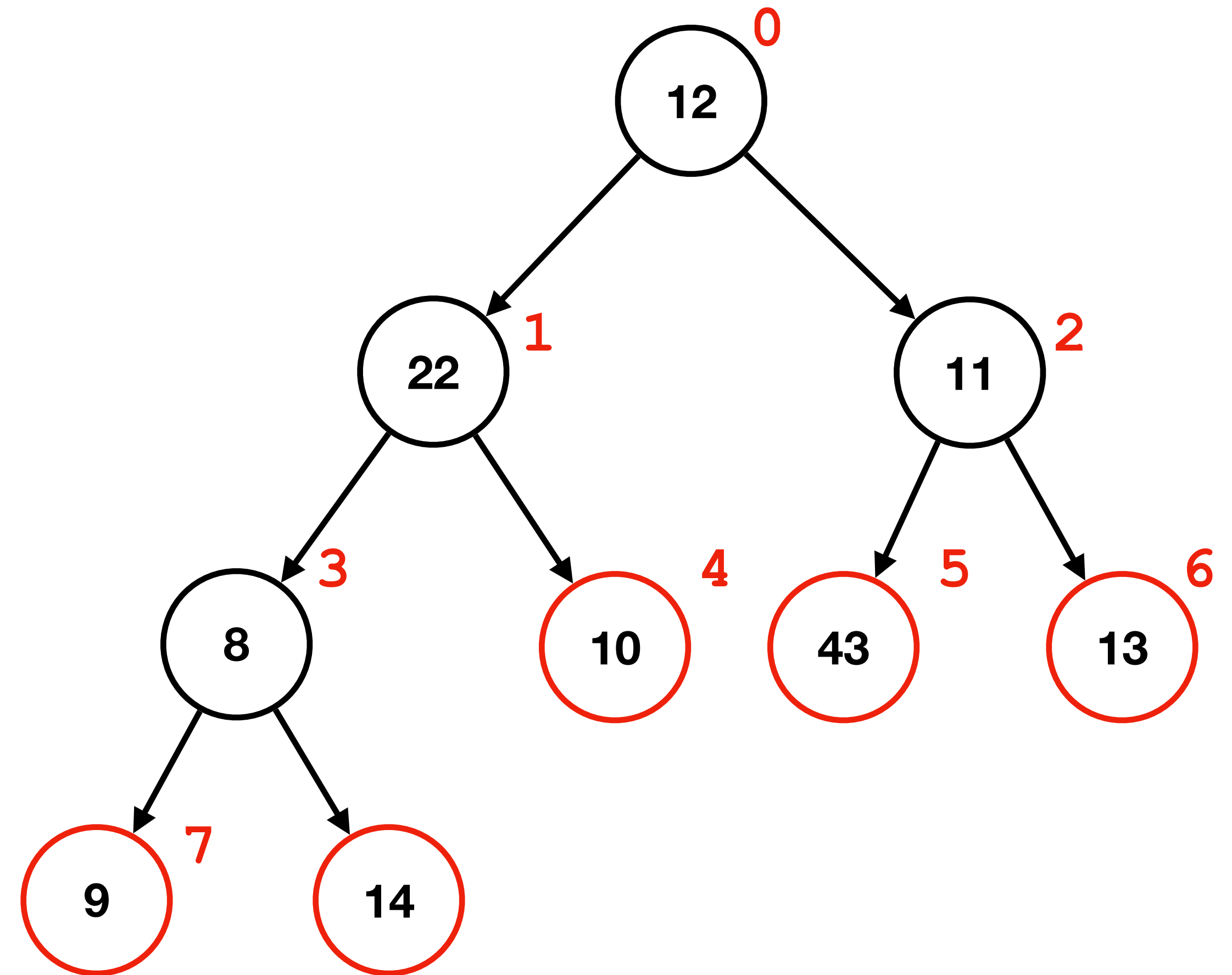
- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?
- *Insight:* To make non-leaves satisfy the value property, perform bubble-down on every non-leaf node, from bottom up.



Heap

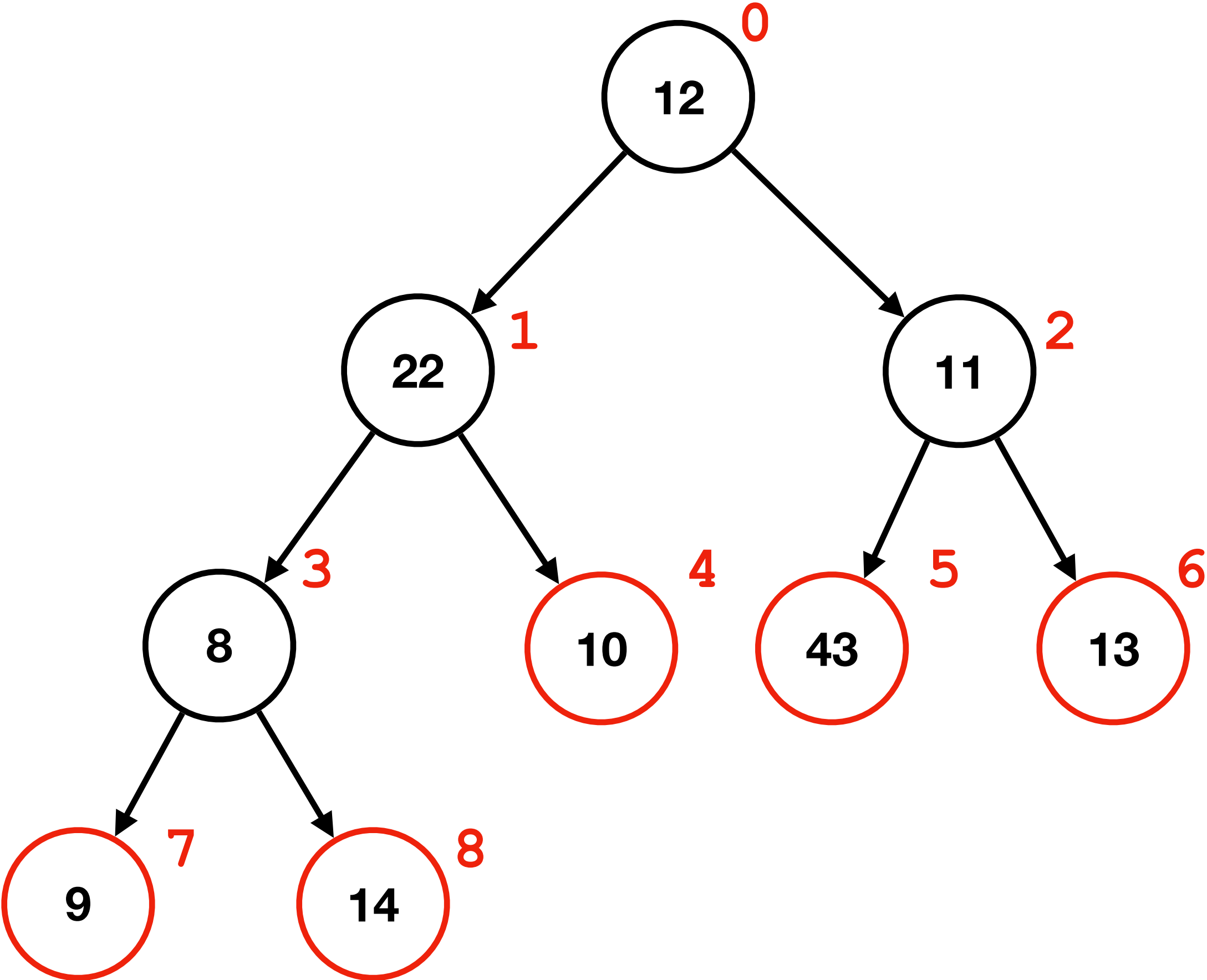
Heapify

- Given an unsorted array:
12, 22, 11, 8, 10, 43, 13, 9, 14
- What if we just call this a *heap*?
- *Insight:* To make non-leaves satisfy the value property, perform bubble-down on every non-leaf node, from bottom up.
- Bubble down assumes that both subtrees are valid heaps.



Heap

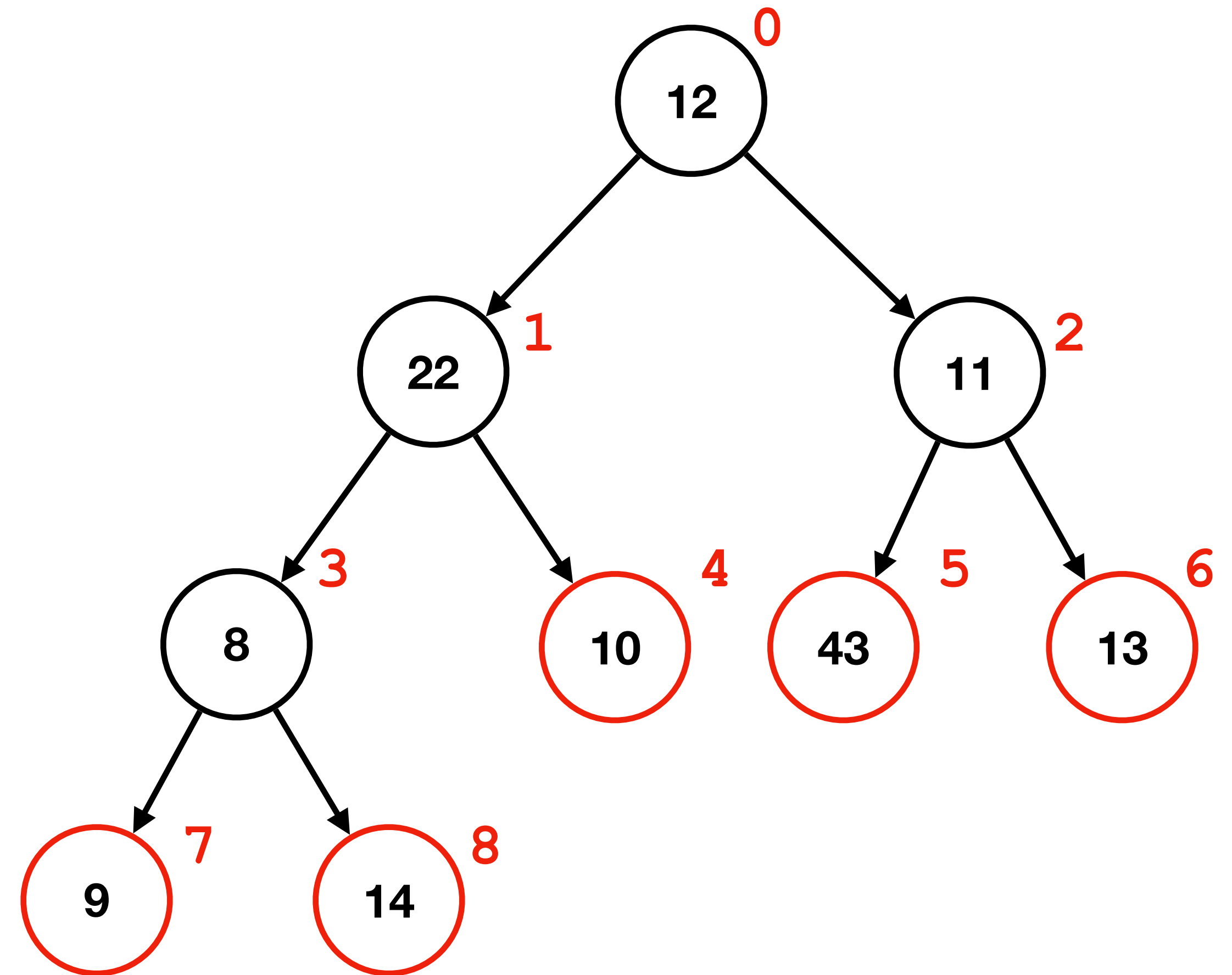
Heapify



Heap

Heapify

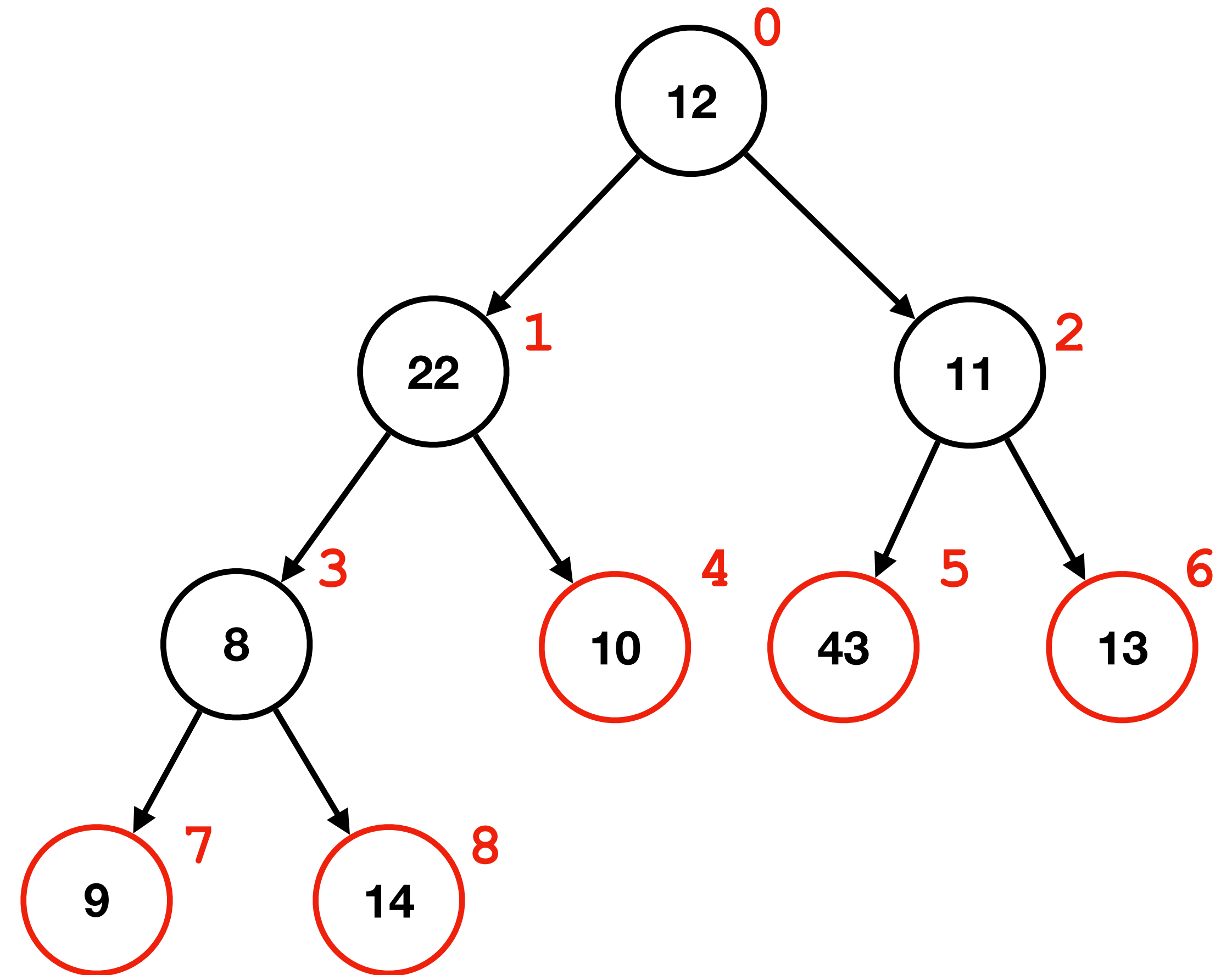
- Question: What is the array index of the last (lowest, rightmost) non-leaf node?



Heap

Heapify

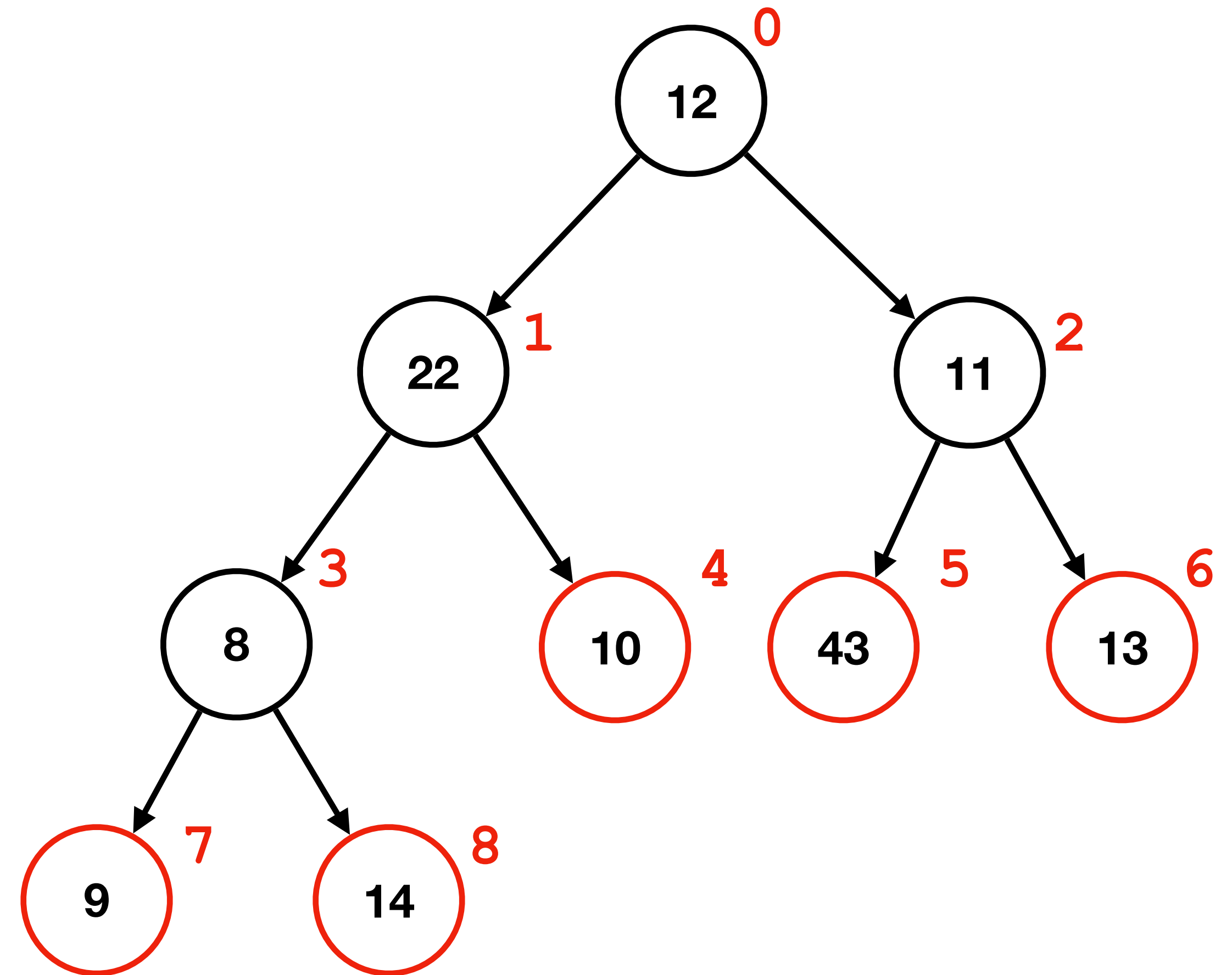
- Question: What is the array index of the last (lowest, rightmost) non-leaf node?
- Answer: $\text{heap_size} / 2 - 1$



Heap

Heapify

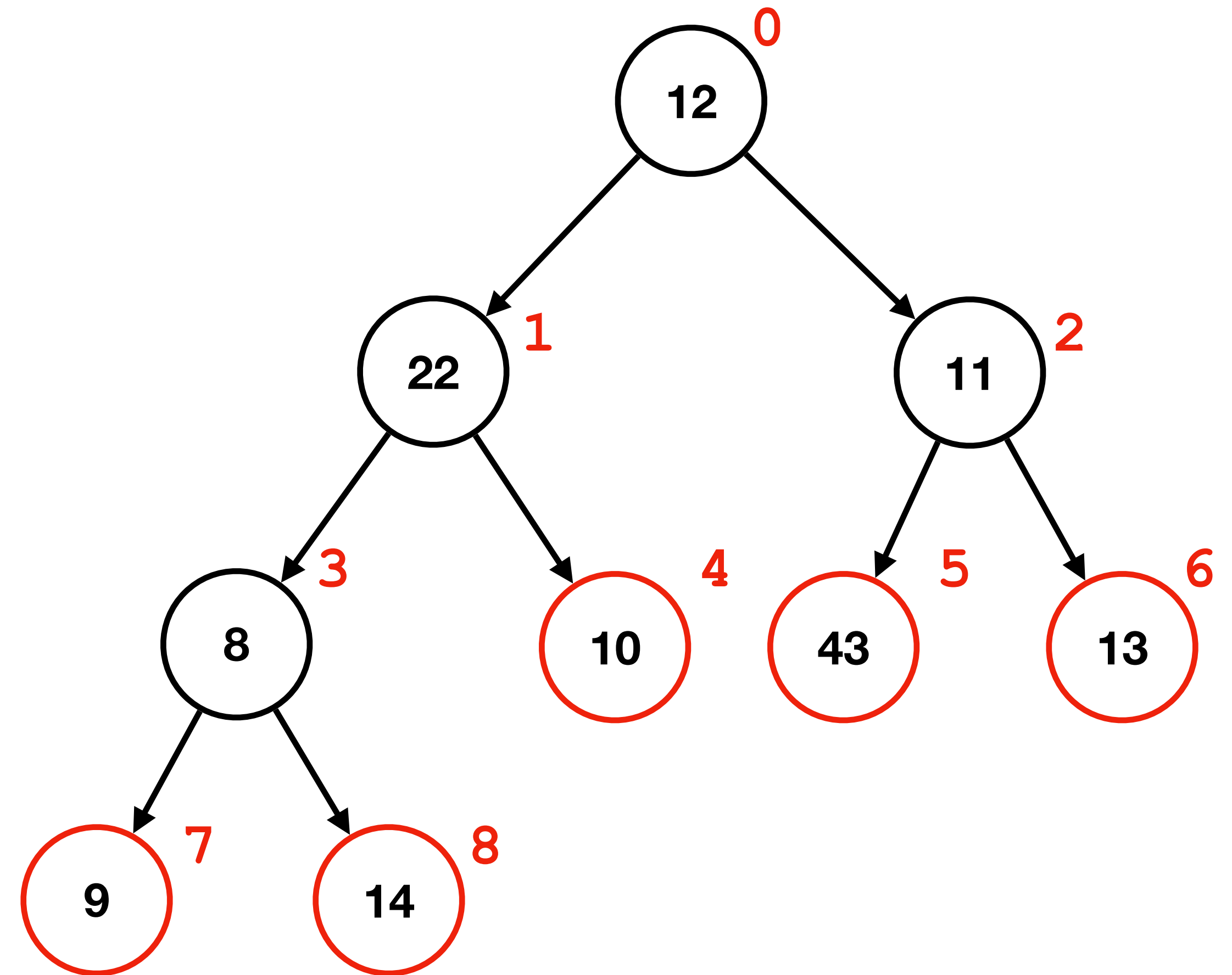
- Question: What is the array index of the last (lowest, rightmost) non-leaf node?
- Answer: $\text{heap_size} / 2 - 1$
 - Last leaf is $(\text{heap_size} - 1)$



Heap

Heapify

- Question: What is the array index of the last (lowest, rightmost) non-leaf node?
- Answer: $\text{heap_size} / 2 - 1$
 - Last leaf is $(\text{heap_size} - 1)$
 - Parent of last leaf is $(\text{heap_size} - 1 - 1) / 2$



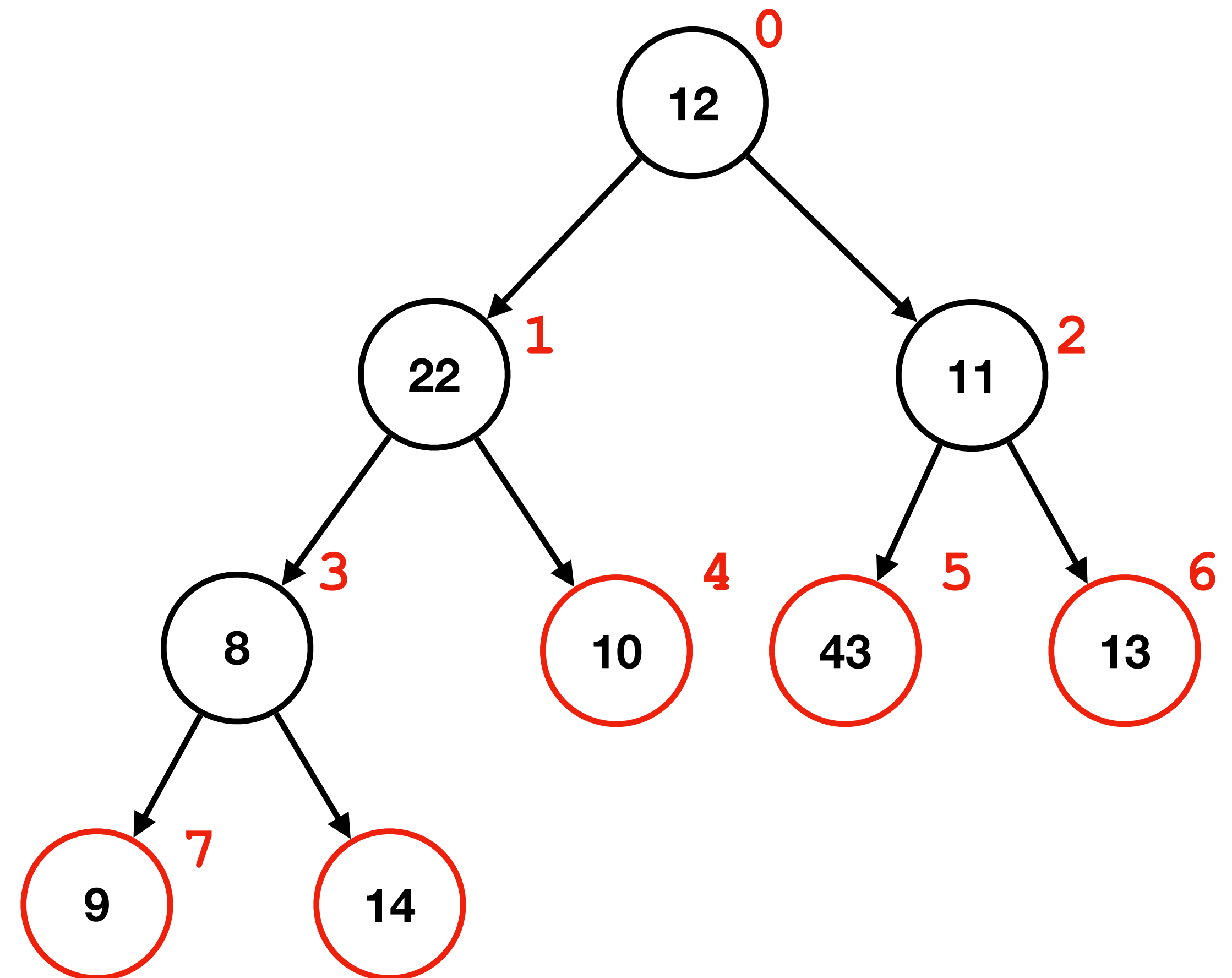
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards: perform `bubble_down` on every non-leaf node



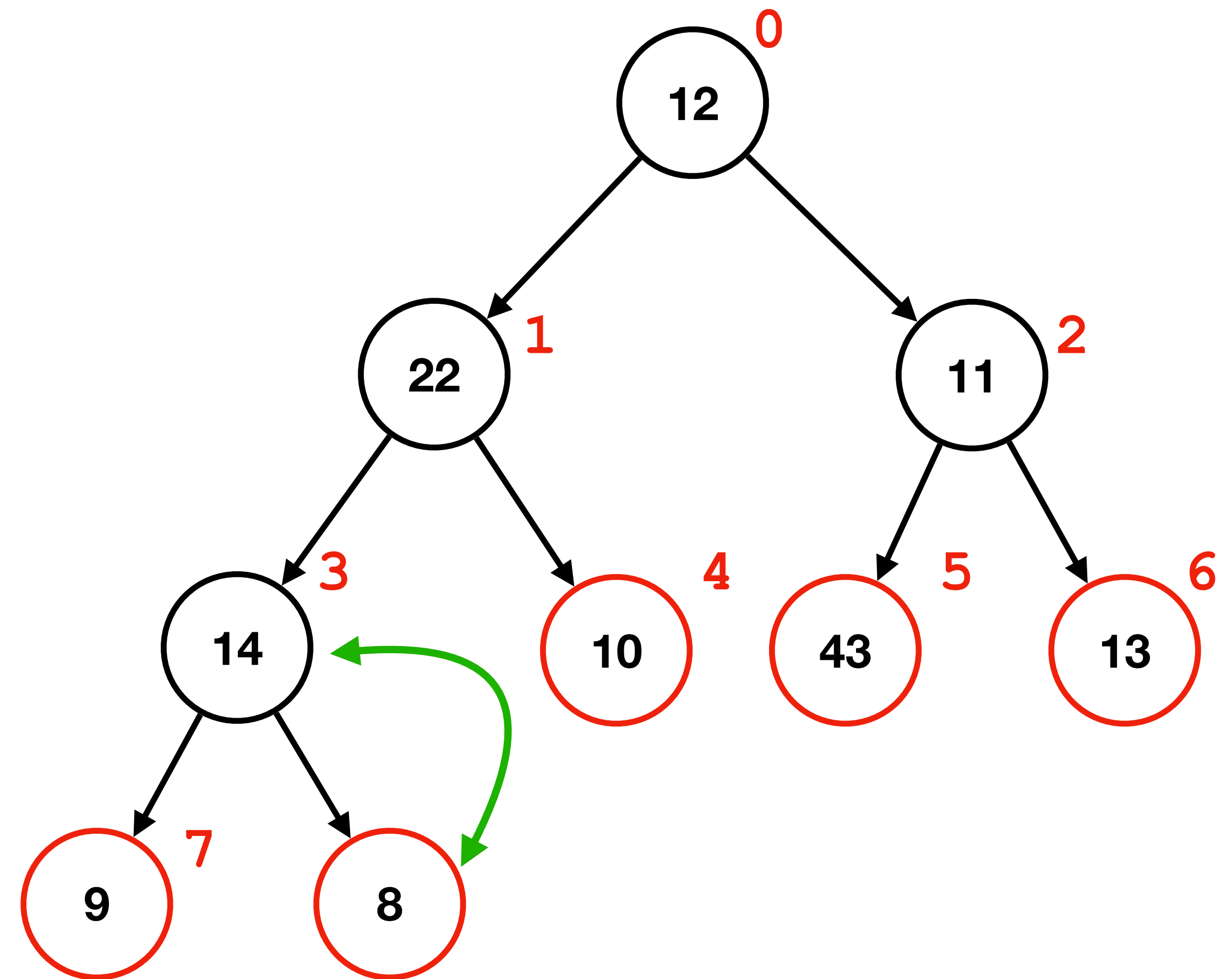
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards: perform `bubble_down` on every non-leaf node



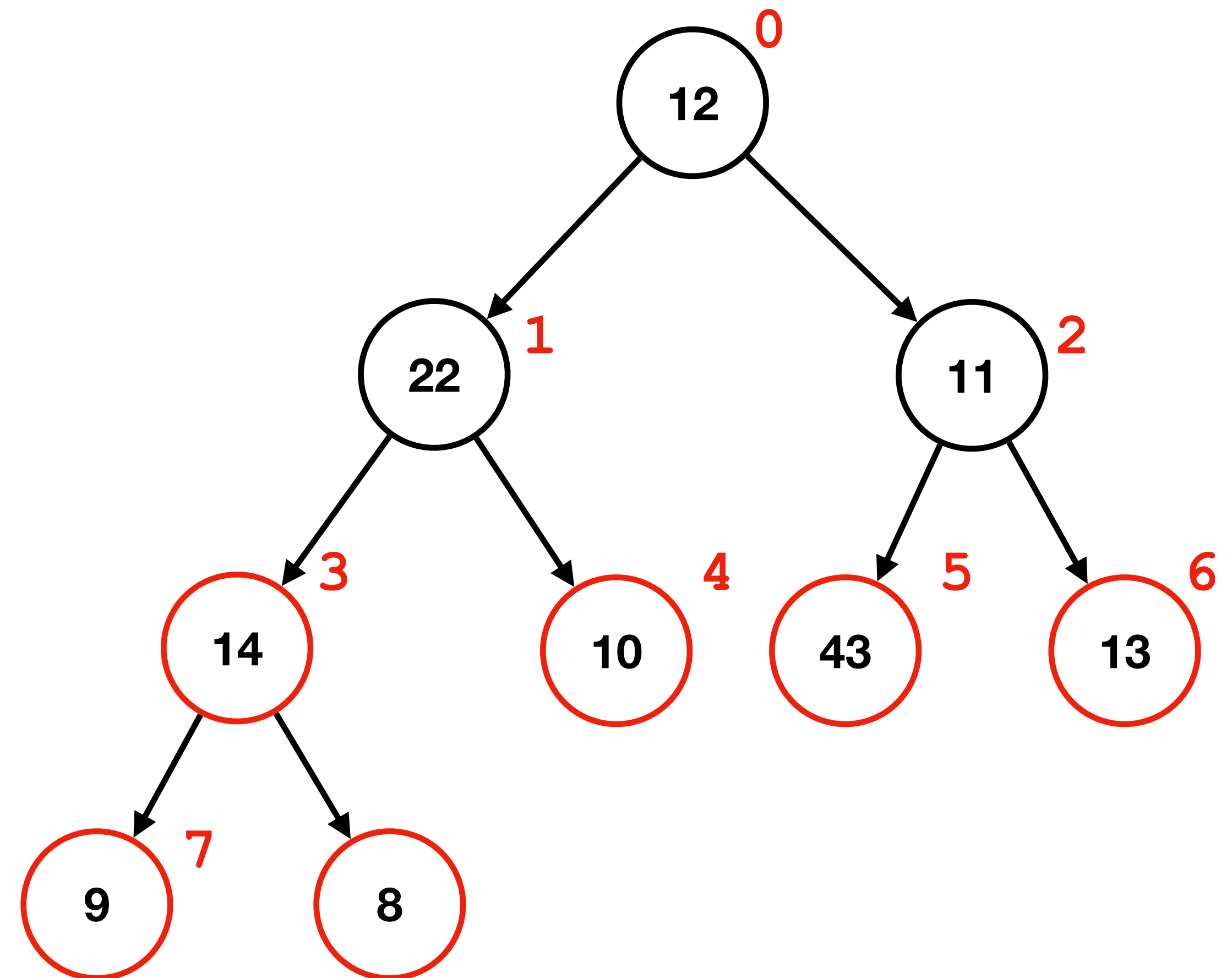
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards: perform `bubble_down` on every non-leaf node



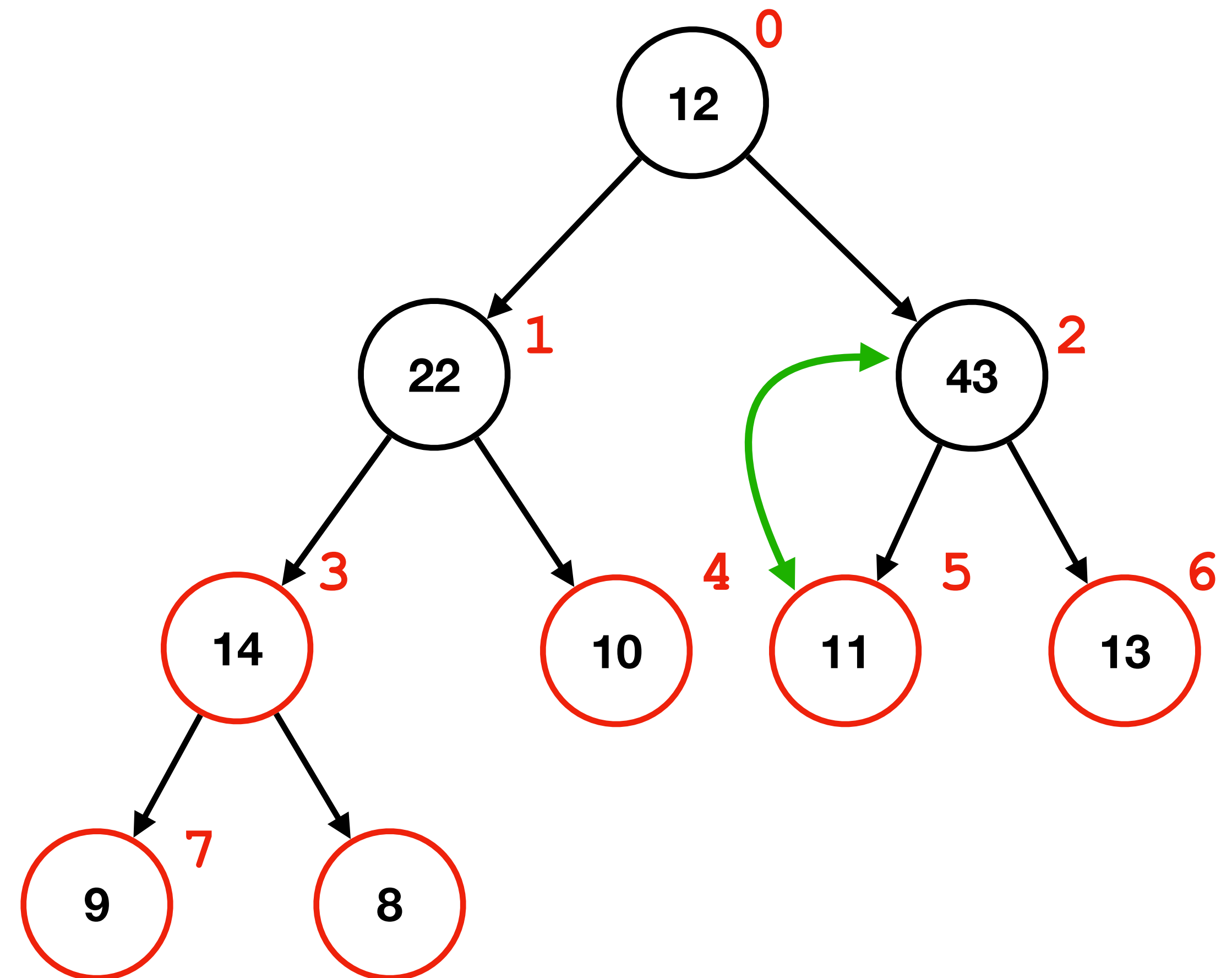
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

- Call this a heap.
- Starting at index $\text{heap_size} / 2 - 1$ and moving backwards:
perform `bubble_down` on every non-leaf node



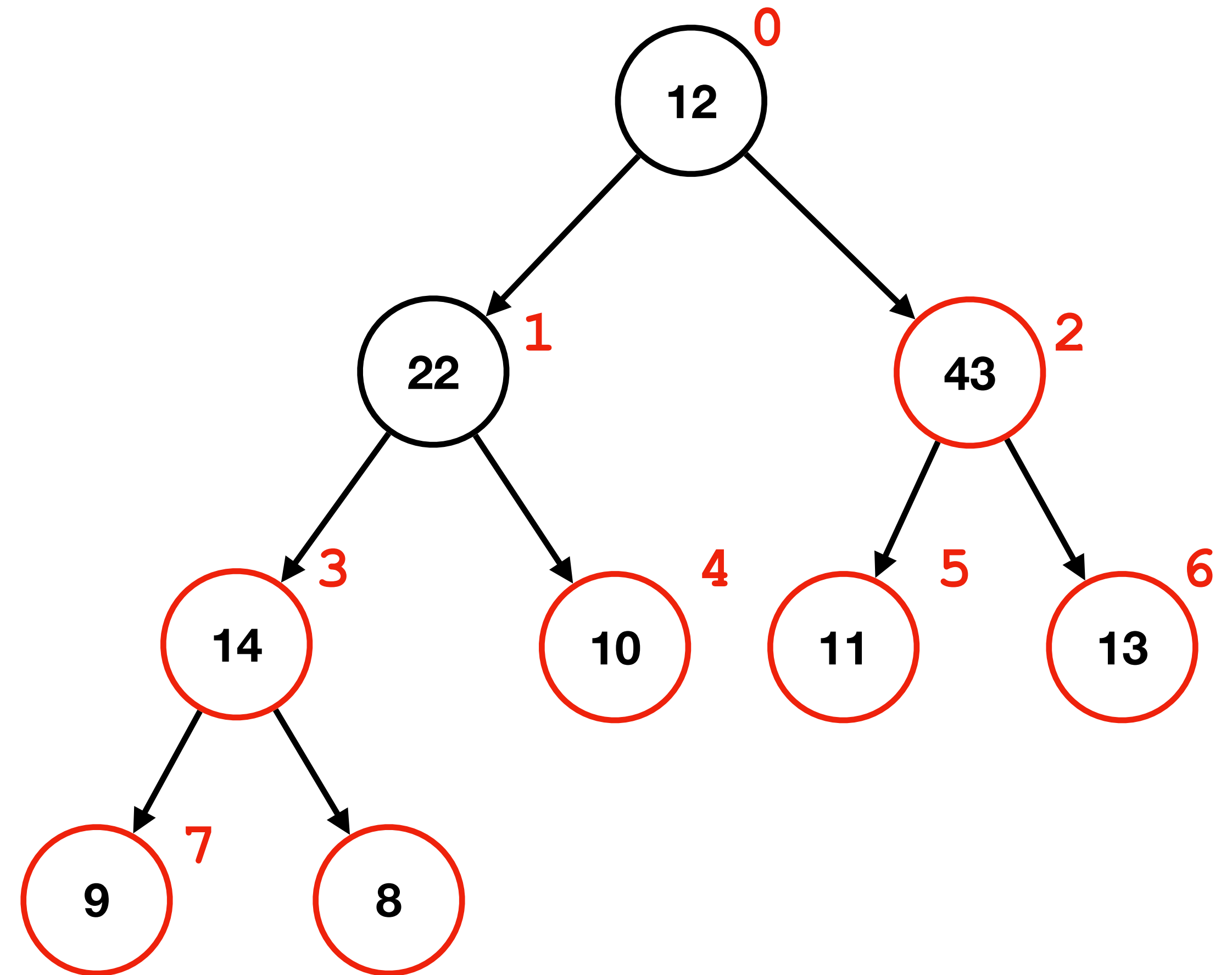
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards: perform `bubble_down` on every non-leaf node



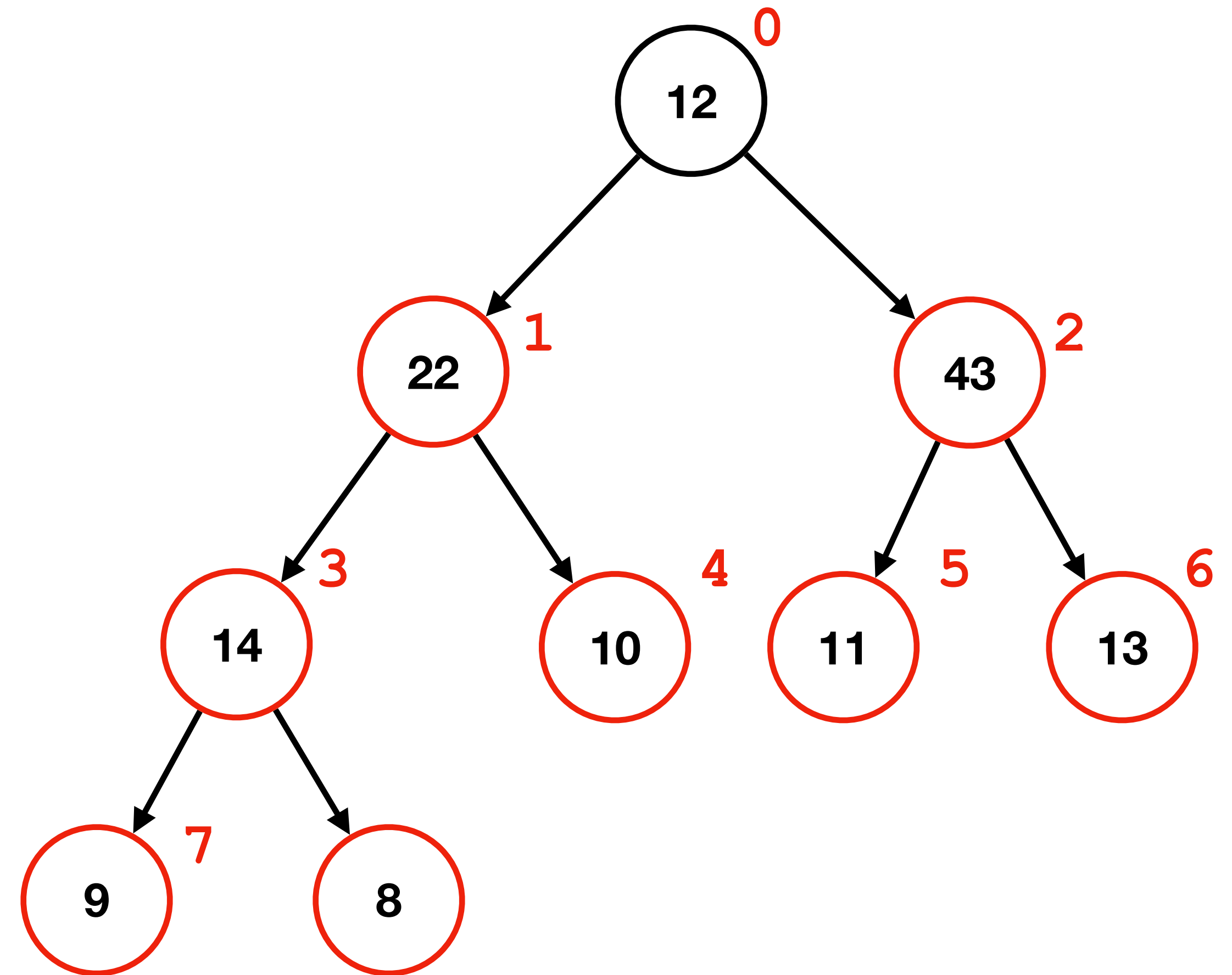
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards: perform `bubble_down` on every non-leaf node



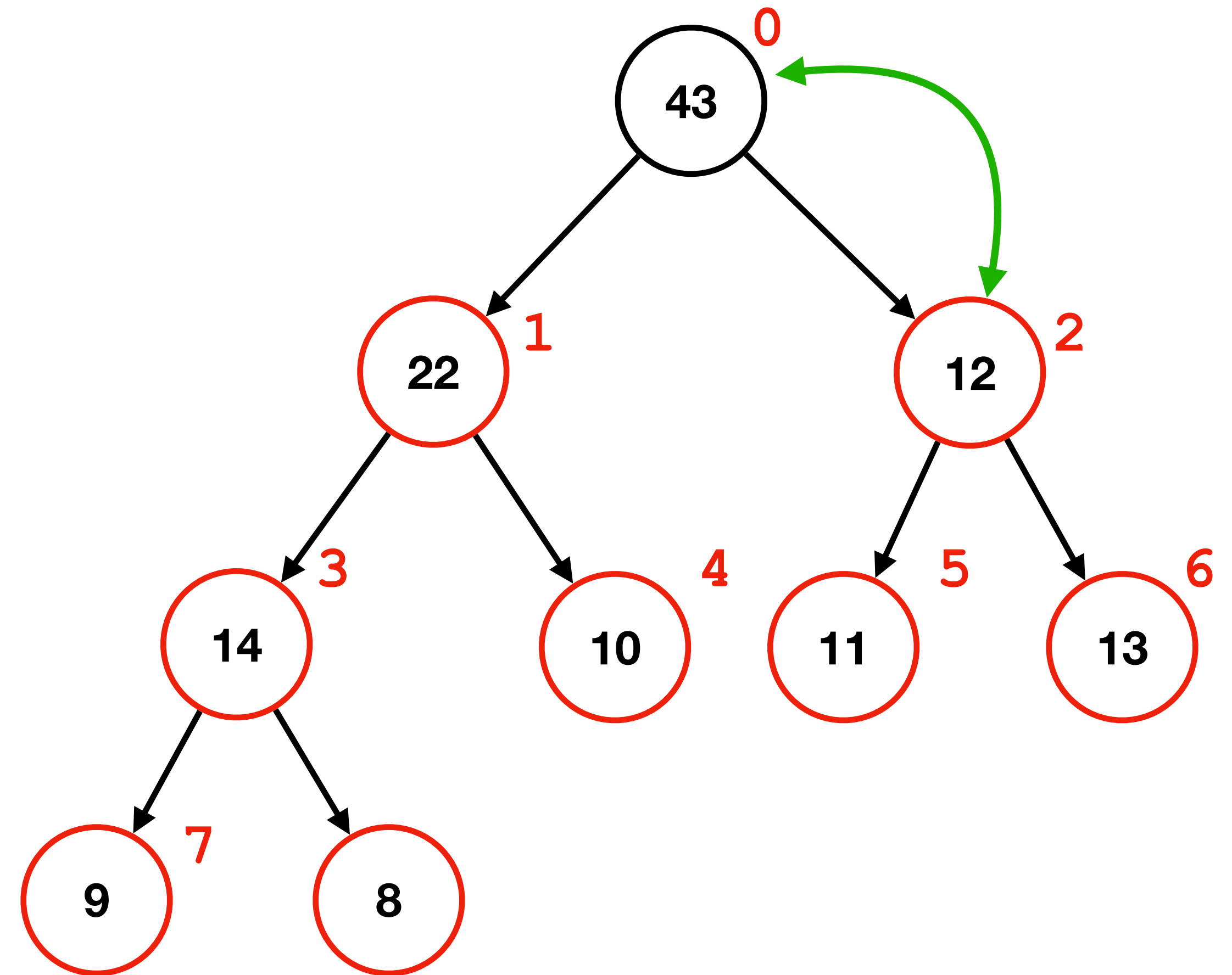
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards: perform `bubble_down` on every non-leaf node



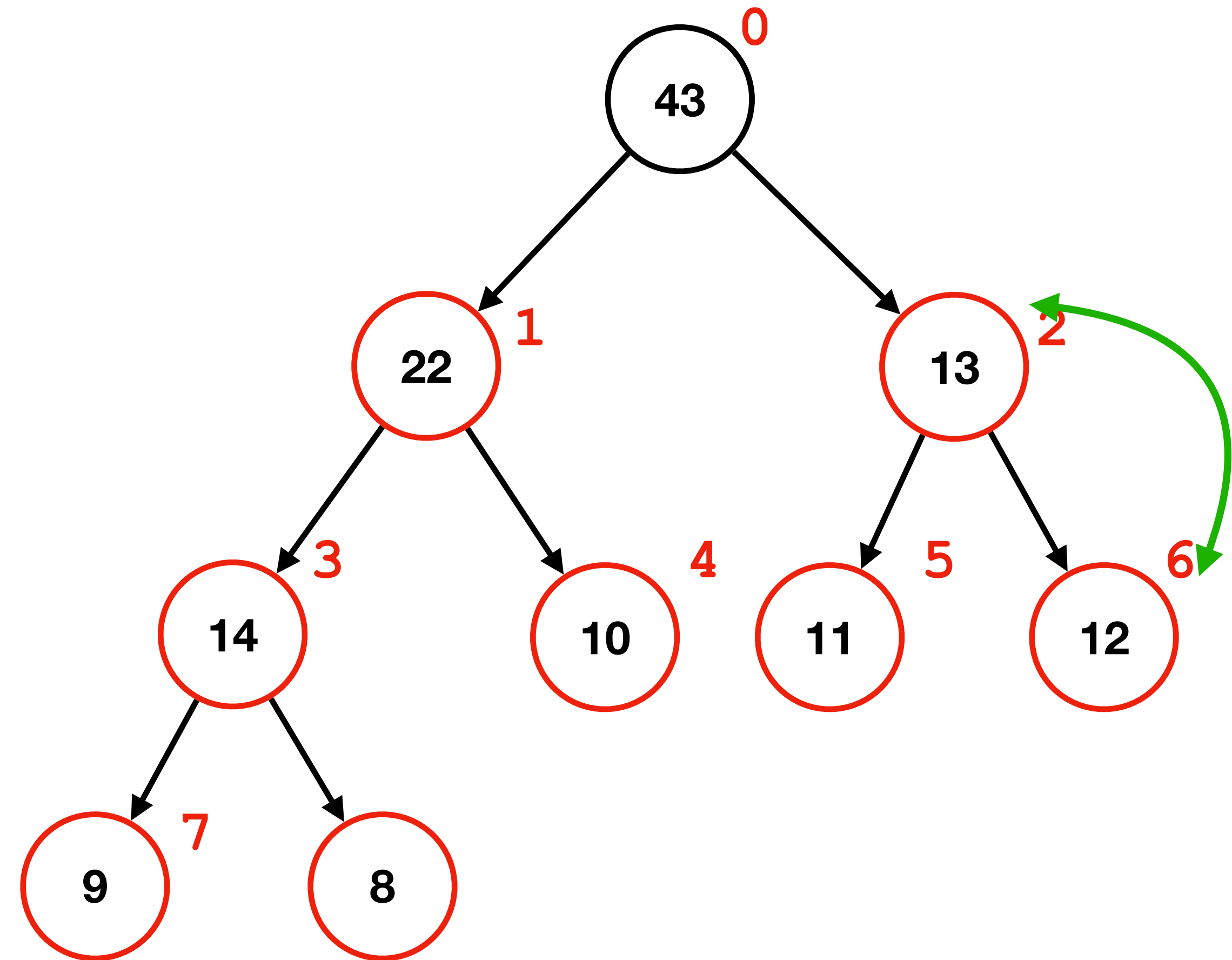
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards:
perform `bubble_down` on every non-leaf node



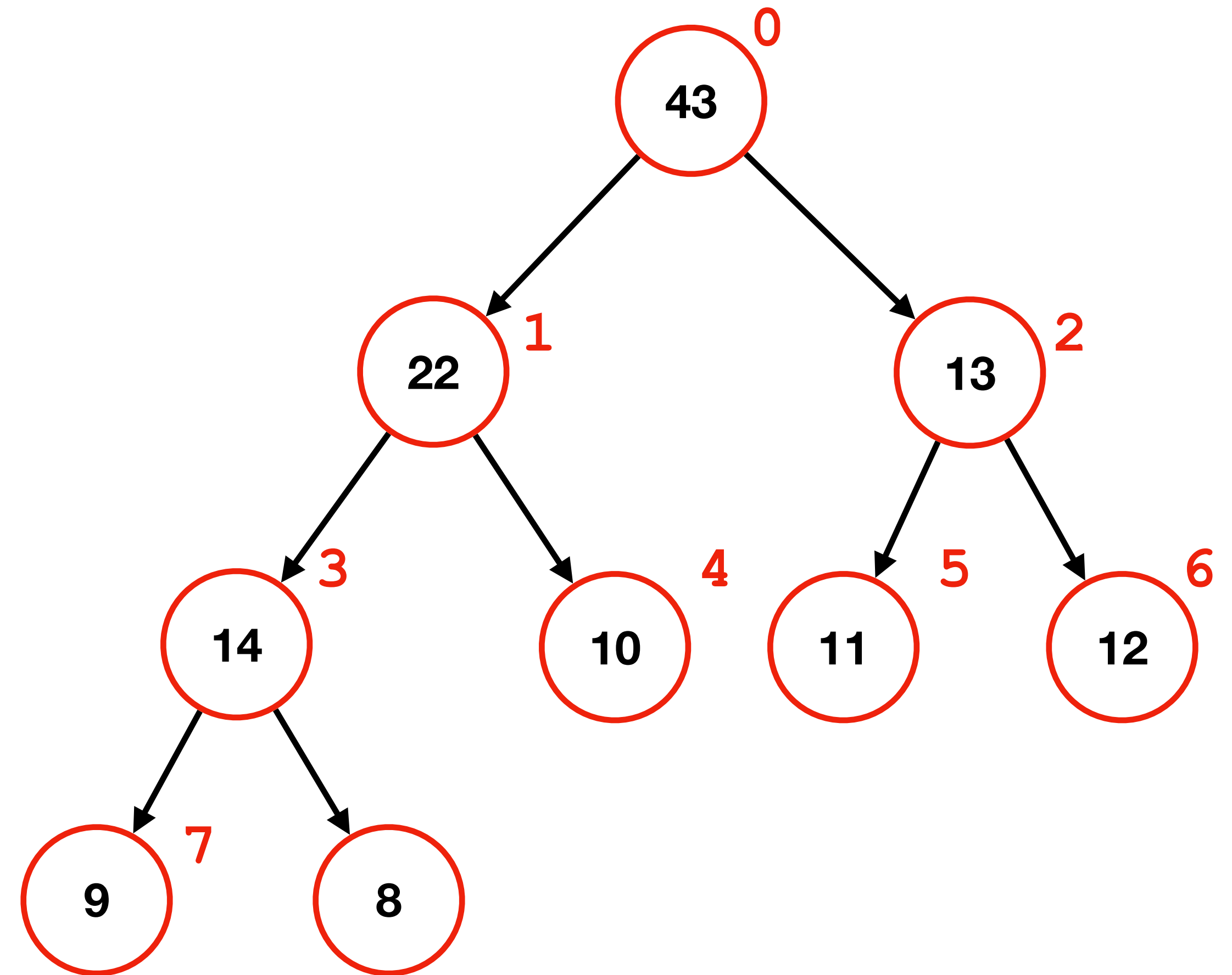
Heap

Heapify

- Given an unsorted array:

12, 22, 11, 8, 10, 43, 13, 9, 14

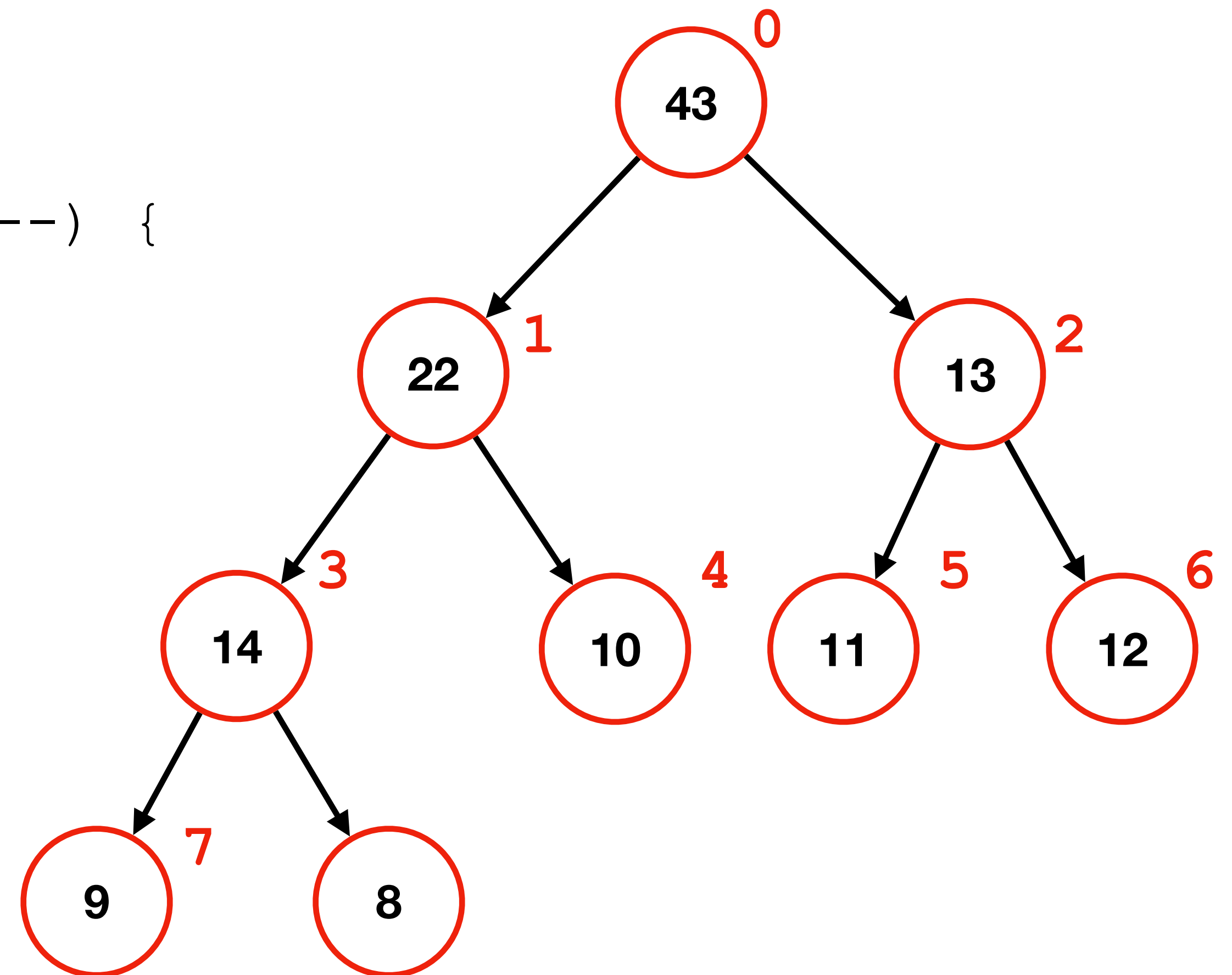
1. Call this a heap.
2. Starting at index $\text{heap_size} / 2 - 1$ and moving backwards:
perform `bubble_down` on every non-leaf node



Heap

Heapify

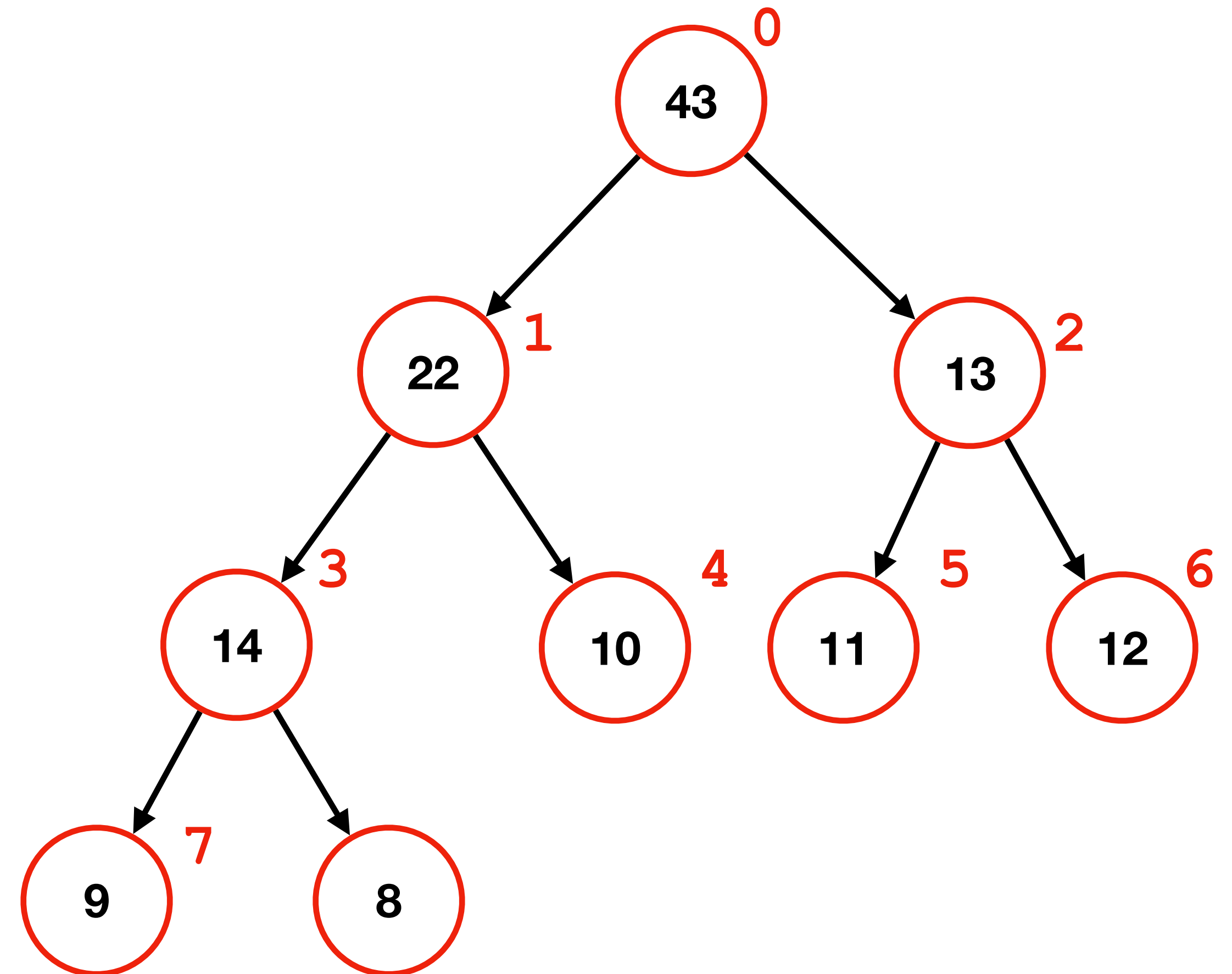
```
void heapify(int *arr, int len)
{
    for (int i = len / 2 - 1; i >= 0; i--) {
        bubble_down(arr, len, i);
    }
}
```



Heap

Heapify Complexity?

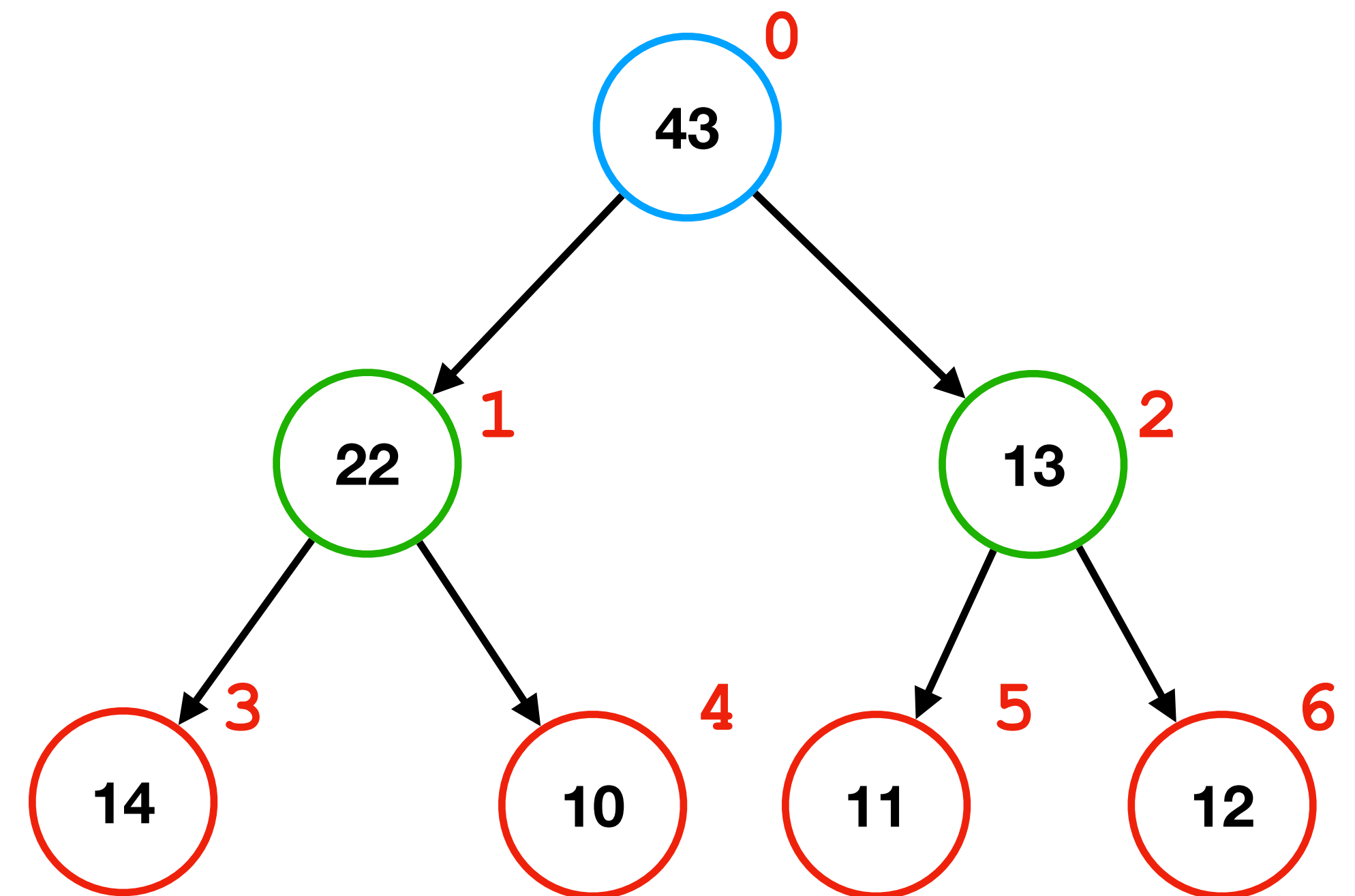
- CMSC 27200.
- Informal analysis:



Heap

Heapify Complexity?

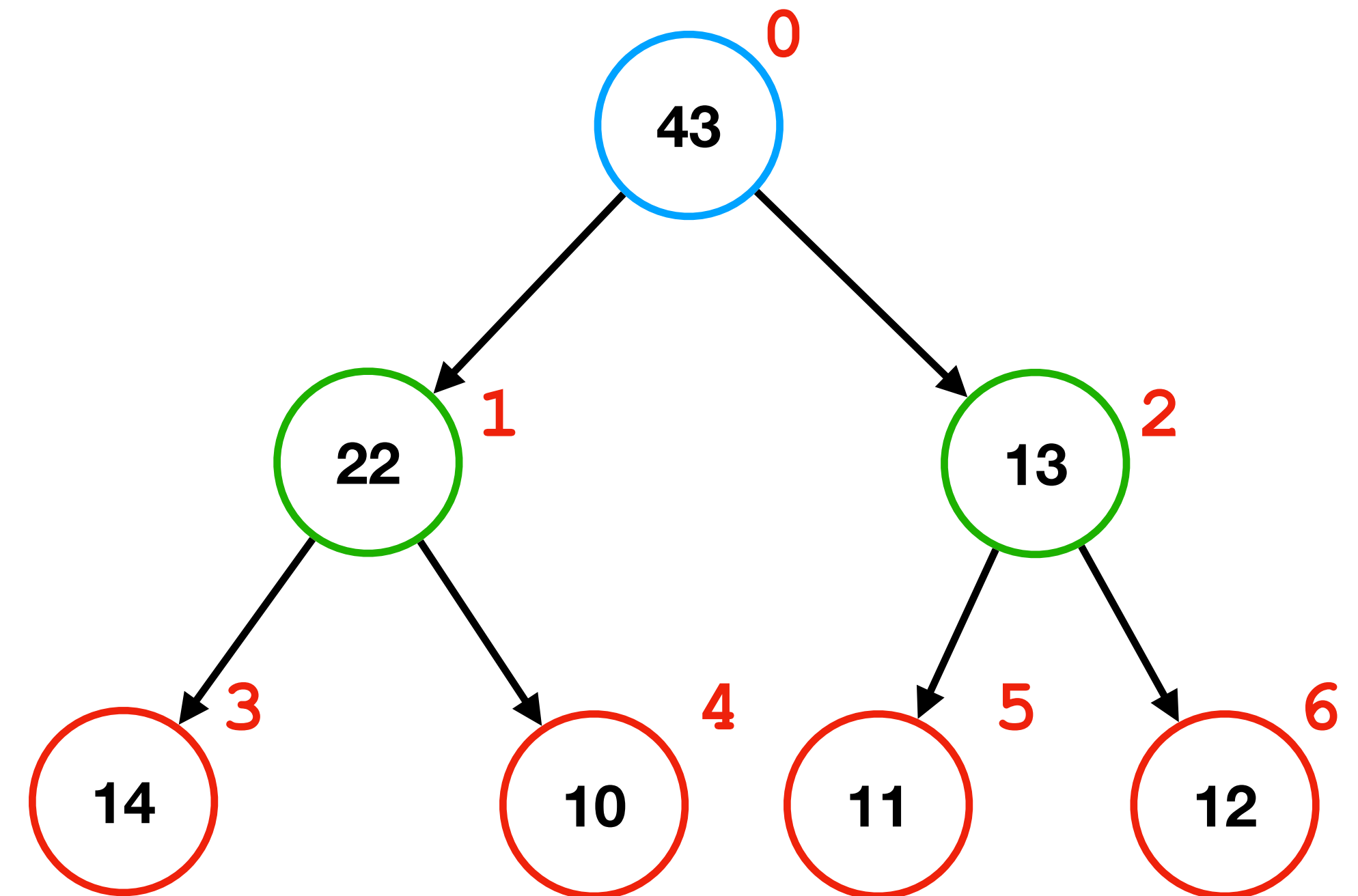
- CMSC 27200.
- Informal analysis:
 - Define level to be the maximum distance from the leaves.
 - red: level 0
 - green: level 1
 - blue: level 2



Heap

Heapify Complexity?

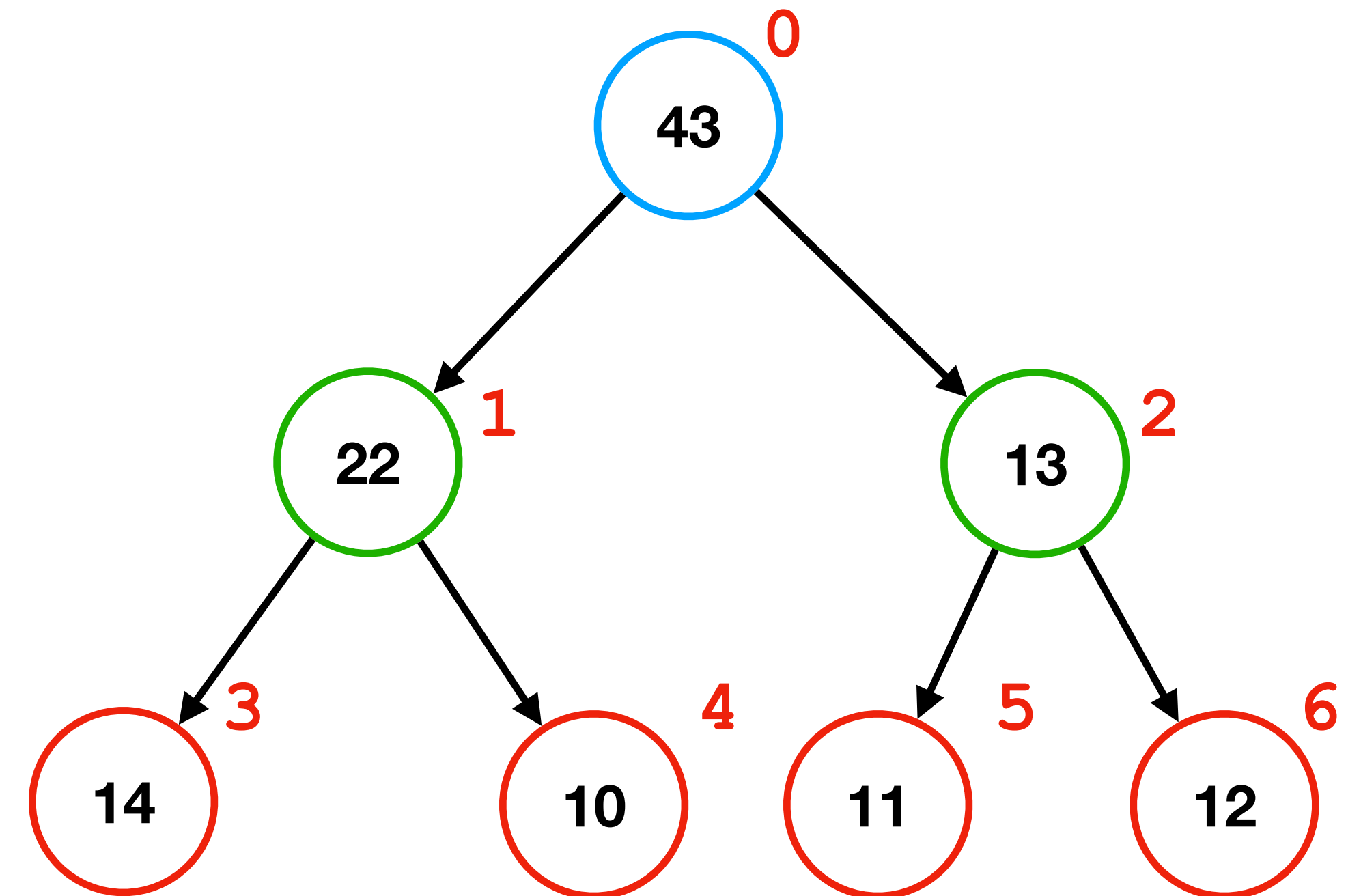
- CMSC 27200.
- Informal analysis:
 - Define *level* to be the maximum distance from the leaves.
 - If n is `heap_size`, there are at most...
 - $\lceil n/2^1 \rceil$ level 0 nodes ($7 / 2 \approx 4$)
 - $\lceil n/2^2 \rceil$ level 1 nodes ($7 / 4 \approx 2$)
 - $\lceil n/2^3 \rceil$ level 2 nodes ($7 / 8 \approx 1$)



Heap

Heapify Complexity?

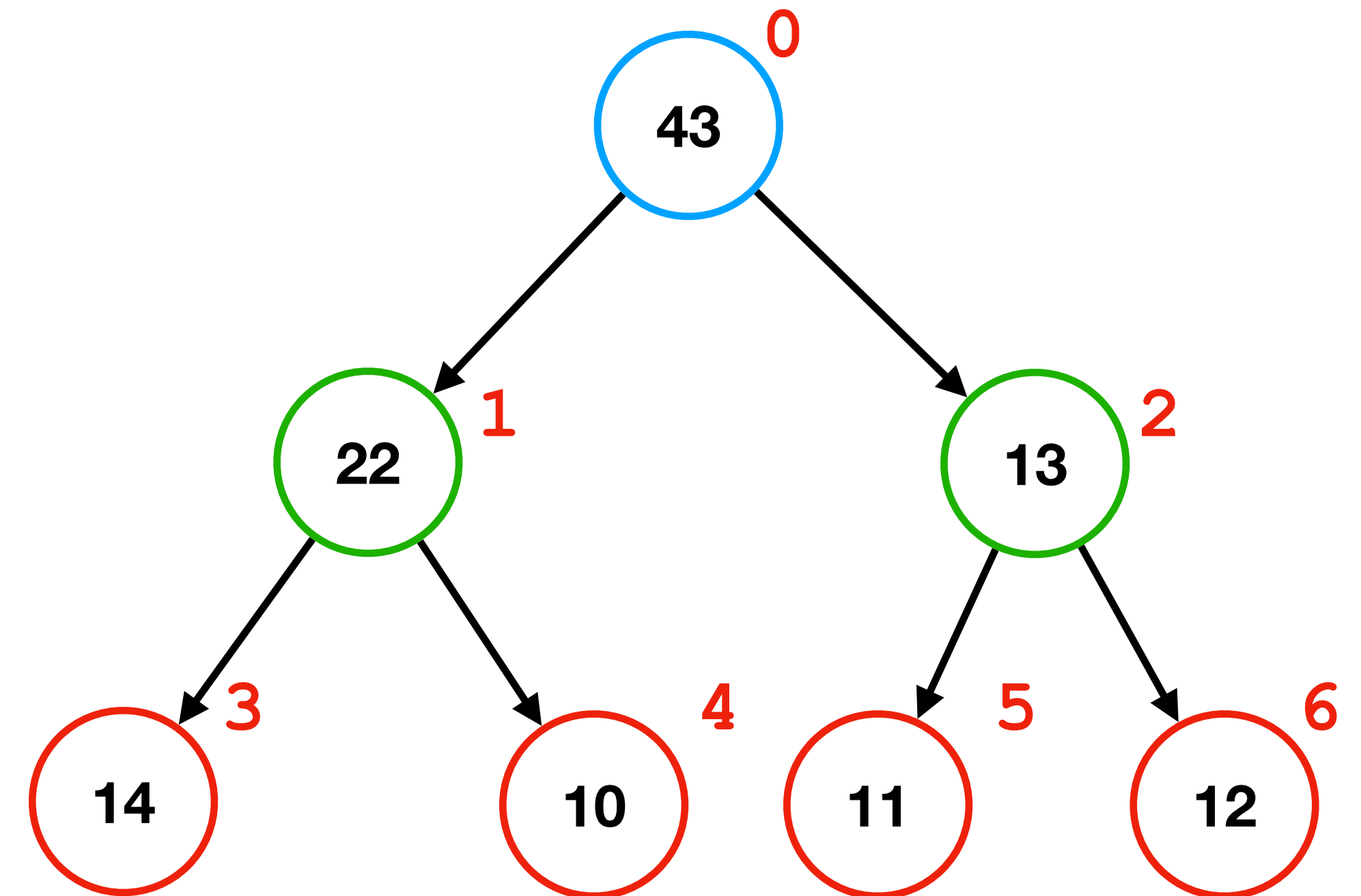
- CMSC 27200.
- Informal analysis:
 - Define level to be the maximum distance from the leaves.
 - If n is `heap_size`, there are at most...
 - $n/2^{l+1}$ nodes for level l



Heap

Heapify Complexity?

- CMSC 27200.
- Informal analysis:
 - Define level to be the maximum distance from the leaves.
 - If n is `heap_size`, there are at most...
 - $n/2^{l+1}$ nodes for level l
 - For nodes with level l , there can be at most l swaps bubbling down.

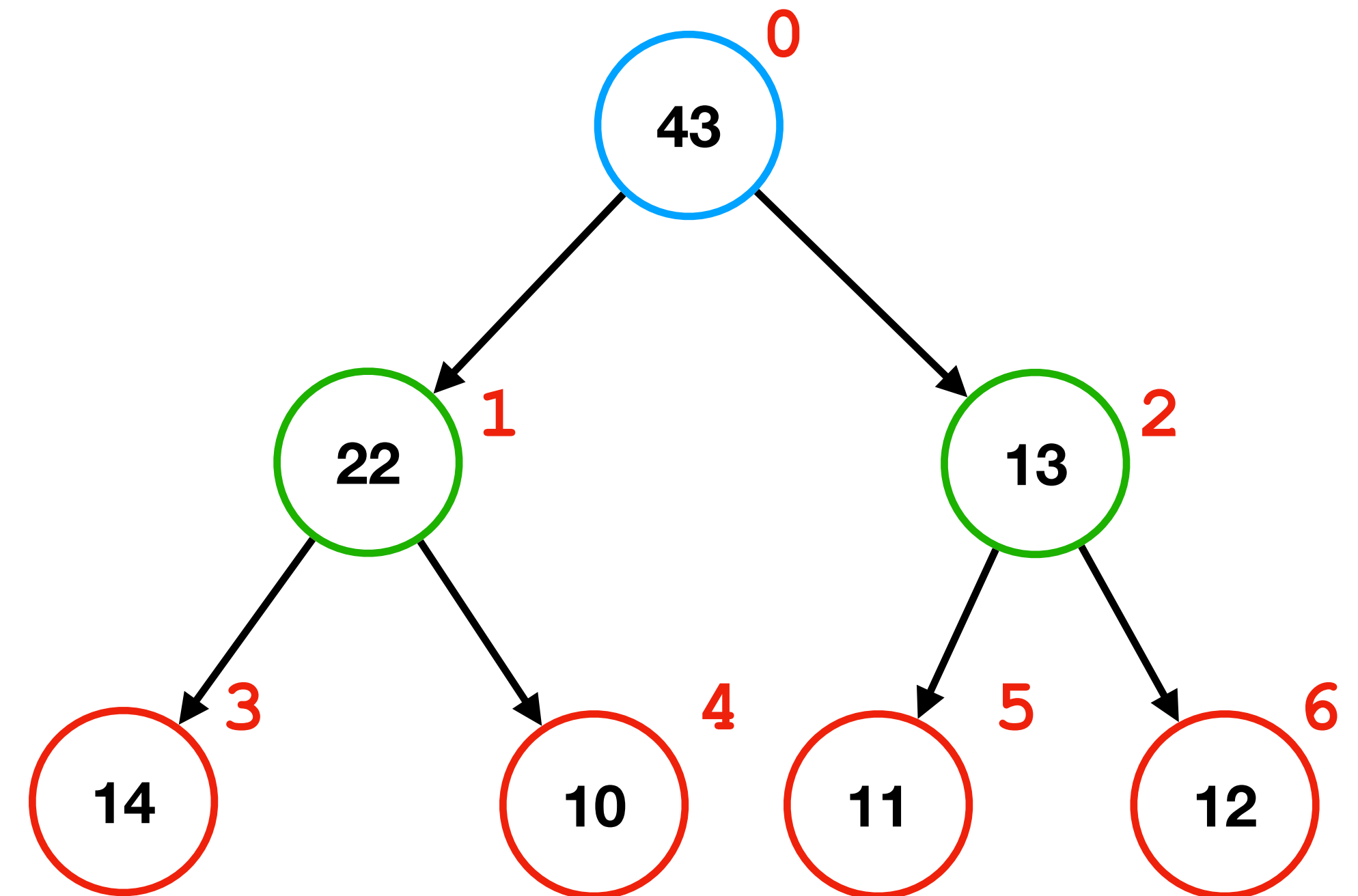


Heap

Heapify Complexity?

- CMSC 27200.
- Informal analysis:

- Total works: $\sum_{l=0}^{\log n} \frac{n}{2^{l+1}} \cdot l$
- $n \left(\sum_{l=0}^{\log n} \frac{l}{2^{l+1}} \right)$
- $\sum_{l=0}^{\log n} \frac{l}{2^{l+1}} < \sum_{l=0}^{\infty} \frac{l}{2^{l+1}} = 1$

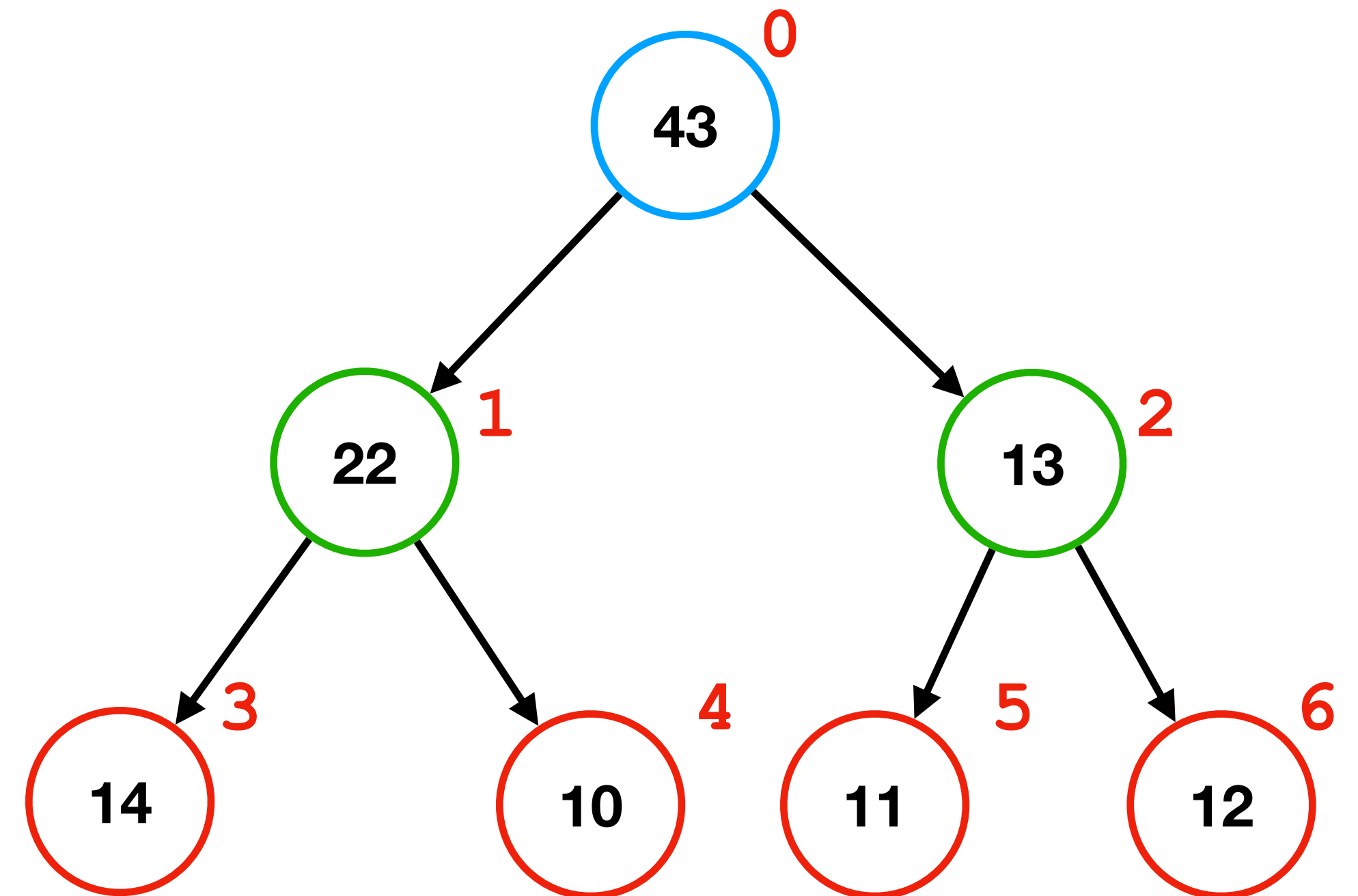


Heap

Heapify Complexity?

- CMSC 27200.
- Informal analysis:

- Total works: $\sum_{l=0}^{\log n} \frac{n}{2^{l+1}} \cdot l < n = O(n)$



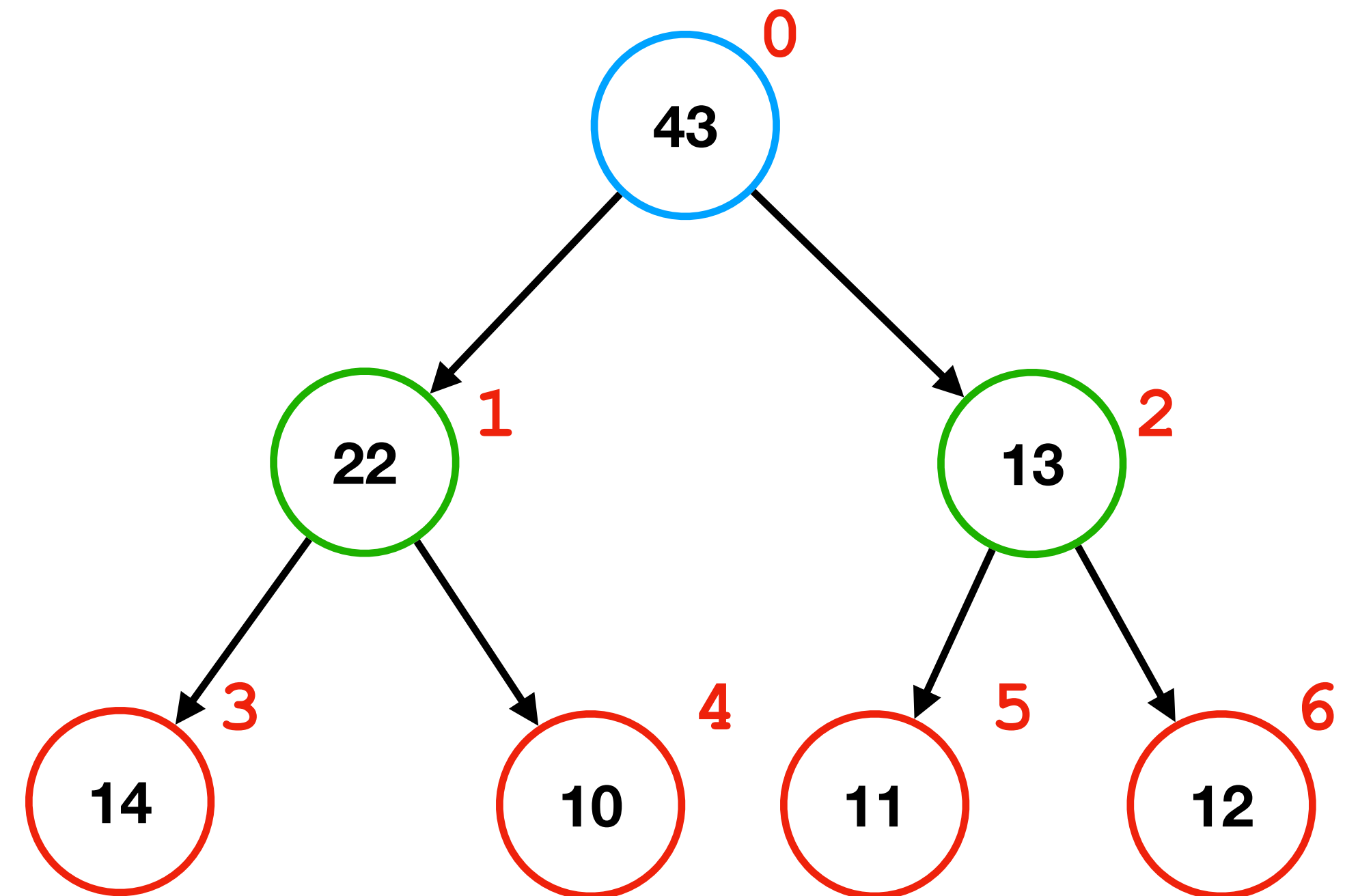
Heap

Heapify Complexity?

- CMSC 27200.
- Informal analysis:

- Total works: $\sum_{l=0}^{\log n} \frac{n}{2^{l+1}} \cdot l < n = O(n)$

- Magic!



Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

- Heapify -- $O(n)$
- `remove_top` and put it to the end

Heap

Heap Sort

```
int remove_top(int *arr, int len)
{
    int top = arr[0];
    arr[0] = arr[len - 1];

    bubble_down(arr, len - 1, 0);

    return top;
}
```

```
for (int i = len - 1; i >= 1; i--) {
    swap(arr, 0, i);
    bubble_down(arr, i, 0);
}
```

Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

- Heapify -- $O(n)$
- For each n :
 - `remove_top` and put it to the end $O(\log n)$
- Complexity: $O(n \log n)$

Heap

Heap Sort Code

```
void heap_sort(int *arr, int len)
{
    for (int i = len / 2 - 1; i >= 0; i--) {
        bubble_down(arr, len, i);
    }

    for (int i = len - 1; i >= 1; i--) {
        swap(arr, 0, i);
        bubble_down(arr, i, 0);
    }
}
```

Heapify

Extract top and put at the end

Heap

Heap Sort

- What is the best way to build a heap from scratch?

12, 22, 11, 8, 10, 43, 13, 9, 14

- Heapify -- $O(n)$
- For each n :
 - `remove_top` and put it to the end $O(\log n)$
- Complexity: $O(n \log n)$
- Space complexity: in-place

Sorting

- $O(n^2)$: Selection, Insertion, Bubble
- $O(n \log n)$: Tree, Merge, Quick
- $O(n \log n)$ without extra space (not even a stack): Heap sort
 - Heap sort is "selection sort with the right data structure."

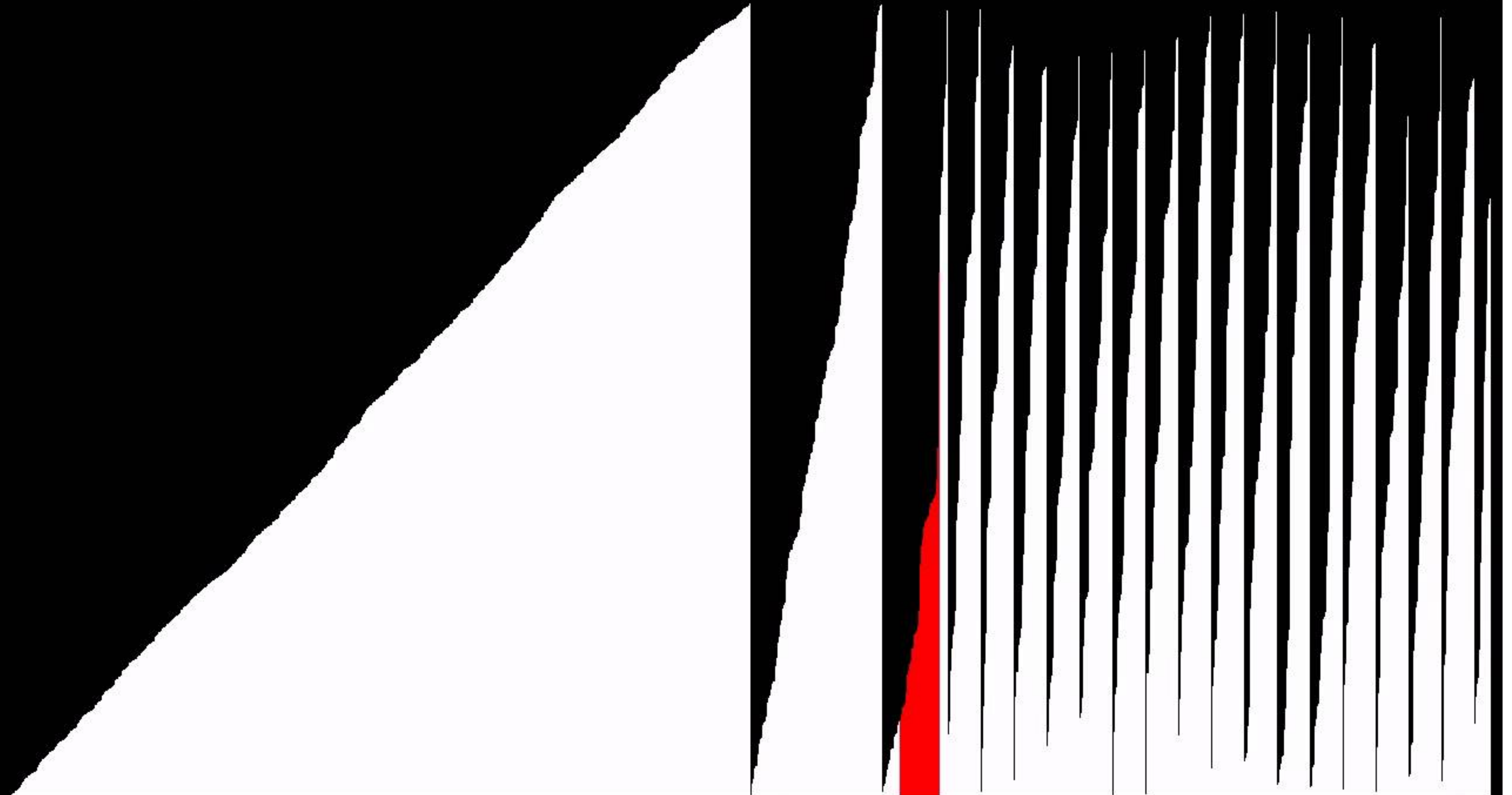
Sorting

Links

- Visualization of heaps: <https://www.cs.usfca.edu/~galles/visualization/Heap.html>
- Dances:
 - Merge sort: https://youtu.be/XaqR3G_NVoo
 - Quick sort: <https://youtu.be/ywWBy6J5gz8>
- Sorting algorithms: <https://youtu.be/kPRA0W1kECg>

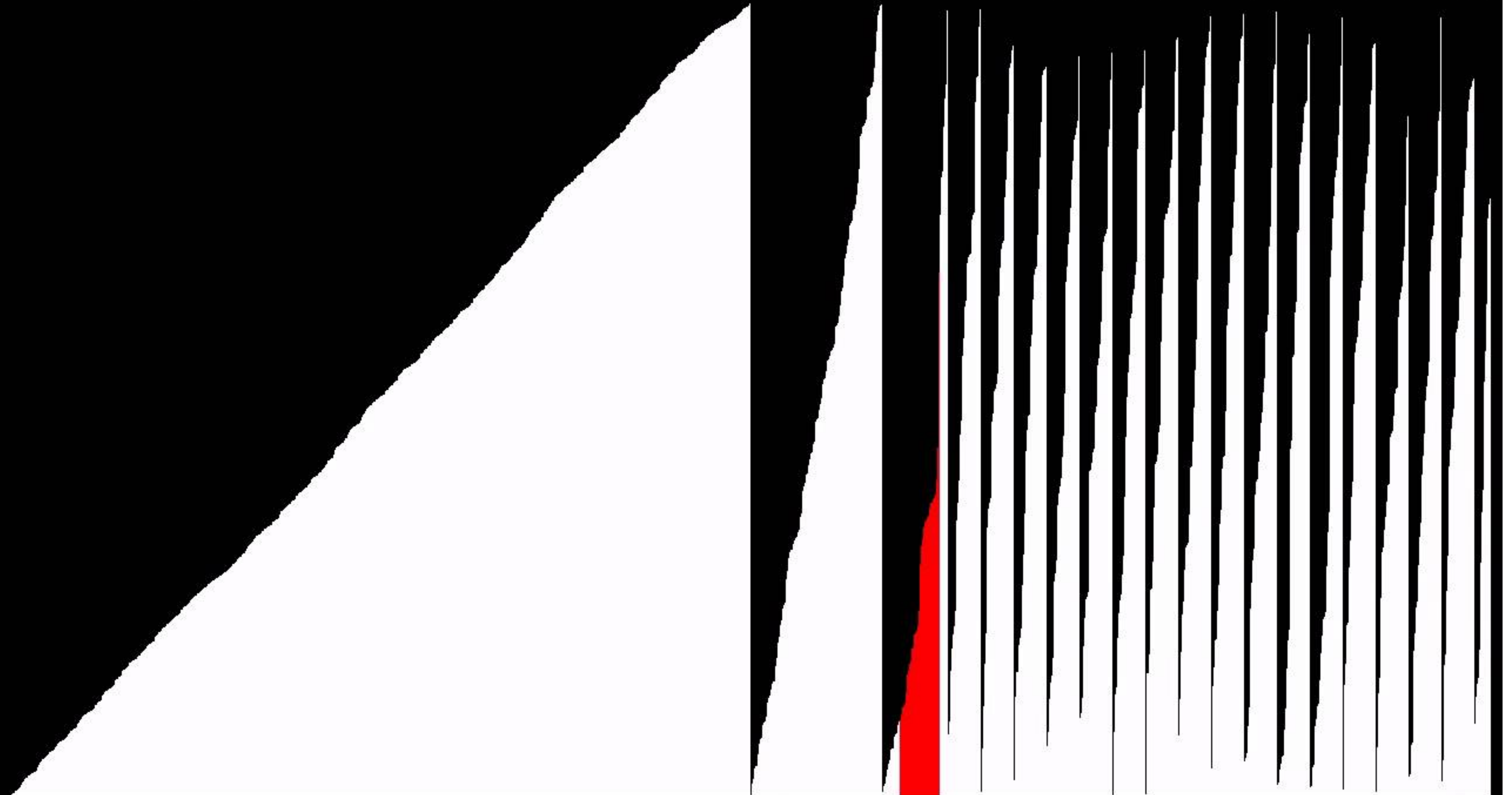
std::stable_sort (gcc) - 8950 comparisons, 20268 array accesses, 1.00 ms delay

<http://panthema.net/2013/sound-of-sorting>



std::stable_sort (gcc) - 8950 comparisons, 20268 array accesses, 1.00 ms delay

<http://panthema.net/2013/sound-of-sorting>



Sorting

Recap

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Four $O(n \log n)$ algorithms: Tree, Merge, Quick, Heap

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Four $O(n \log n)$ algorithms: Tree, Merge, Quick, Heap
- There are a lot more sorting algorithms...

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Bubble, Insertion
- Four $O(n \log n)$ algorithms: Timsort, Merge, Quick, Heap
- There are a lot more sorting algorithms

Name ↕	Best ↕	Average ↕	Worst ↕
Quicksort	$n \log n$	$n \log n$	n^2
Merge sort	$n \log n$	$n \log n$	$n \log n$
In-place merge sort	—	—	$n \log^2 n$
Introsort	$n \log n$	$n \log n$	$n \log n$
Heapsort	$n \log n$	$n \log n$	$n \log n$
Insertion sort	n	n^2	n^2
Block sort	n	$n \log n$	$n \log n$
Timsort	n	$n \log n$	$n \log n$
Selection sort	n^2	n^2	n^2
Cubesort	n	$n \log n$	$n \log n$
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$
Bubble sort	n	n^2	n^2
Exchange sort	n^2	n^2	n^2
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)
Cycle sort	n^2	n^2	n^2
Library sort	$n \log n$	$n \log n$	n^2
Patience sorting	n	$n \log n$	$n \log n$
Smoothsort	n	$n \log n$	$n \log n$
Strand sort	n	n^2	n^2
Tournament sort	$n \log n$	$n \log n$	$n \log n$
Cocktail shaker	n	n^2	n^2

From Wikipedia, the free encyclopedia

In [computer science](#), **bogosort**^{[1][2]} (also known as **permutation sort** and **stupid sort**^[3]) is a [sorting algorithm](#) based on the [generate and test](#) paradigm. The function successively generates [permutations](#) of its input until it finds one that is sorted. It is not considered useful for sorting, but may be used for educational purposes, to contrast it with more efficient algorithms.

Two versions of this algorithm exist: a deterministic version that enumerates all permutations until it hits a sorted one,^{[2][4]} and a [randomized](#) version that randomly permutes its input. An analogy for the working of the latter version is to sort a [deck of cards](#) by throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted. In a worst-case scenario with this version, the random source is of low quality and happens to make the sorted permutation unboundedly unlikely to occur. The algorithm's name is a [portmanteau](#) of the words *bogus* and *sort*.^[5]

Bogosort

Class	Sorting
Data structure	Array
Worst-case performance	Unbounded (randomized version), $O(n \times n!)$ (deterministic version)
Best-case performance	$\Omega(n)$ ^[1]
Average performance	$\Theta(n \times n!)$ ^[1]
Worst-case space complexity	$O(1)$

Description of the algorithm [\[edit \]](#)

Pseudocode [\[edit \]](#)

The following is a description of the randomized algorithm in [pseudocode](#):

```
while not sorted(deck):
    shuffle(deck)
```


From Wikipedia, the free encyclopedia

In [computer science](#), **bogosort**^{[1][2]} (also known as **permutation sort** and **stupid sort**^[3]) is a [sorting algorithm](#) based on the [generate and test](#) paradigm. The function successively generates [permutations](#) of its input until it finds one that is sorted. It is not considered useful for sorting, but may be used for educational purposes, to contrast it with more efficient algorithms.

Two versions of this algorithm exist: a deterministic version that enumerates all permutations until it hits a sorted one,^{[2][4]} and a [randomized](#) version that randomly permutes its input. An analogy for the working of the latter version is to sort a deck of cards by throwing the deck into the air, picking the cards up at random, and repeating the process until the deck is sorted. In a worst-case scenario with this version, the random source is of low quality and happens to make the sorted permutation unboundedly unlikely to occur. The algorithm's name is a [portmanteau](#) of the words *bogus* and *sort*.^[5]

Bogosort

Class	Sorting
Data structure	Array
Worst-case performance	Unbounded (randomized version), $O(n \times n!)$ (deterministic version)
Best-case performance	$\Omega(n)$ ^[1]
Average performance	$\Theta(n \times n!)$ ^[1]
Worst-case space complexity	$O(1)$

Description of the algorithm [[edit](#)]

Pseudocode [[edit](#)]

The following is a description of the randomized algorithm in [pseudocode](#):

```
while not sorted(deck):
    shuffle(deck)
```

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Four $O(n \log n)$ algorithms: Tree, Merge, Quick, Heap

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Four $O(n \log n)$ algorithms: Tree, Merge, Quick, Heap
- There are a lot more sorting algorithms...

Sorting

Recap

- Three $O(n^2)$ algorithms: Selection, Insertion, Bubble
- Four $O(n \log n)$ algorithms: Tree, Merge, Quick, Heap
- There are a lot more sorting algorithms...
- ... we have time for one more weird one

Counting Sort

Counting Sort

- Count the occurrences of every number
- Output each number as many times as it occurs in the original list

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

[illegible]

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Input

4	8	4	2	9	9	6	2	9
---	---	---	---	---	---	---	---	---

Counts

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

Counting Sort

Output

--	--	--	--	--	--	--	--	--

Counts

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

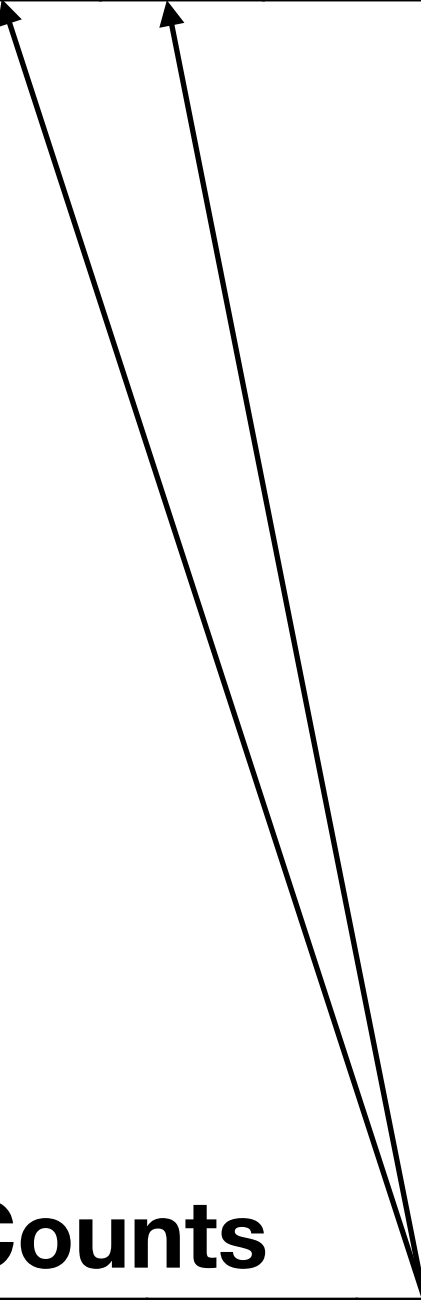
Counting Sort

Output

2	2							
---	---	--	--	--	--	--	--	--

Counts

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



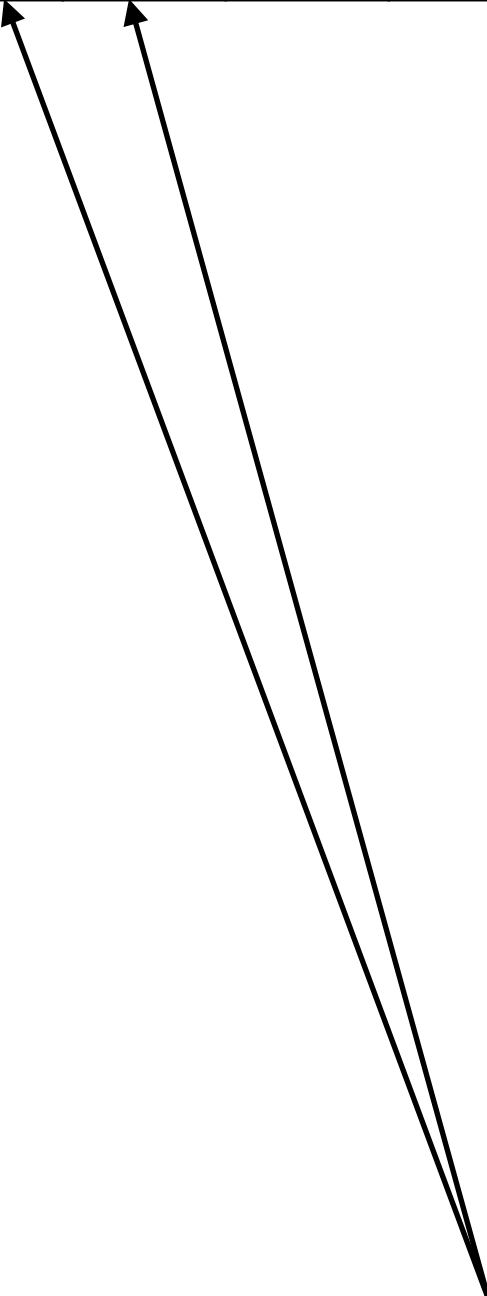
Counting Sort

Output

2	2	4	4					
---	---	---	---	--	--	--	--	--

Counts

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



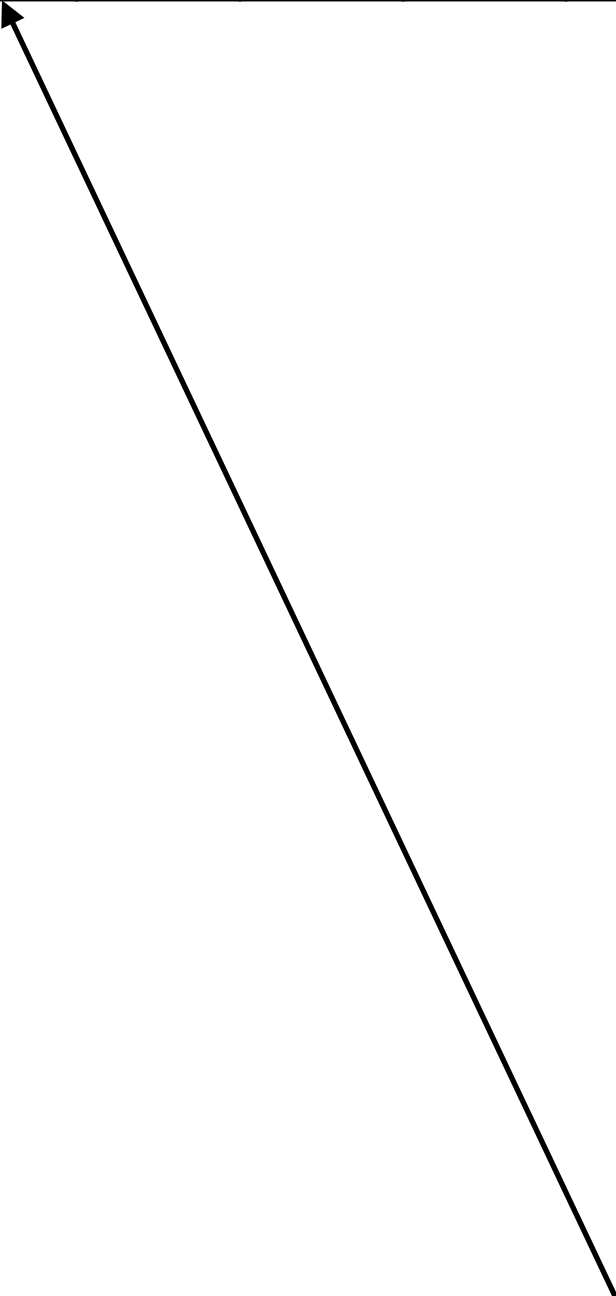
Counting Sort

Output

2	2	4	4	6				
---	---	---	---	---	--	--	--	--

Counts

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



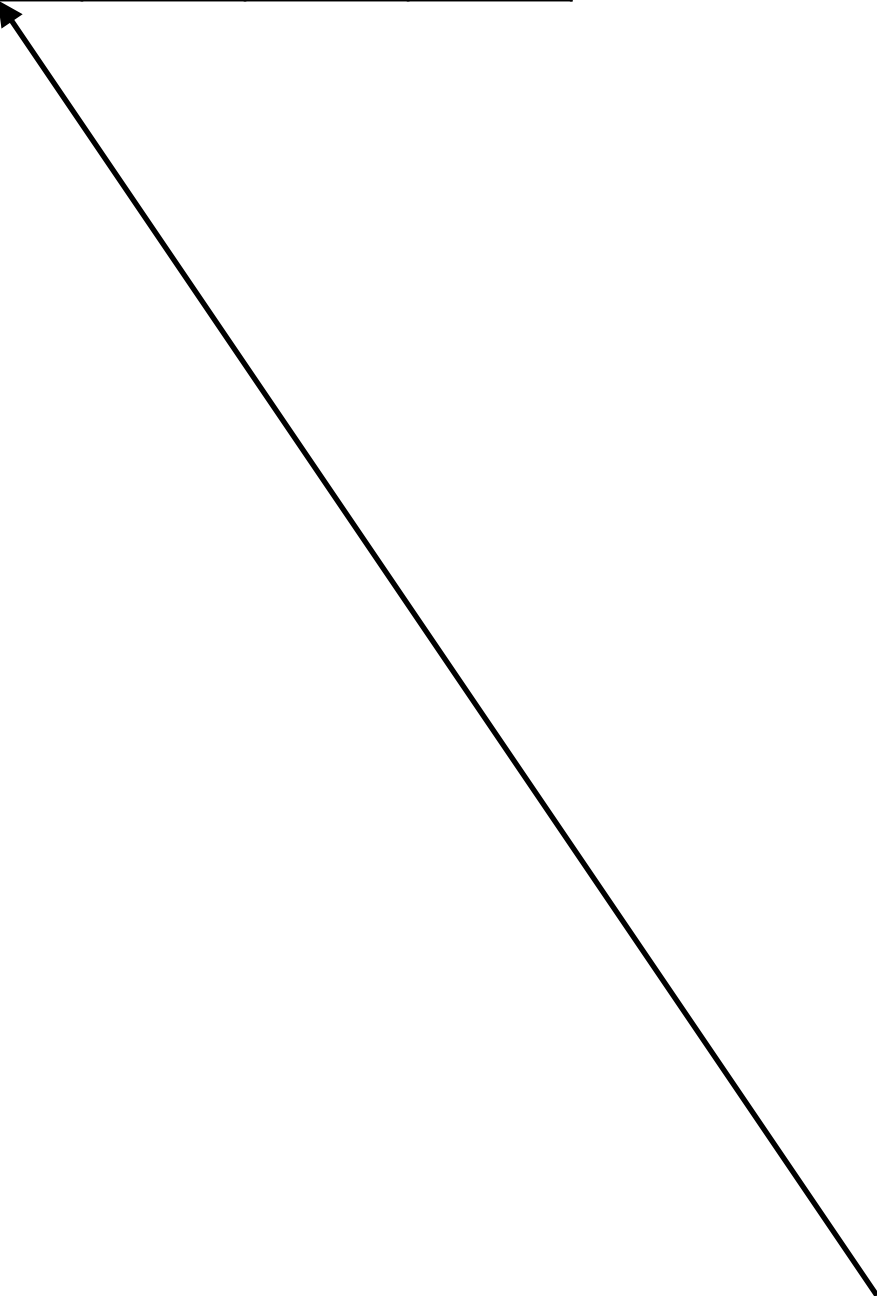
Counting Sort

Output

2	2	4	4	6	8			
---	---	---	---	---	---	--	--	--

Counts

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



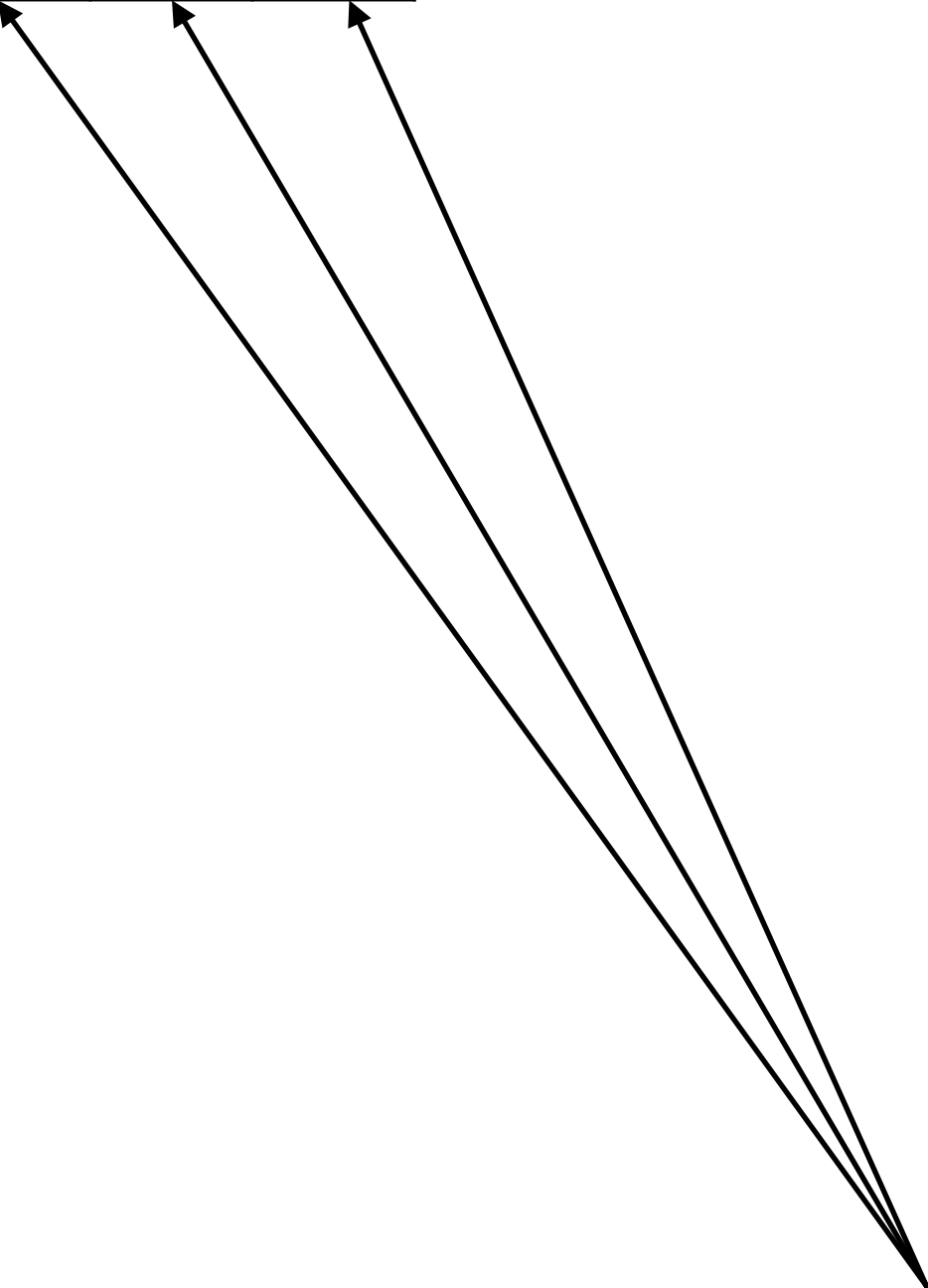
Counting Sort

Output

2	2	4	4	6	8	9	9	9
---	---	---	---	---	---	---	---	---

Counts

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



Counting Sort

Complexity

Counting Sort

Complexity

1. Find the range of values: $O(n)$

Counting Sort

Complexity

1. Find the range of values: $O(n)$
2. Initialize array: $O(n)$

Counting Sort

Complexity

1. Find the range of values: $O(n)$
2. Initialize array: $O(n)$
3. Scan the list to count: $O(n)$

Counting Sort

Complexity

1. Find the range of values: $O(n)$
2. Initialize array: $O(n)$
3. Scan the list to count: $O(n)$
4. Scan the counts to output: $O(n)$

Counting Sort

Complexity

1. Find the range of values: $O(n)$
2. Initialize array: $O(n)$
3. Scan the list to count: $O(n)$
4. Scan the counts to output: $O(n)$

Overall complexity: $O(n)$

Counting Sort

Limitations?

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:
 - Counts: $O(\text{Range})$ -- if the input is sparse, this can be a lot

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:
 - Counts: $O(\text{Range})$ -- if the input is sparse, this can be a lot
 - Output: $O(n)$

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:
 - Counts: $O(\text{Range})$ -- if the input is sparse, this can be a lot
 - Output: $O(n)$
- This is almost a Map!

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:
 - Counts: $O(\text{Range})$ -- if the input is sparse, this can be a lot
 - Output: $O(n)$
- This is almost a Map!
 - Key: Integer

Counting Sort

Limitations?

- Only apply to integers -- need to use the value as array indices
- Need extra space:
 - Counts: $O(\text{Range})$ -- if the input is sparse, this can be a lot
 - Output: $O(n)$
- This is almost a Map!
 - Key: Integer
 - Value: Counts

Counting Sort

Limitations?

Counting Sort

Limitations?

- Can we make this work with any value?

Counting Sort

Limitations?

- Can we make this work with any value?
 - Sure, instead of having an array of integers, we can have an array of whatever values

Counting Sort

Limitations?

- Can we make this work with any value?
 - Sure, instead of having an array of integers, we can have an array of whatever values
- Can we make this work with any key?

Counting Sort

Limitations?

- Can we make this work with any value?
 - Sure, instead of having an array of integers, we can have an array of whatever values
- Can we make this work with any key?
 - Turn any key into an integer

Counting Sort

Limitations?

- Can we make this work with any value?
 - Sure, instead of having an array of integers, we can have an array of whatever values
- Can we make this work with any key?
 - Turn any key into an integer
 - Make the range of the integer reasonable