Sorting III

CS143: lecture 15

Byron Zhong, July 16

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

- 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
  - Hospital triage
  - Covid vaccine eligibility

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

	`insert`	`get_top`	`remove_top`
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

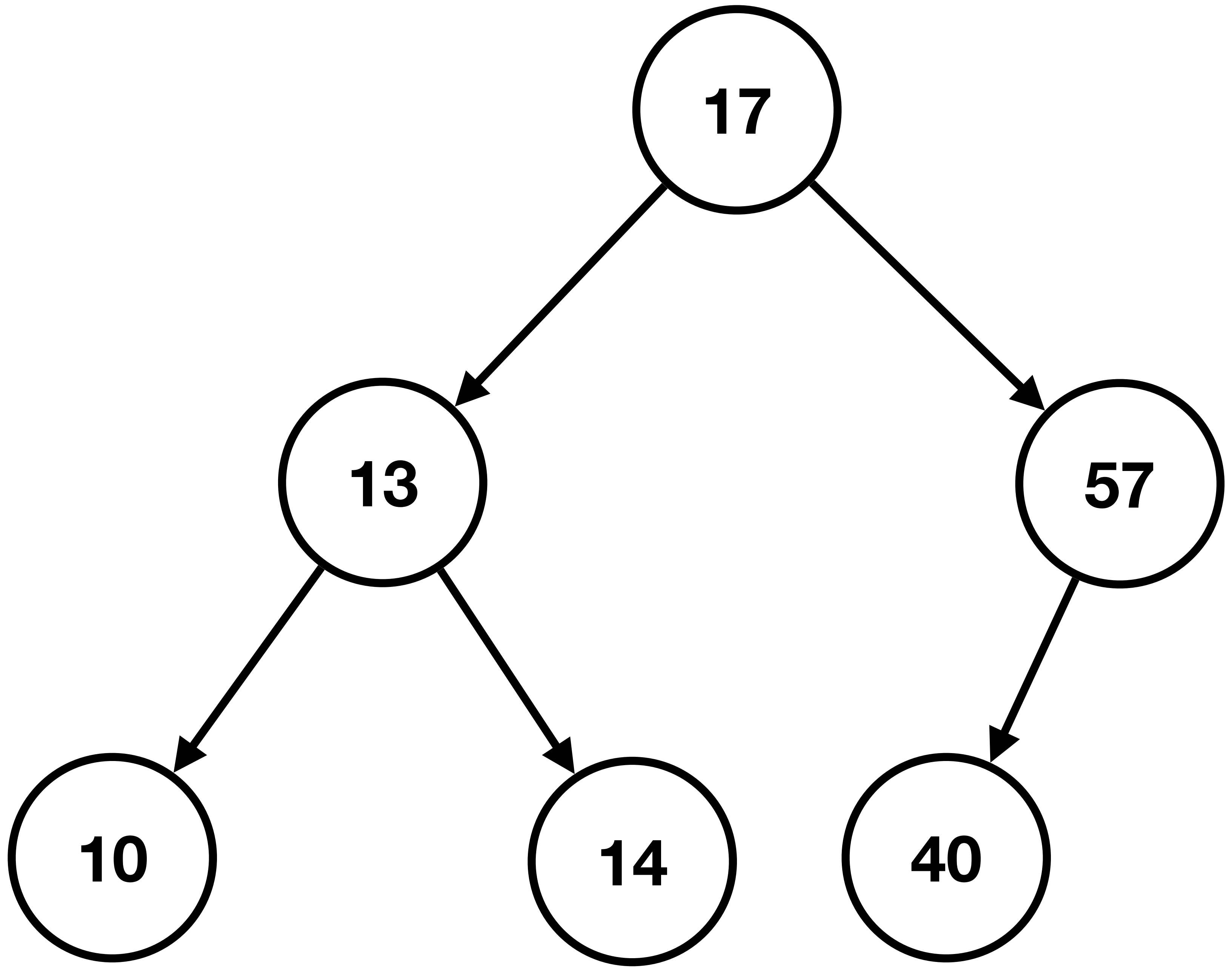
Heap

Shape property

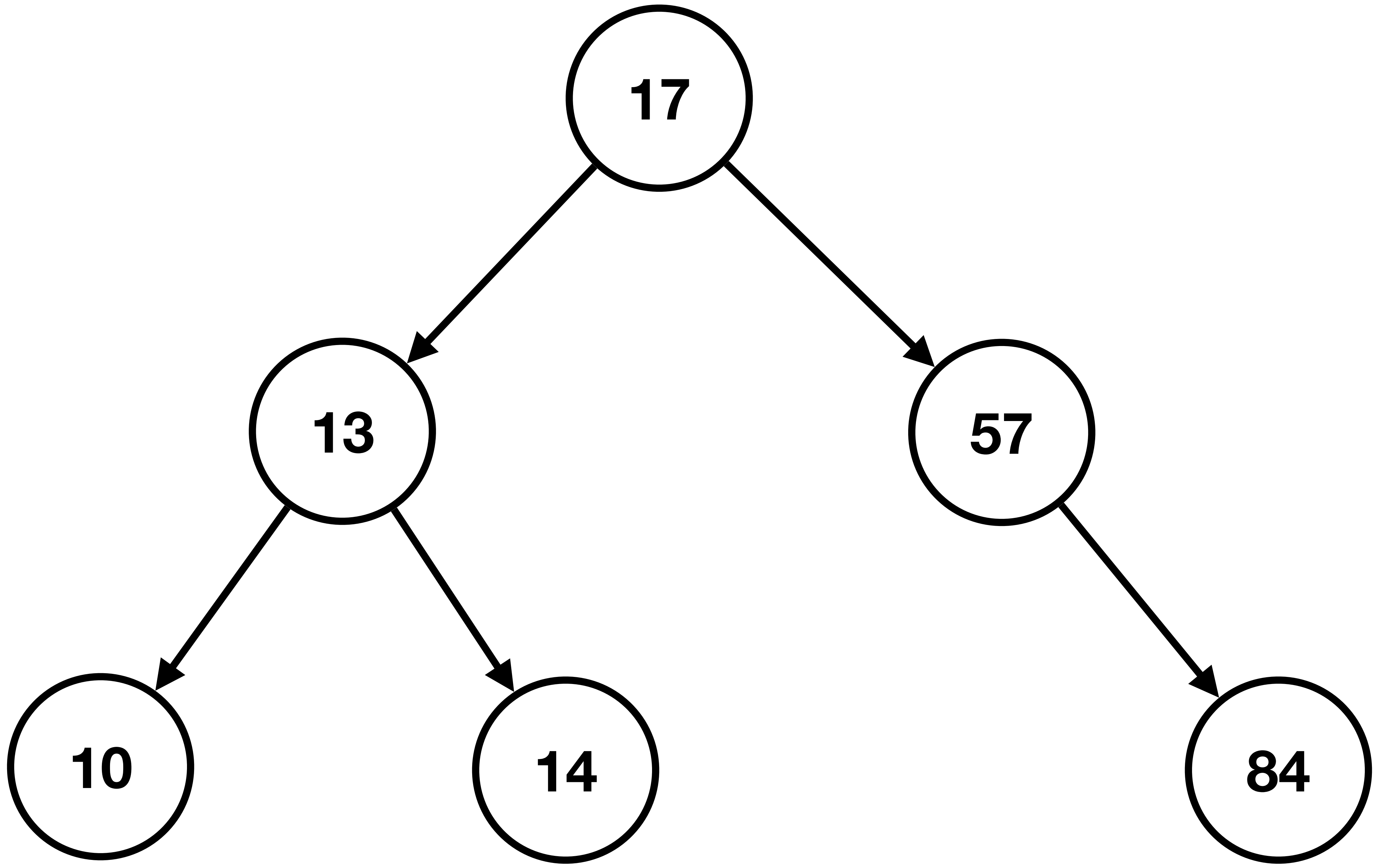
- A heap is a *complete* binary tree.
  - A binary tree is *complete* if and only if:
    - Each level is full, except possibly the last level;
    - The last level is filled from left to right

Heap

Shape property



Complete



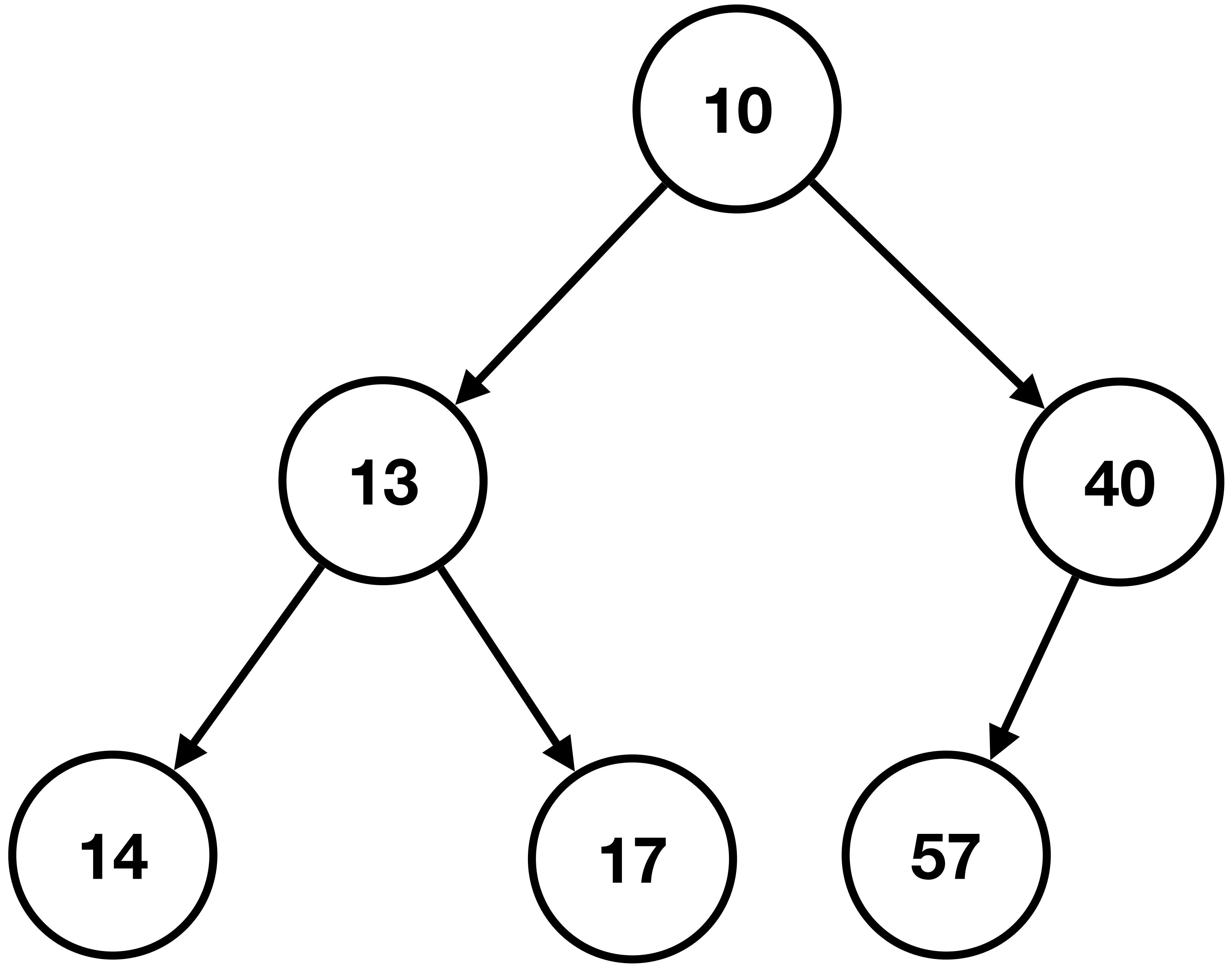
Incomplete

Heap

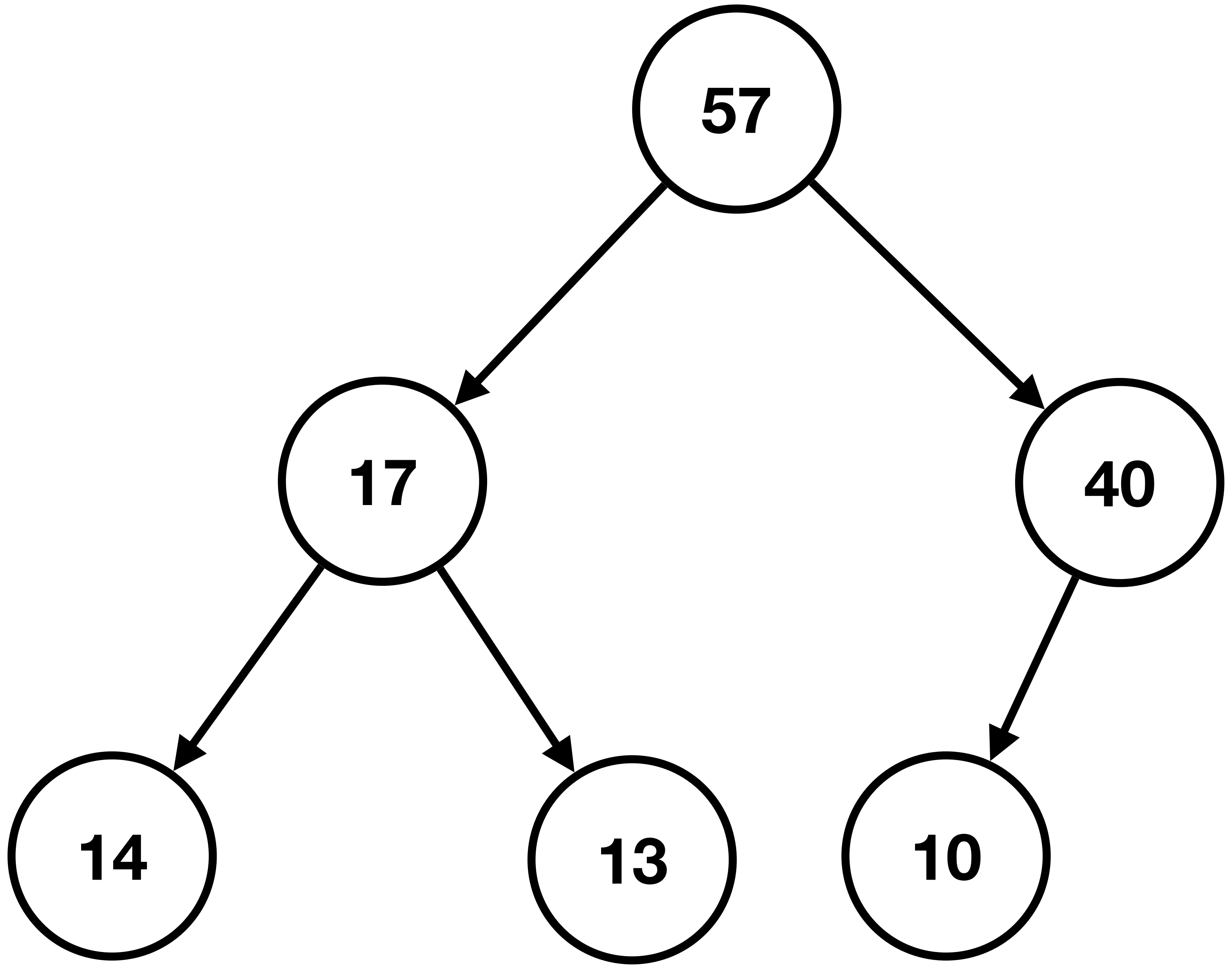
Value Property

- 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.
- <u>*This is not a BST.*</u>

Heap



Min-heap

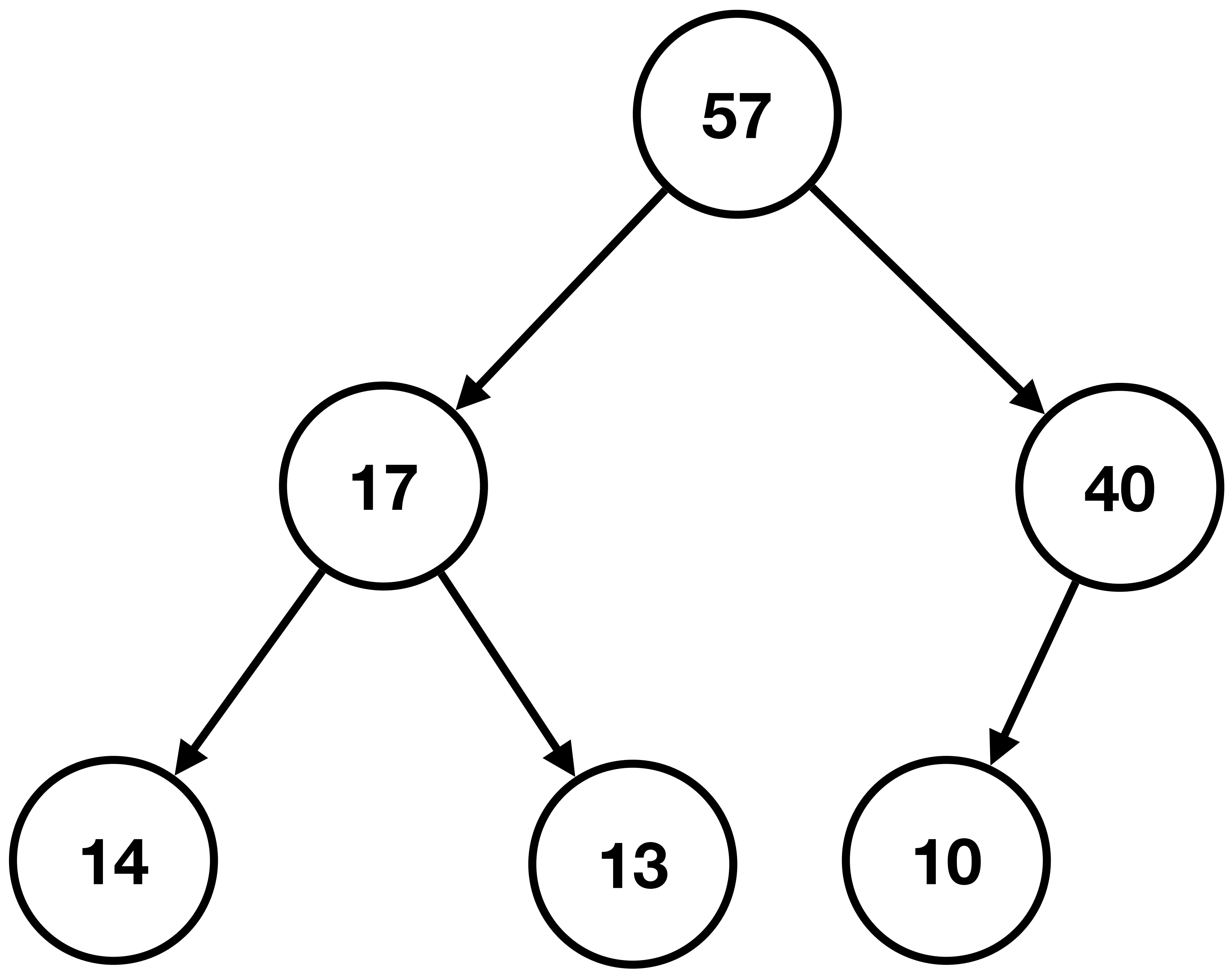


Max-heap

Heap

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

Heap

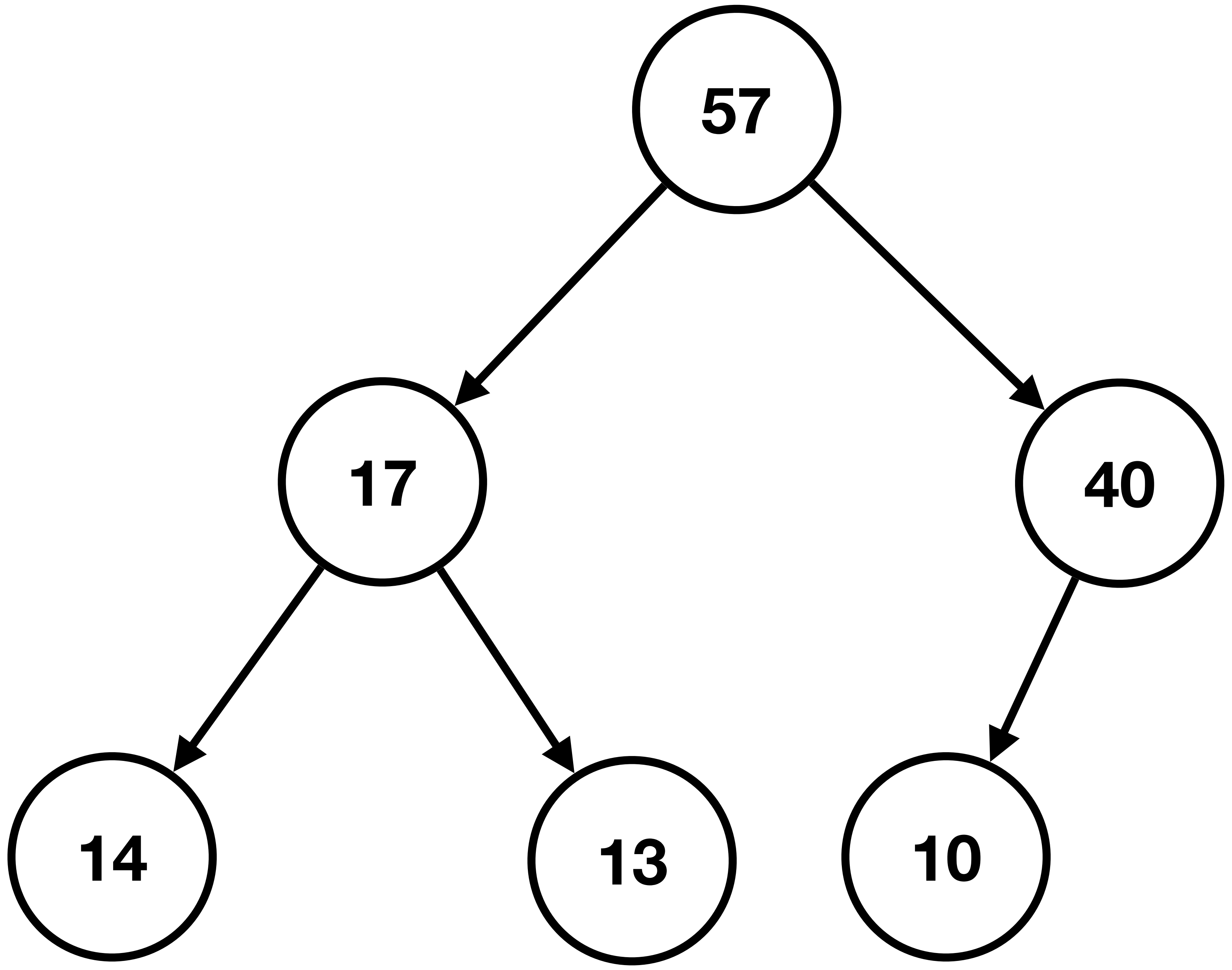


- Being <u>complete</u> means every 6-element heap has this shape.
- If a 7th element is added, the new trees will always have the same shape.

Heap

What is the best way to store this?

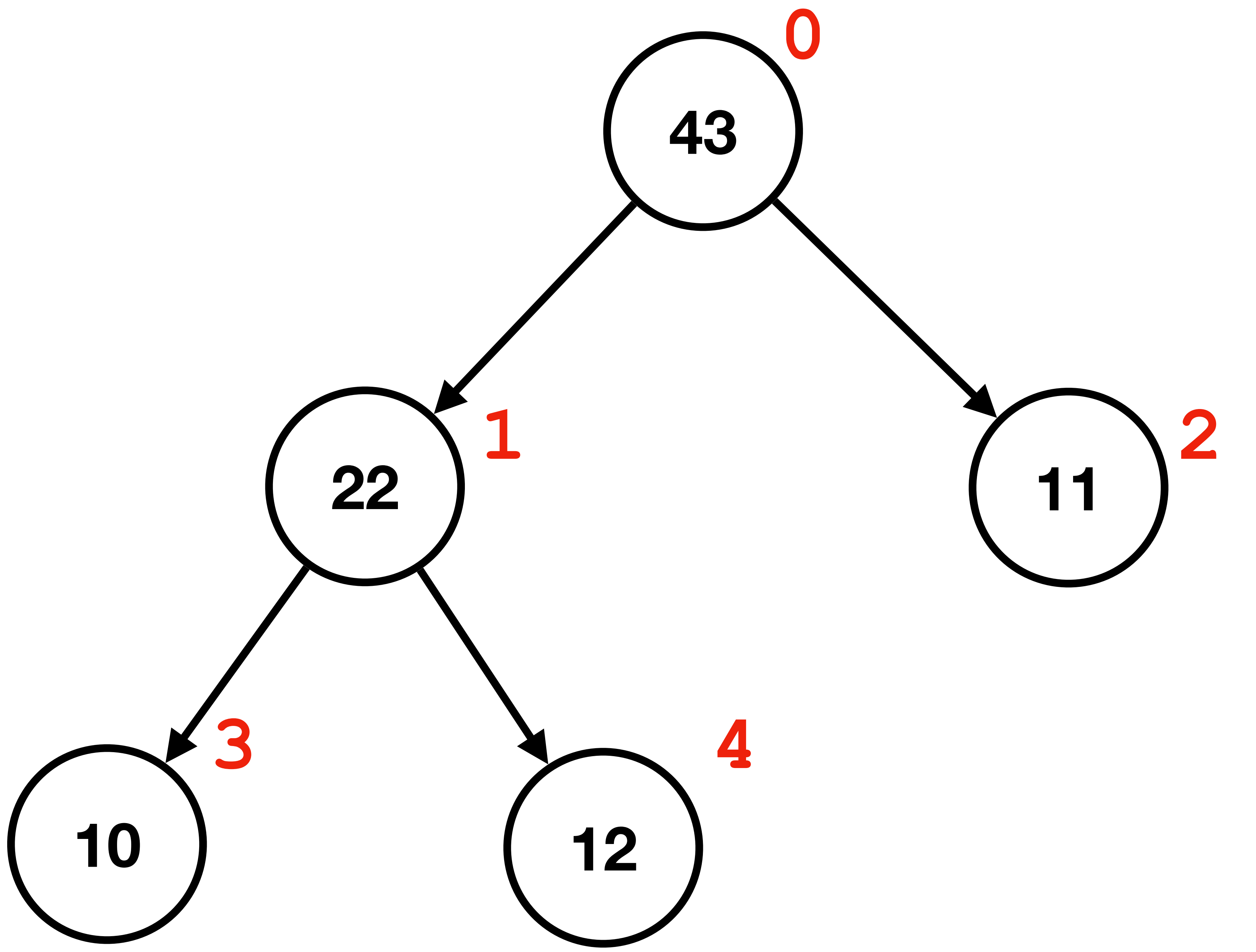
- Could use nodes and pointers...
- Or, we can use a data structure that provides constant-time access to elements:
- array



Heap

What is the best way to store this?

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

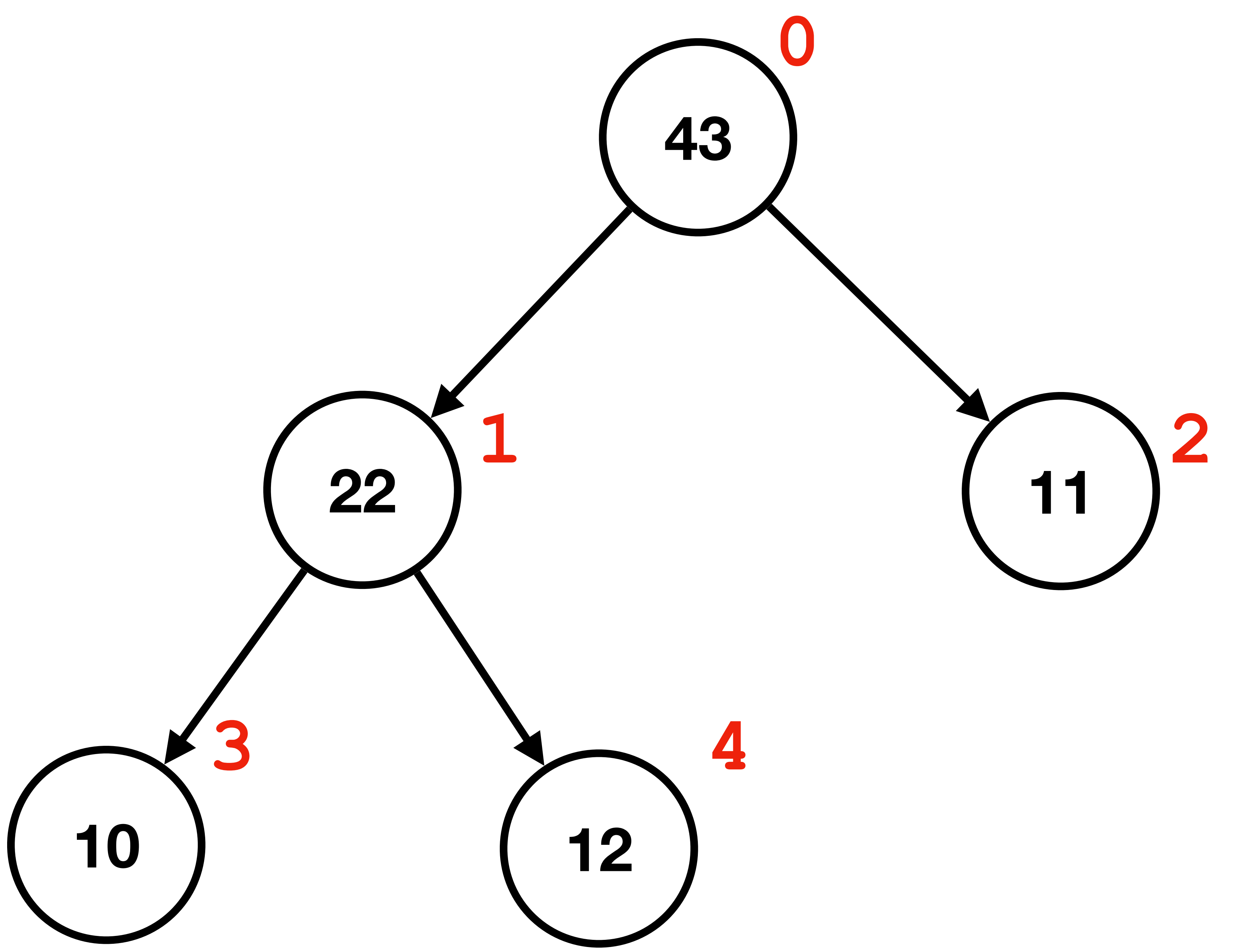


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

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: $\lfloor (i - 1)/2 \rfloor$



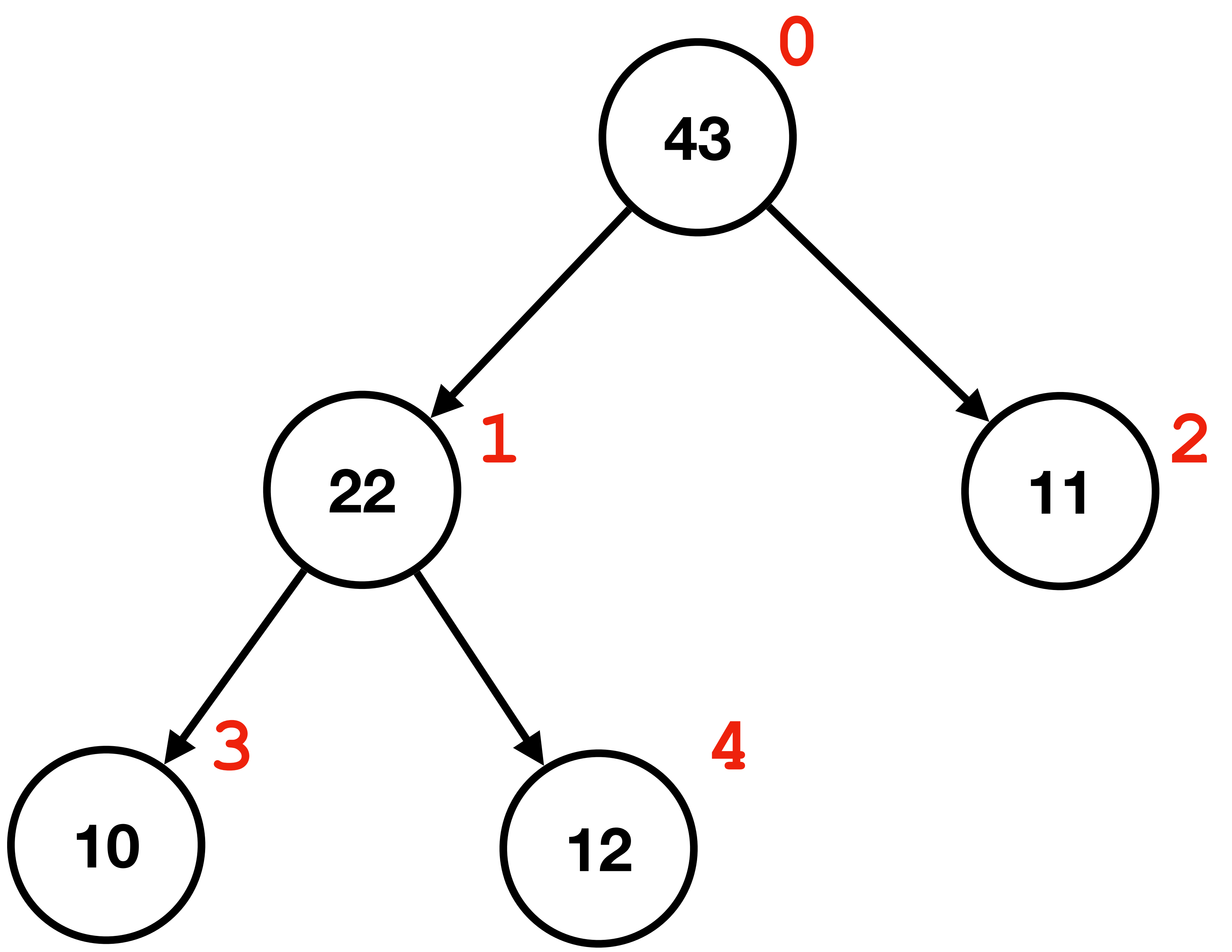
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

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];
```

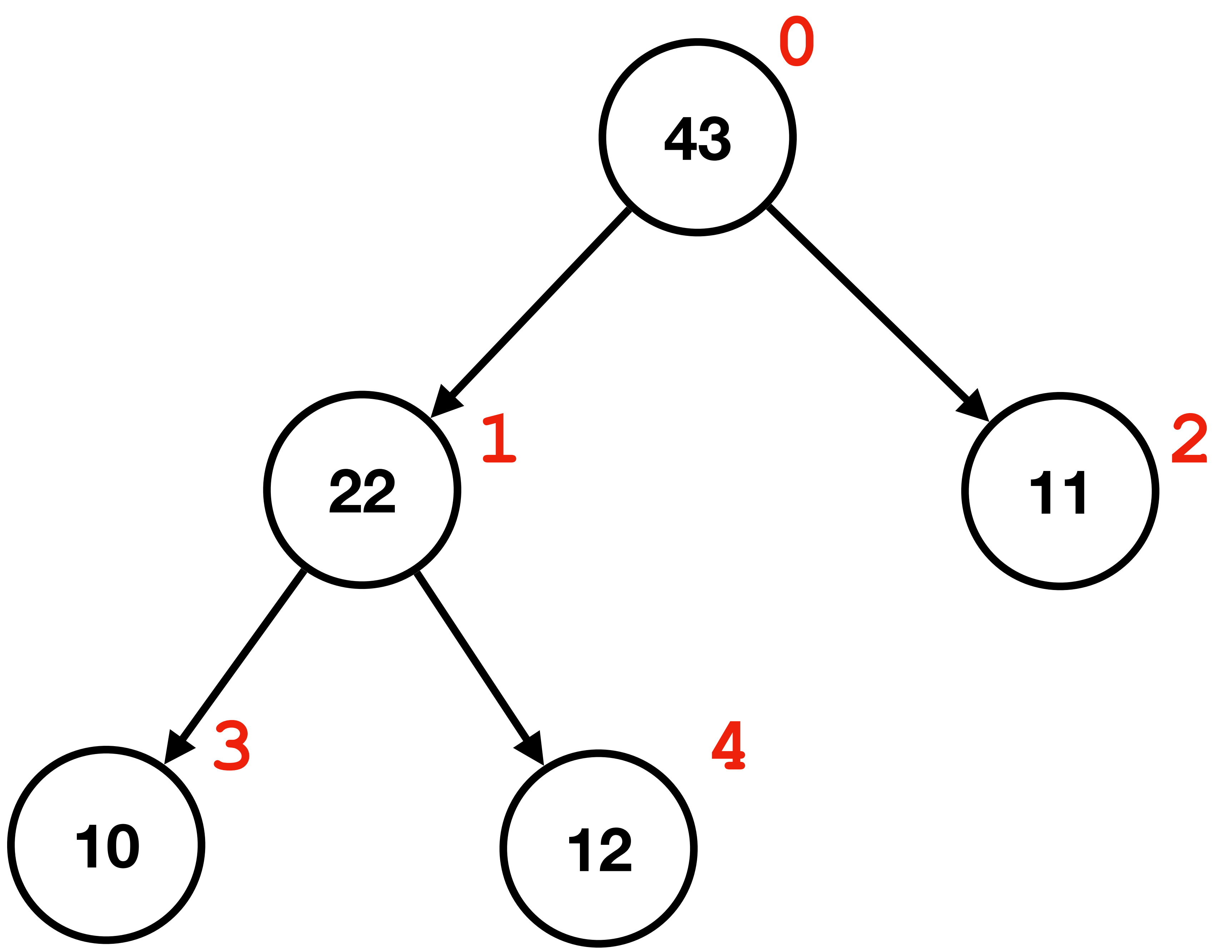


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

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

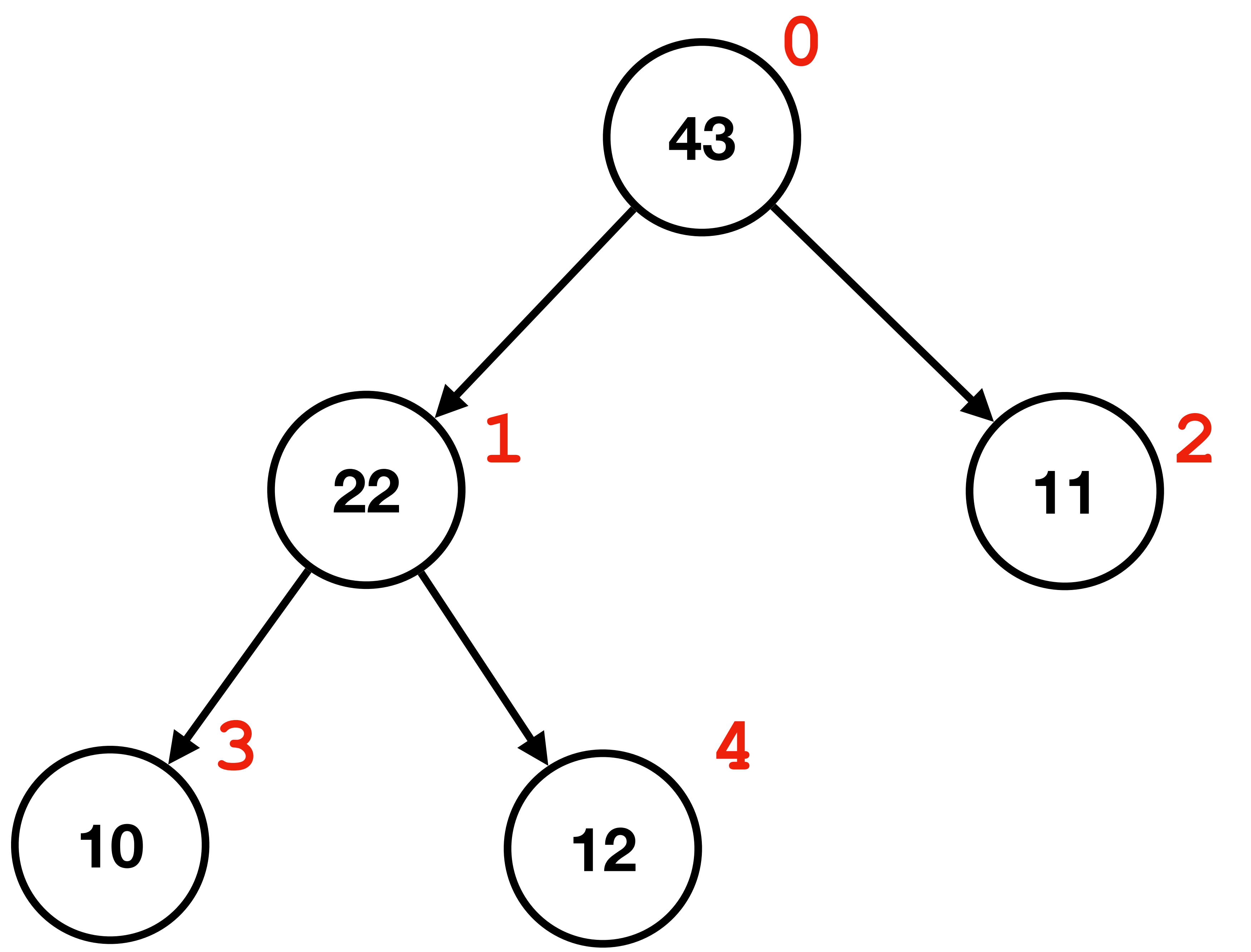


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

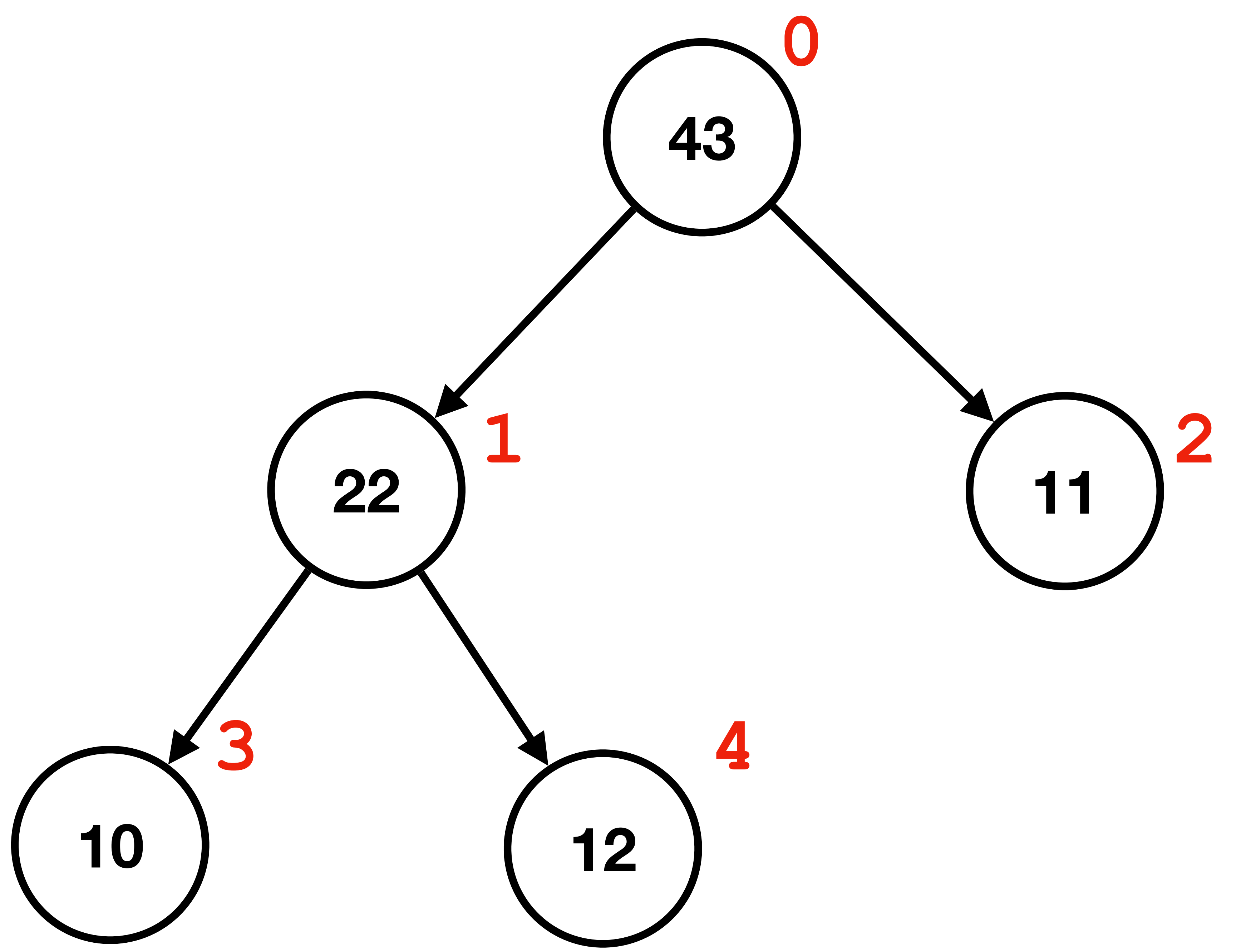
- `get_top()`
- $O(1)$ operation: `h[0]`
- How about insertion and removal?



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

Heap
Operations

- Insertion and removal scheme:
    1. Restore *<u>shape</u>* property first, ignoring value property
    2. Restore *<u>value</u>* property without changing the shape

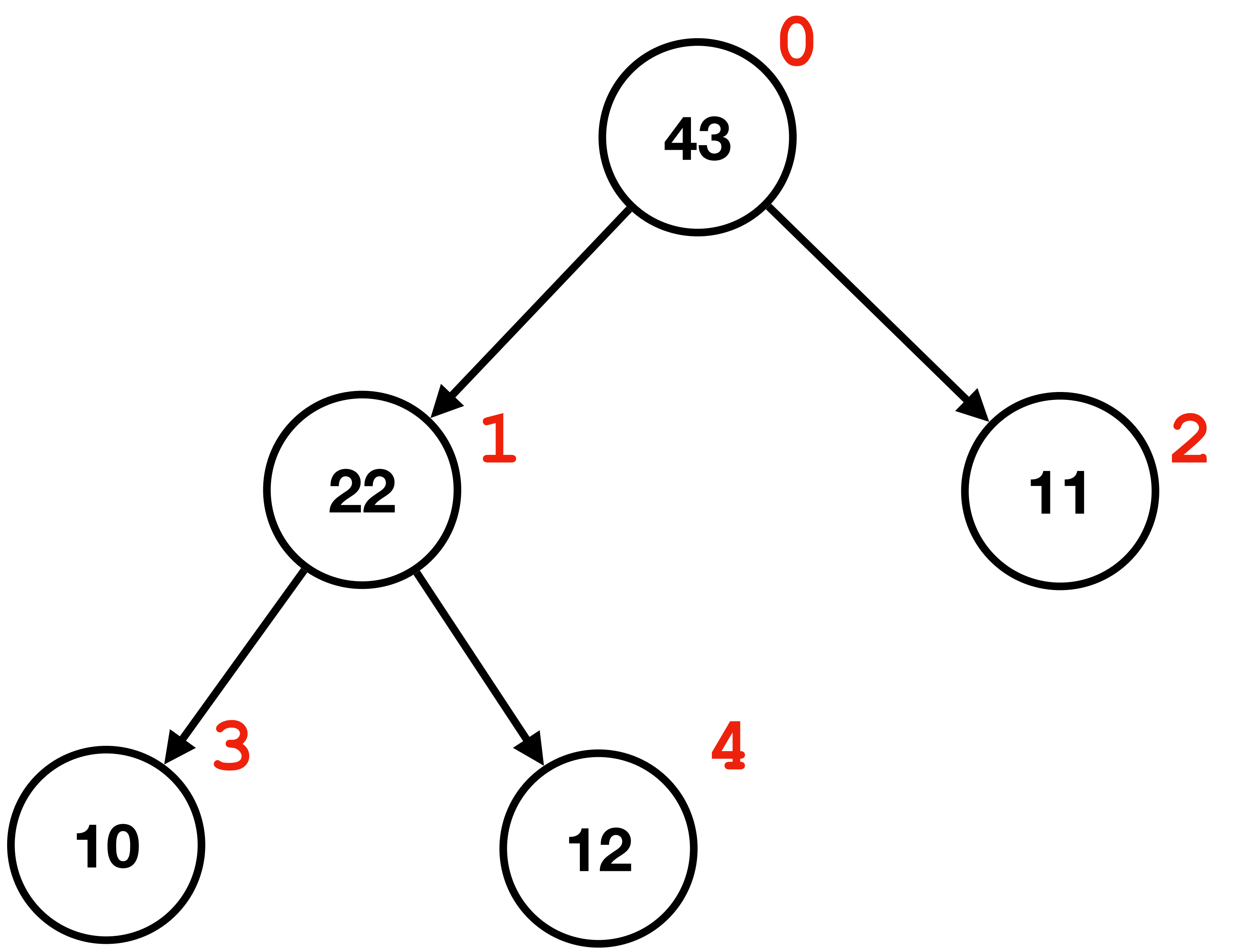


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

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



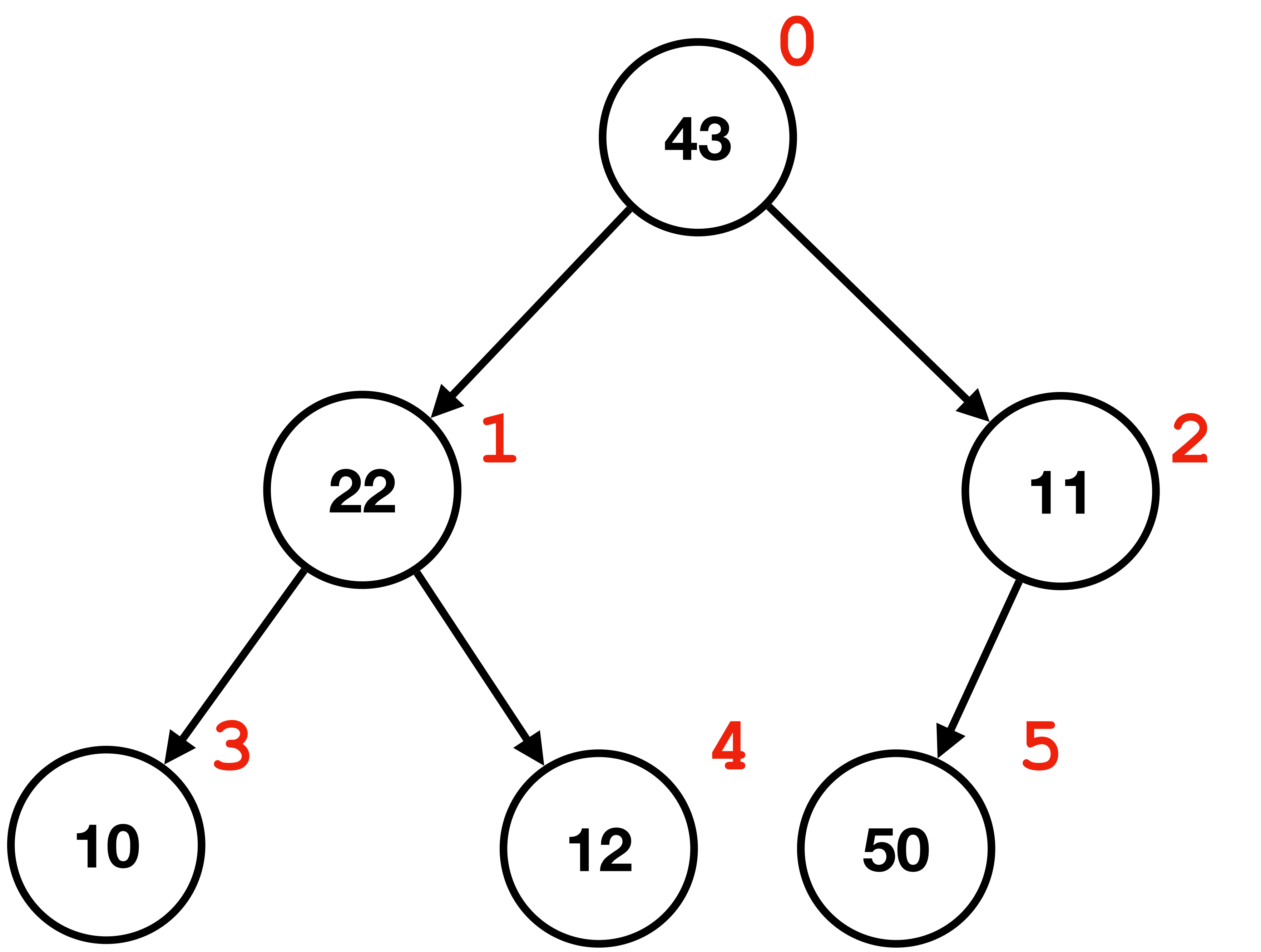
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

Heap

Operations

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

1. Insert item at element `h[heap_size + 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



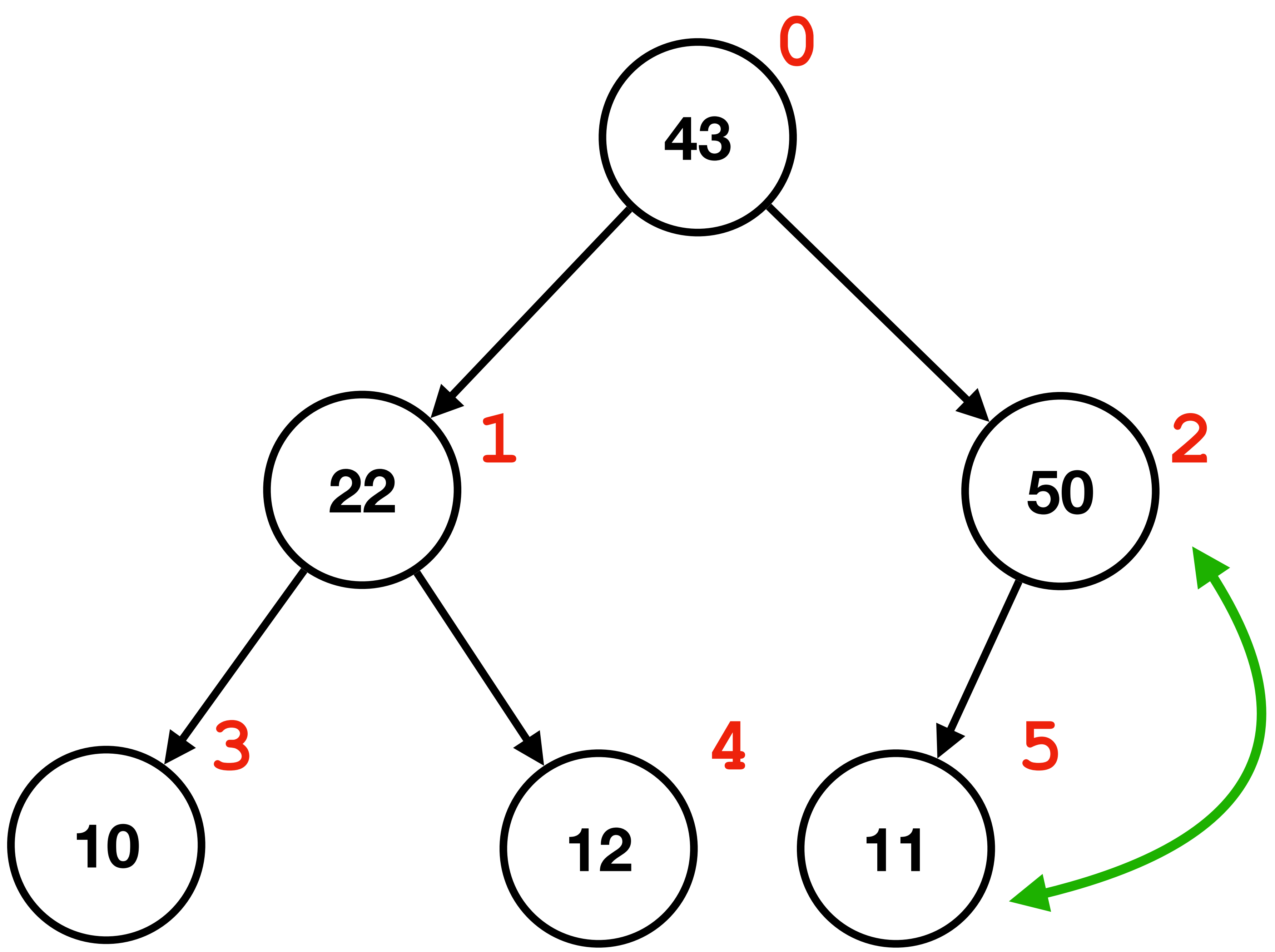
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12	50					

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



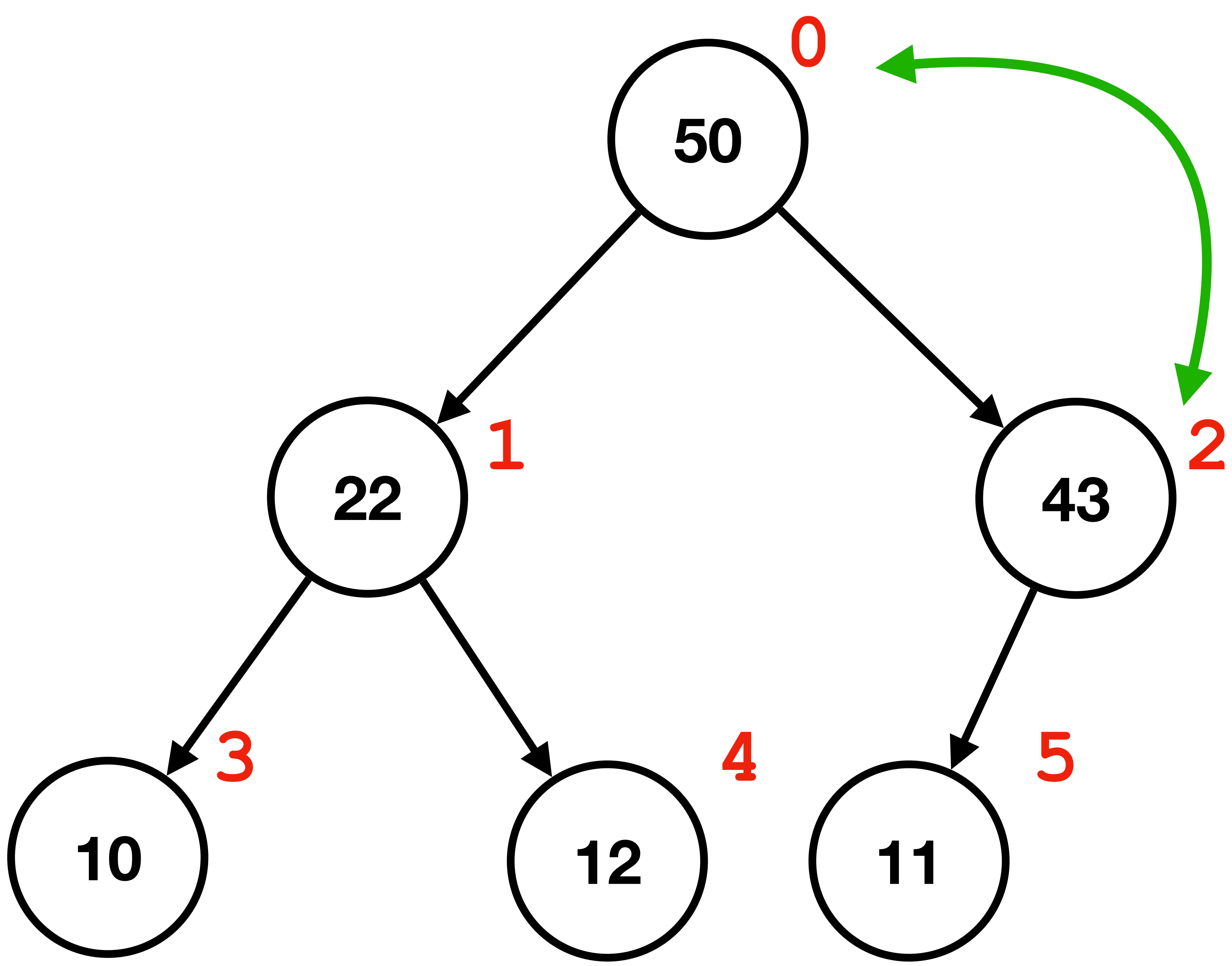
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	50	10	12	11					

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)`

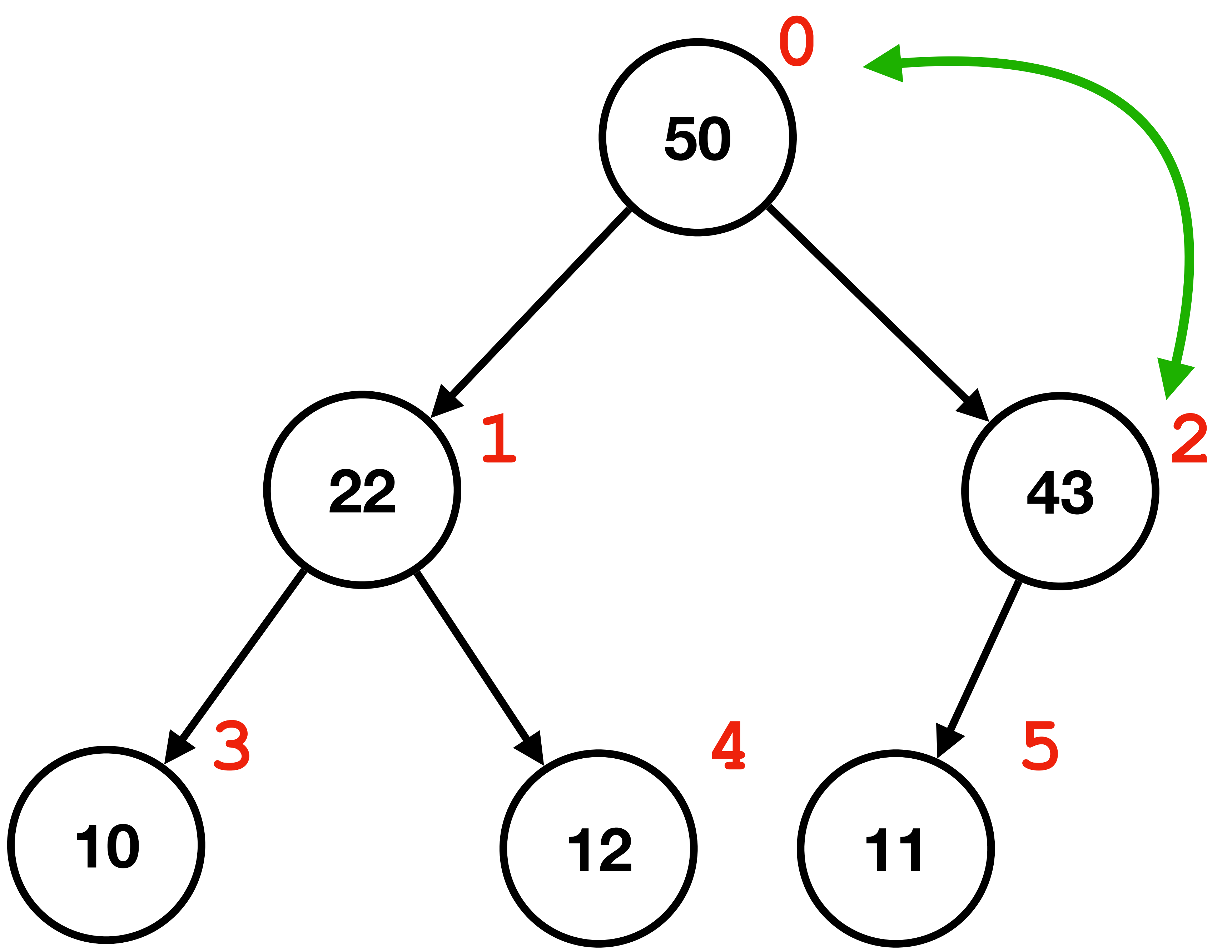
- Worst case complexity: $O(\log n)$

1. Insert item at element `h[heap_size + 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



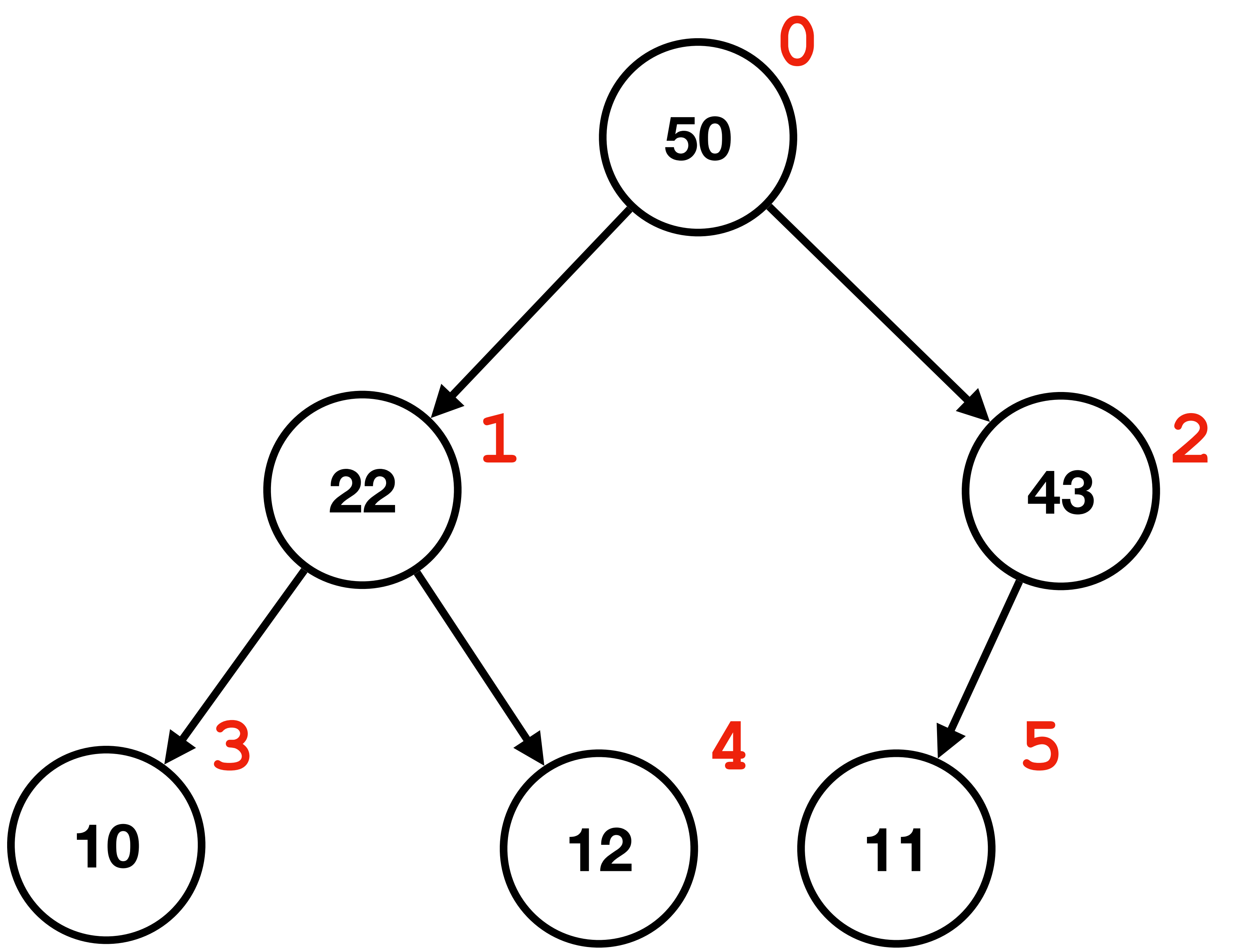
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
50	22	43	10	12	11					

Heap

Operations

- `remove_top()`

1. Remove root (save for later return)

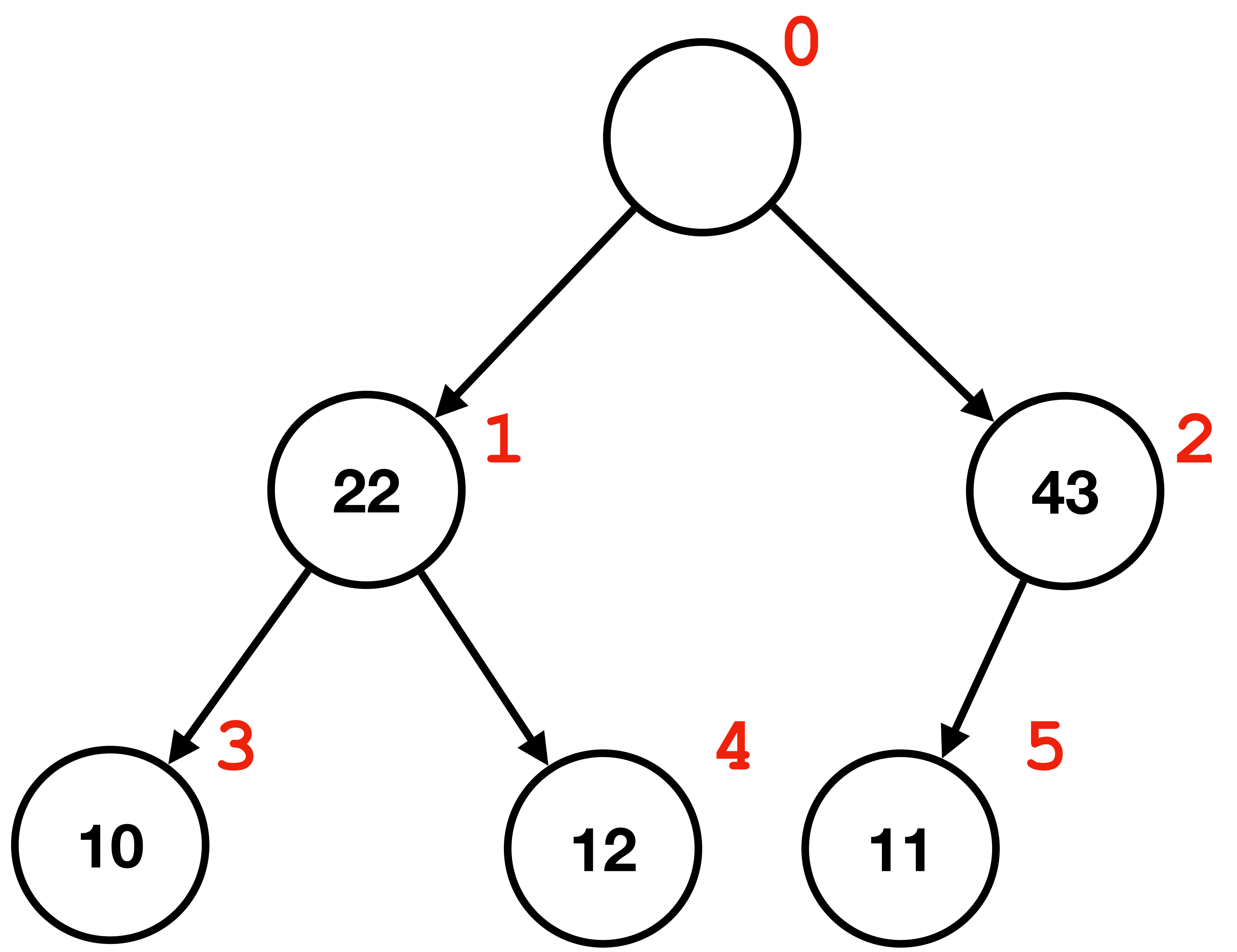


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
50	22	43	10	12	11					

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					

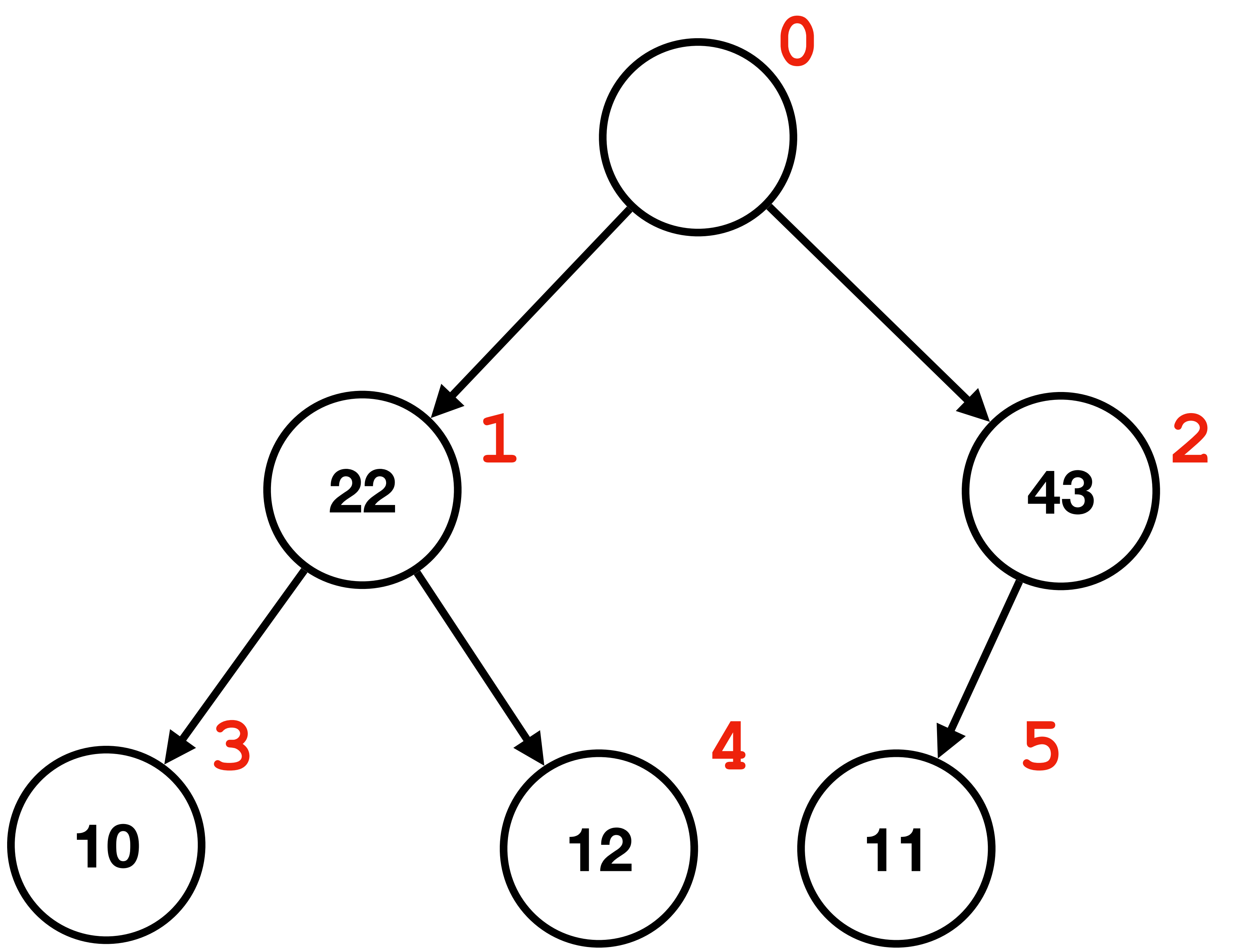
Heap

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					

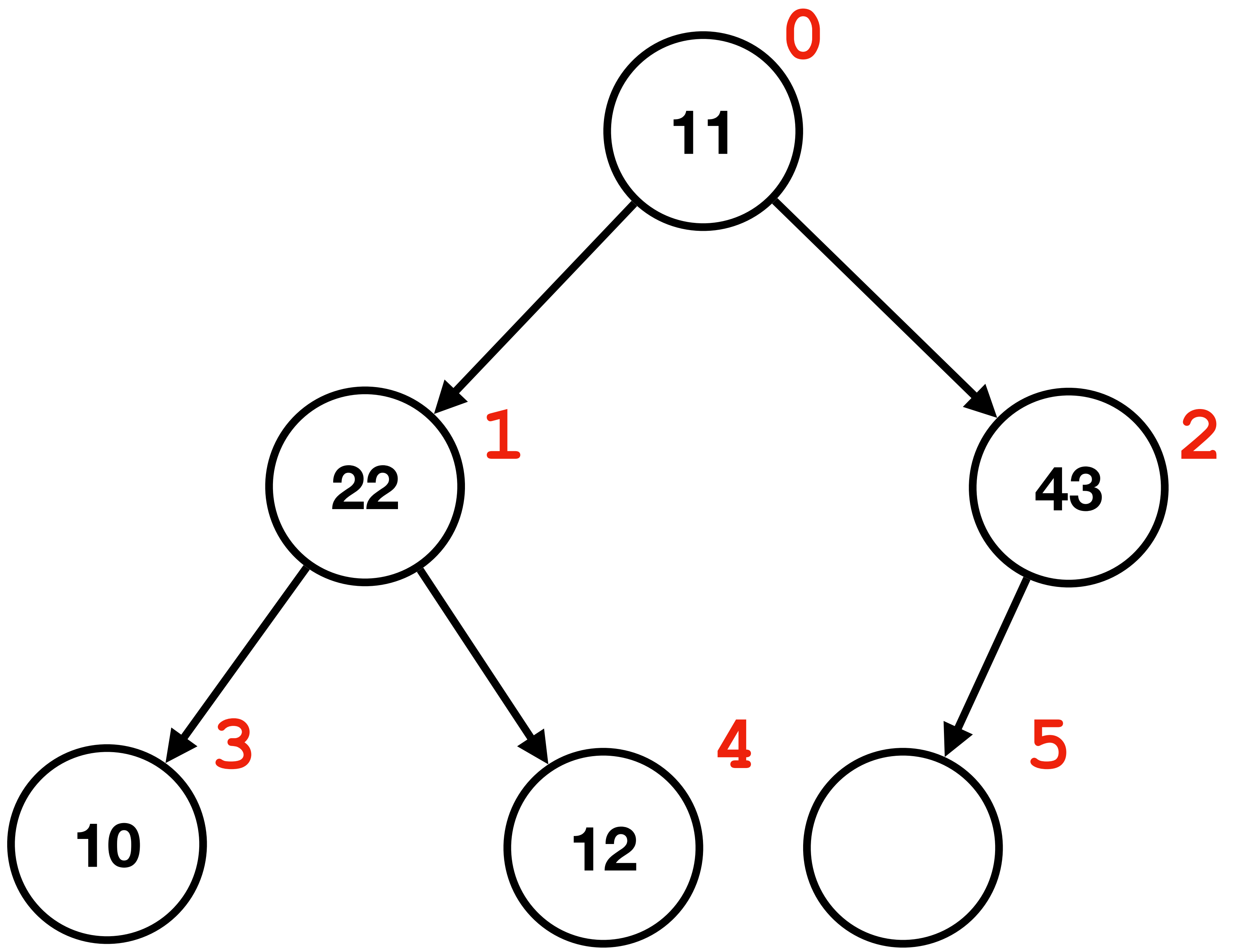
Heap

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						

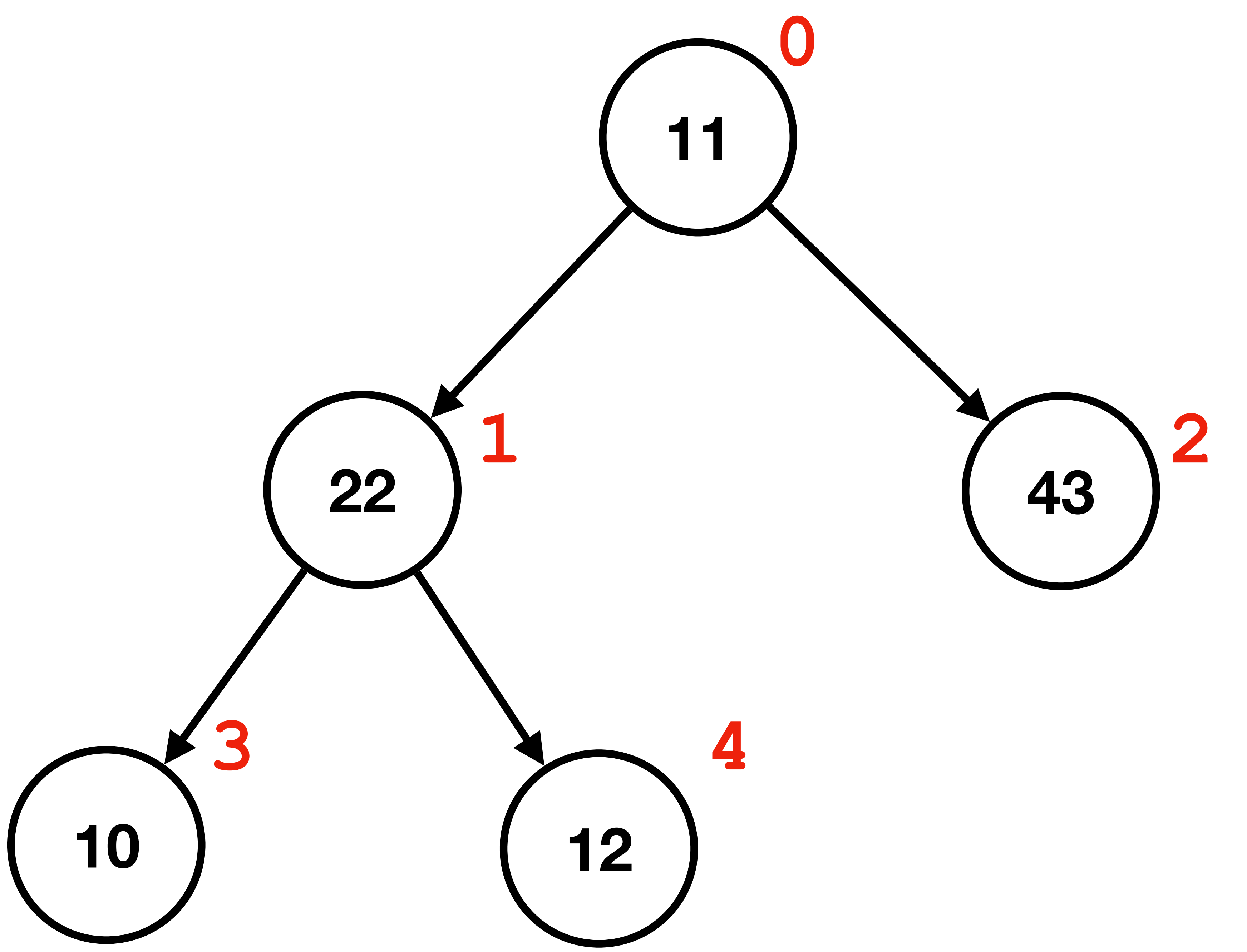
Heap

Operations

- `remove_top()`

1. Remove root (save for later return)

2. Maintain shape: replace root with last element in the array
   (Decrement `heap_size`)



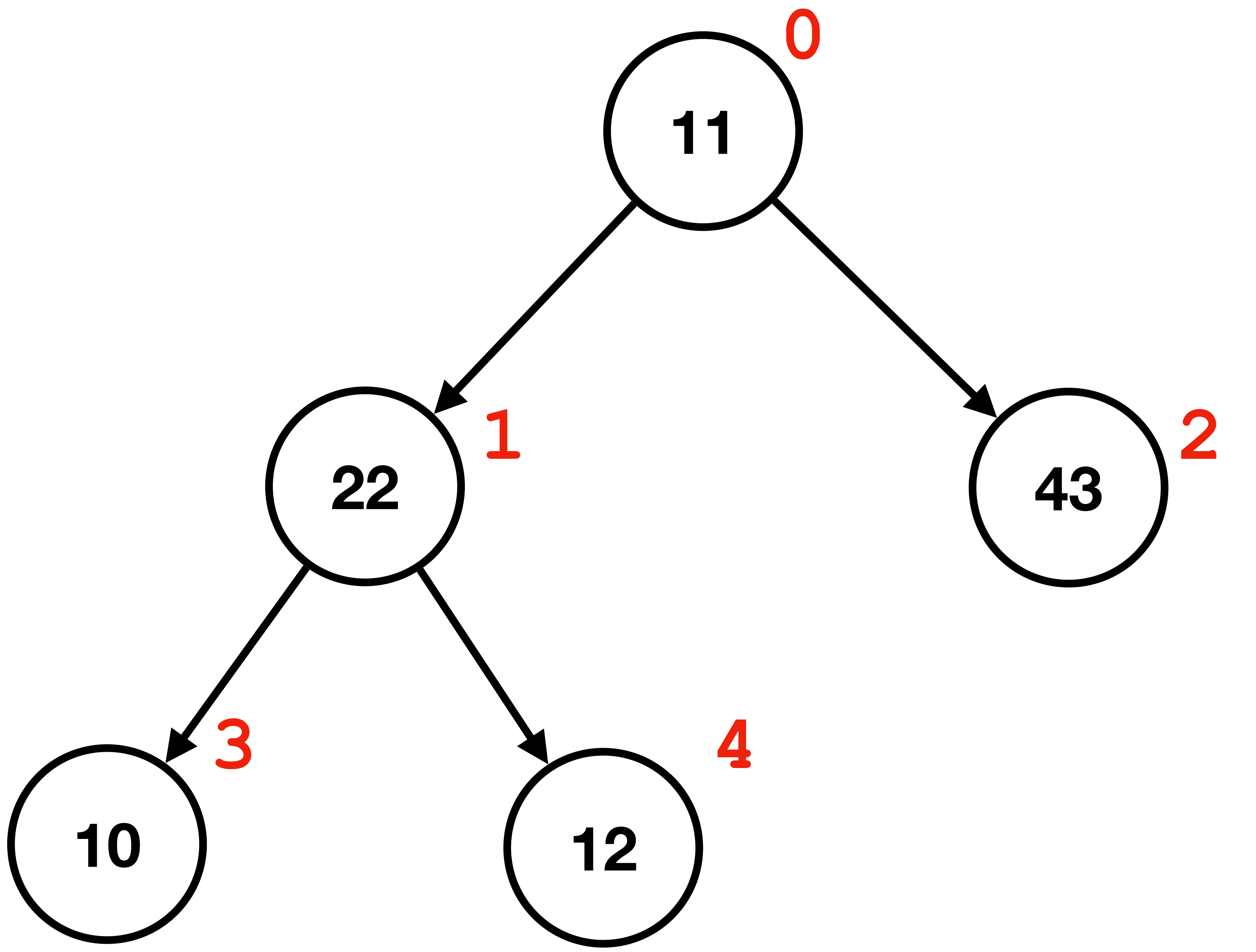
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
11	22	43	10	12						

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 >= both of its children, if correct stop
    2. If not, swapped with the larger



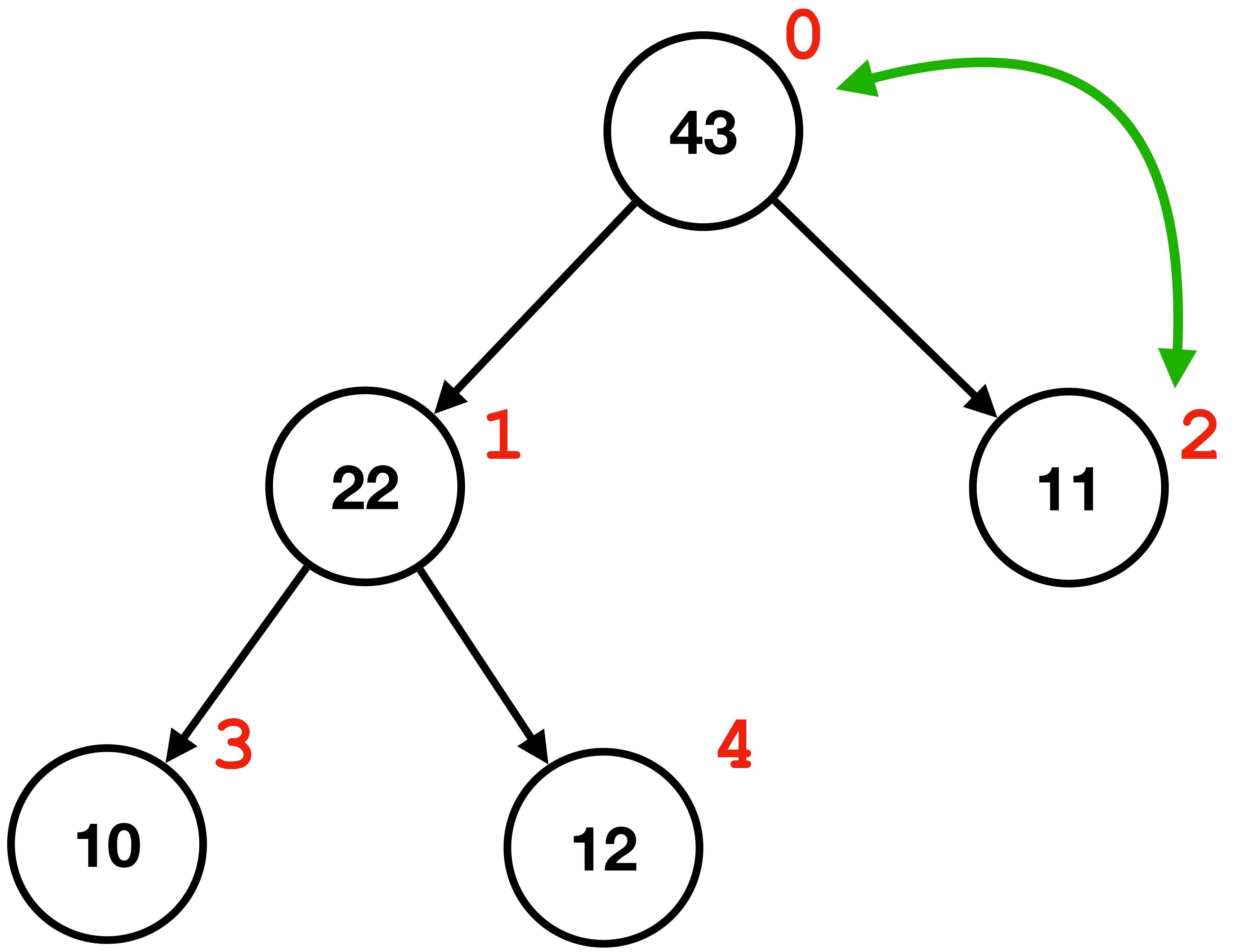
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
11	22	43	10	12						

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 >= both of its children, if correct stop
   2. If not, swapped with the larger



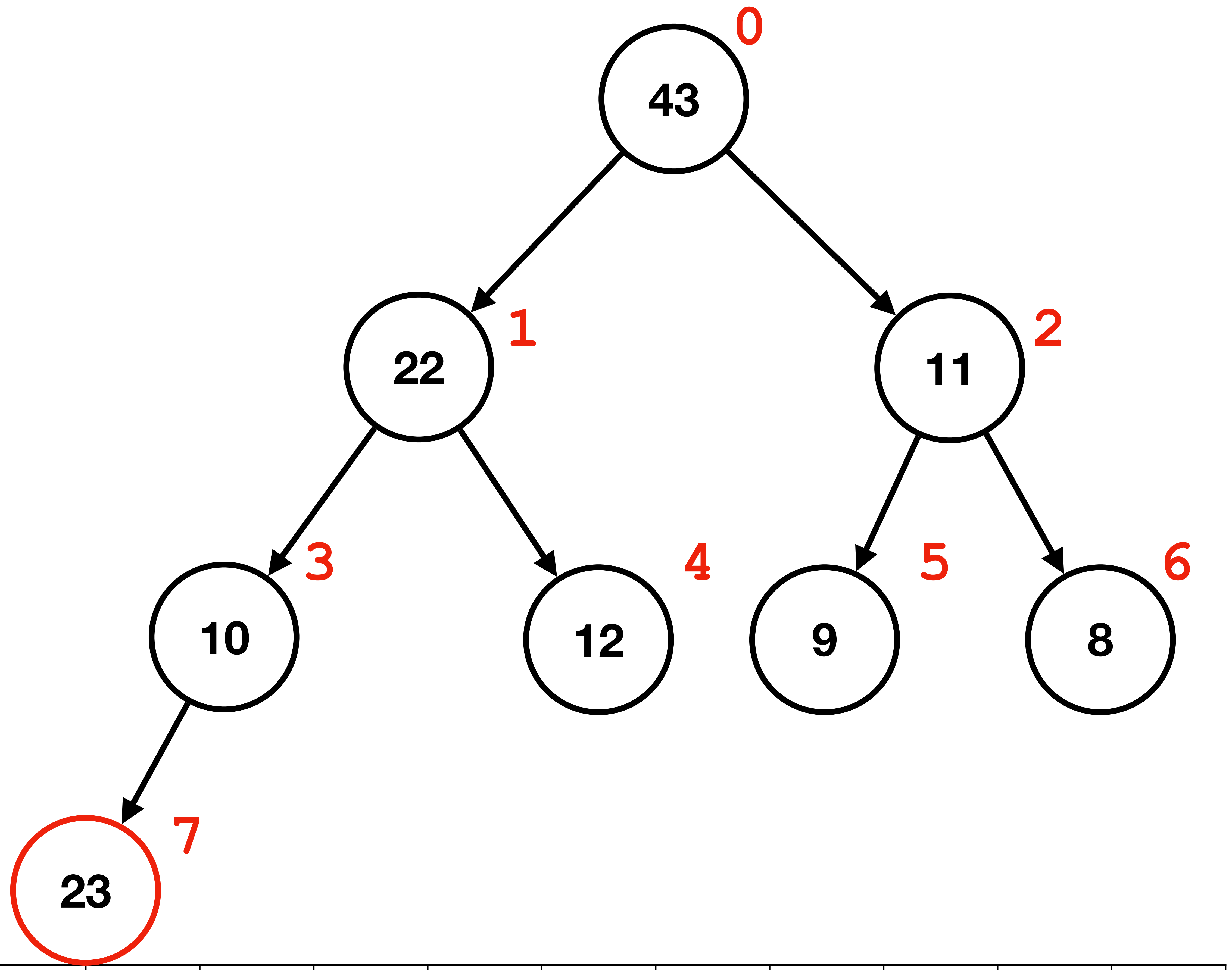
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12						

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;
        }
    }
}
```



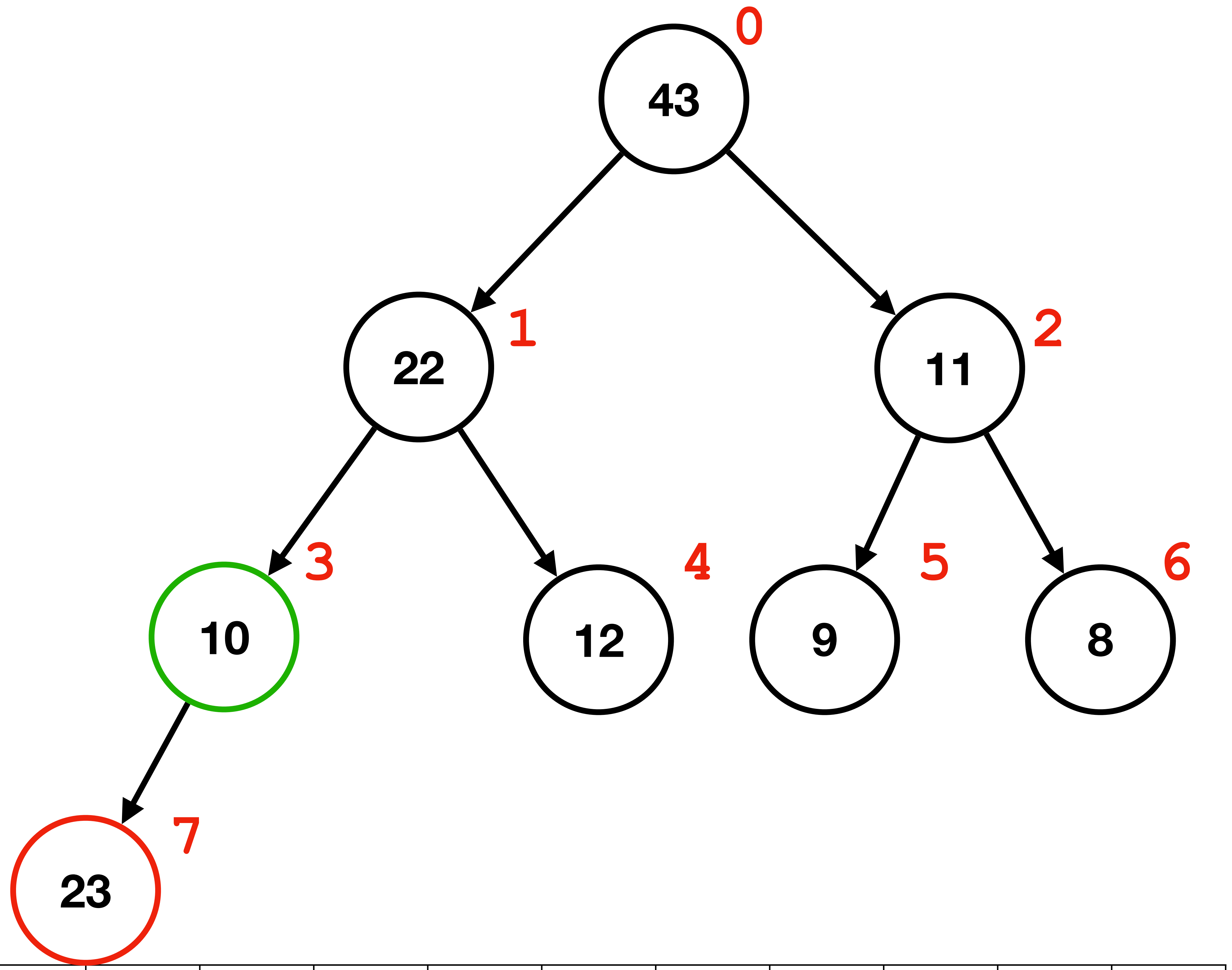
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12	9	8	23			

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;
        }
    }
}
```



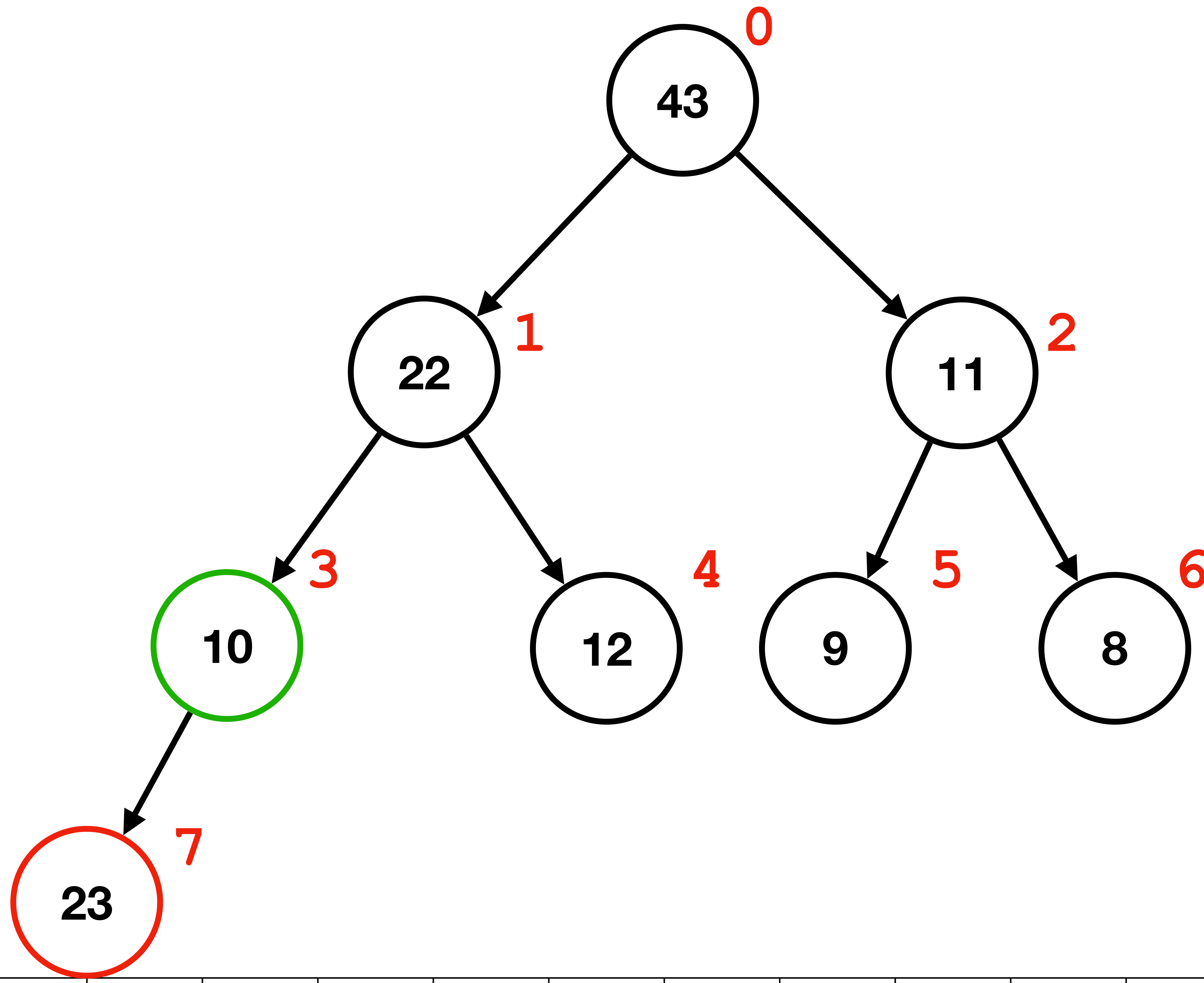
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12	9	8	23			

Heap

Bubble Up

```c
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;
        }
    }
}
```



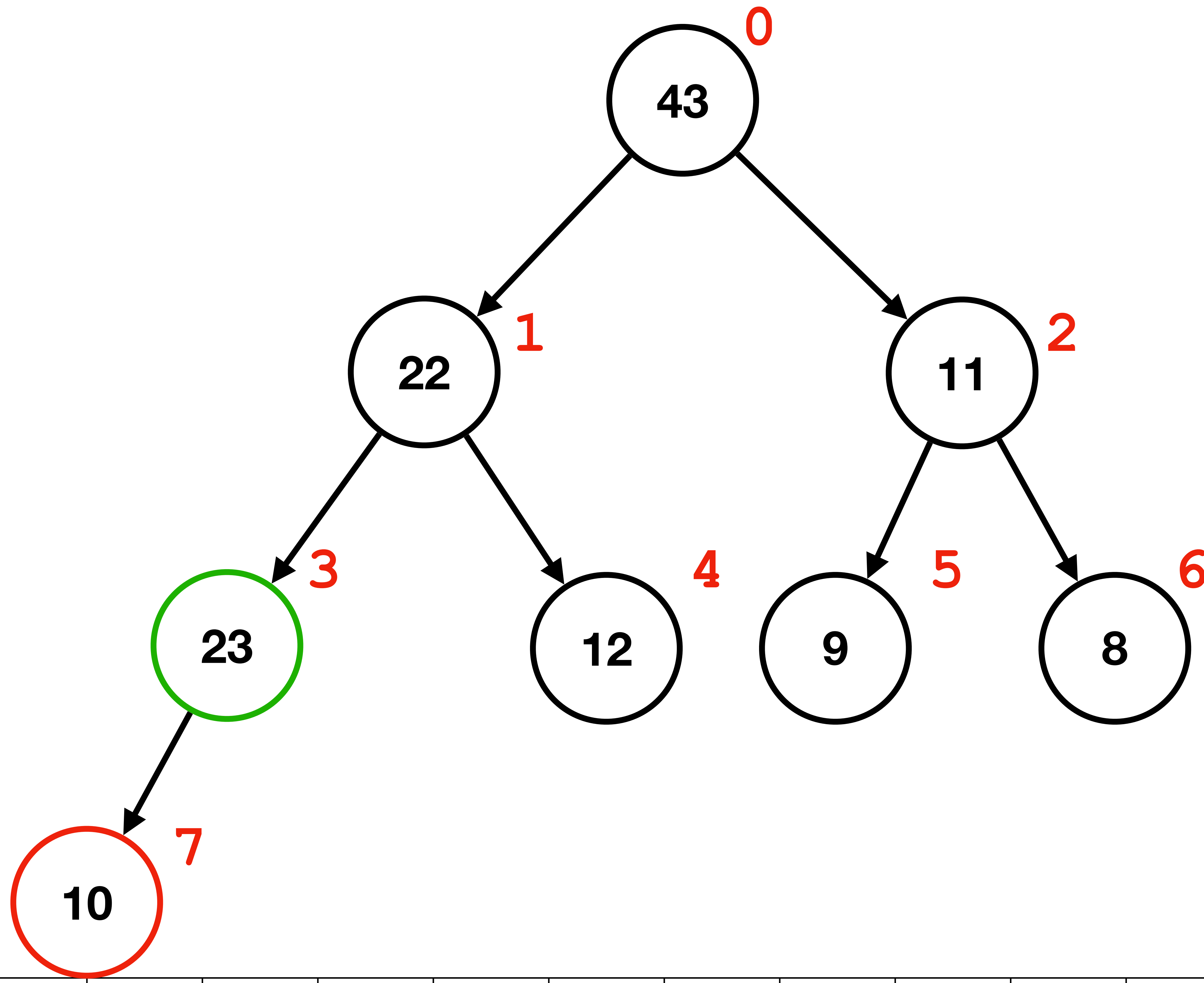
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	10	12	9	8	23			

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;
        }
    }
}
```



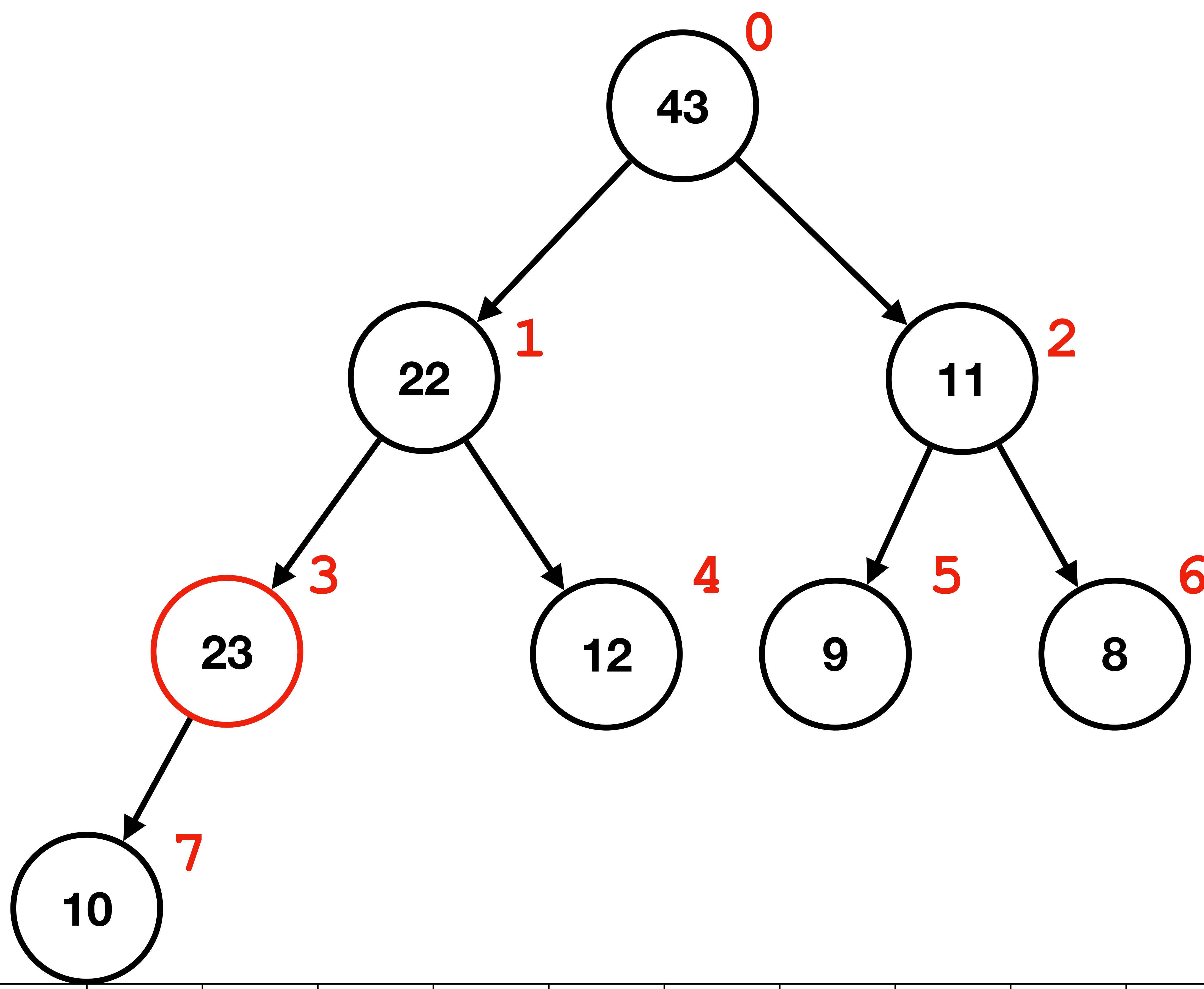
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	23	12	9	8	10			

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;
        }
    }
}
```



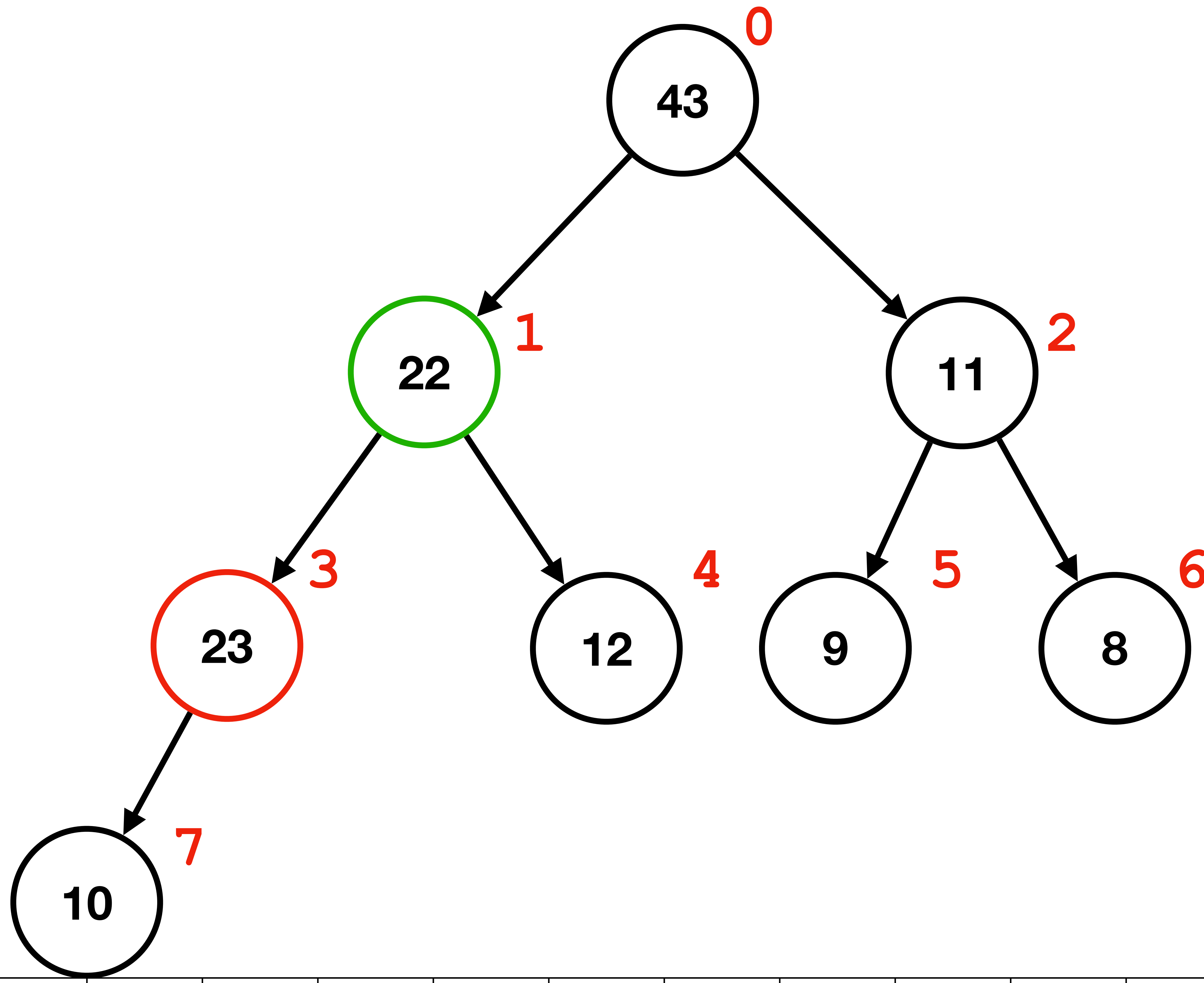
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	23	12	9	8	10			

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;
        }
    }
}
```



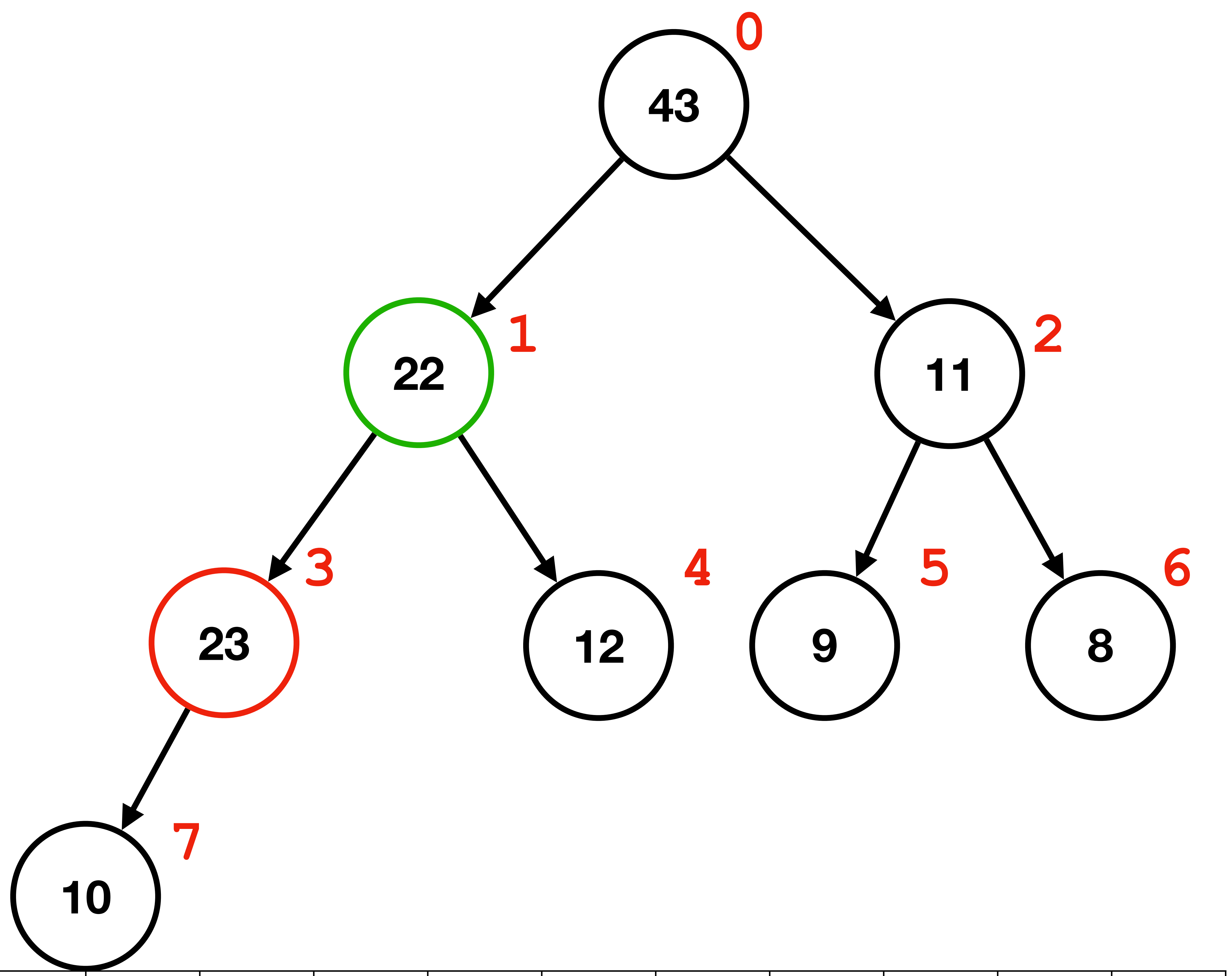
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	23	12	9	8	10			

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;
        }
    }
}
```



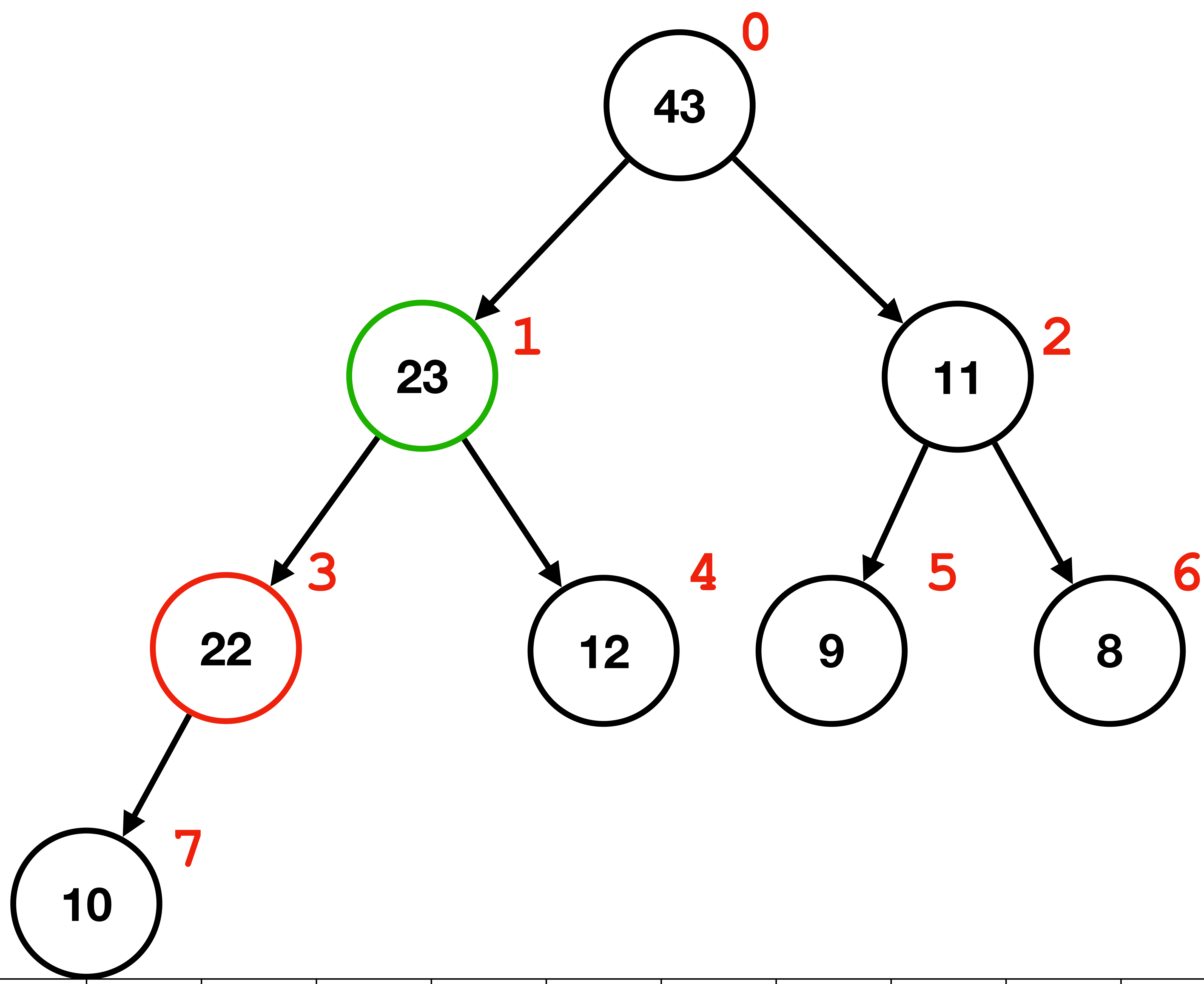
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	22	11	23	12	9	8	10			

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;
        }
    }
}
```



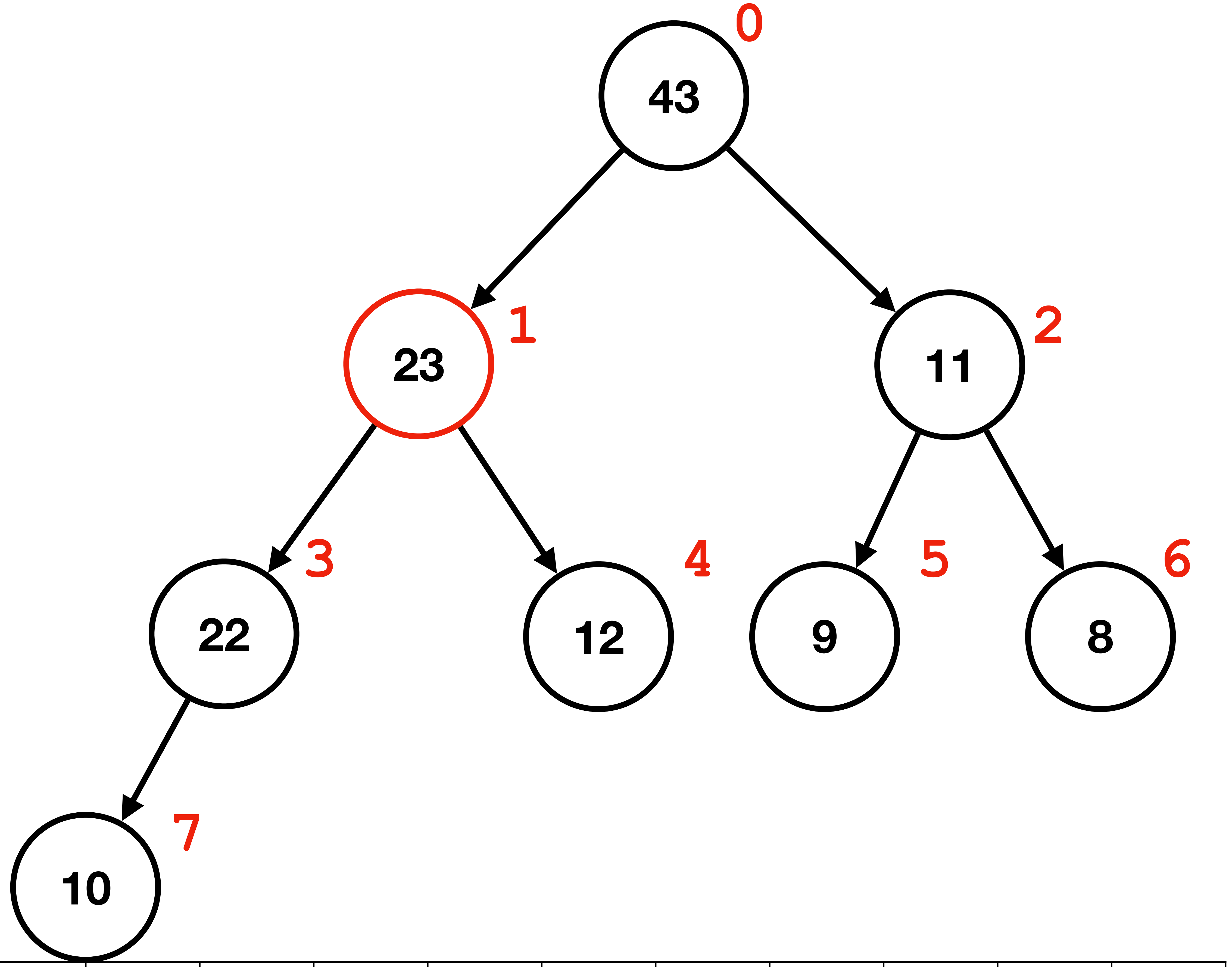
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10			

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;
        }
    }
}
```



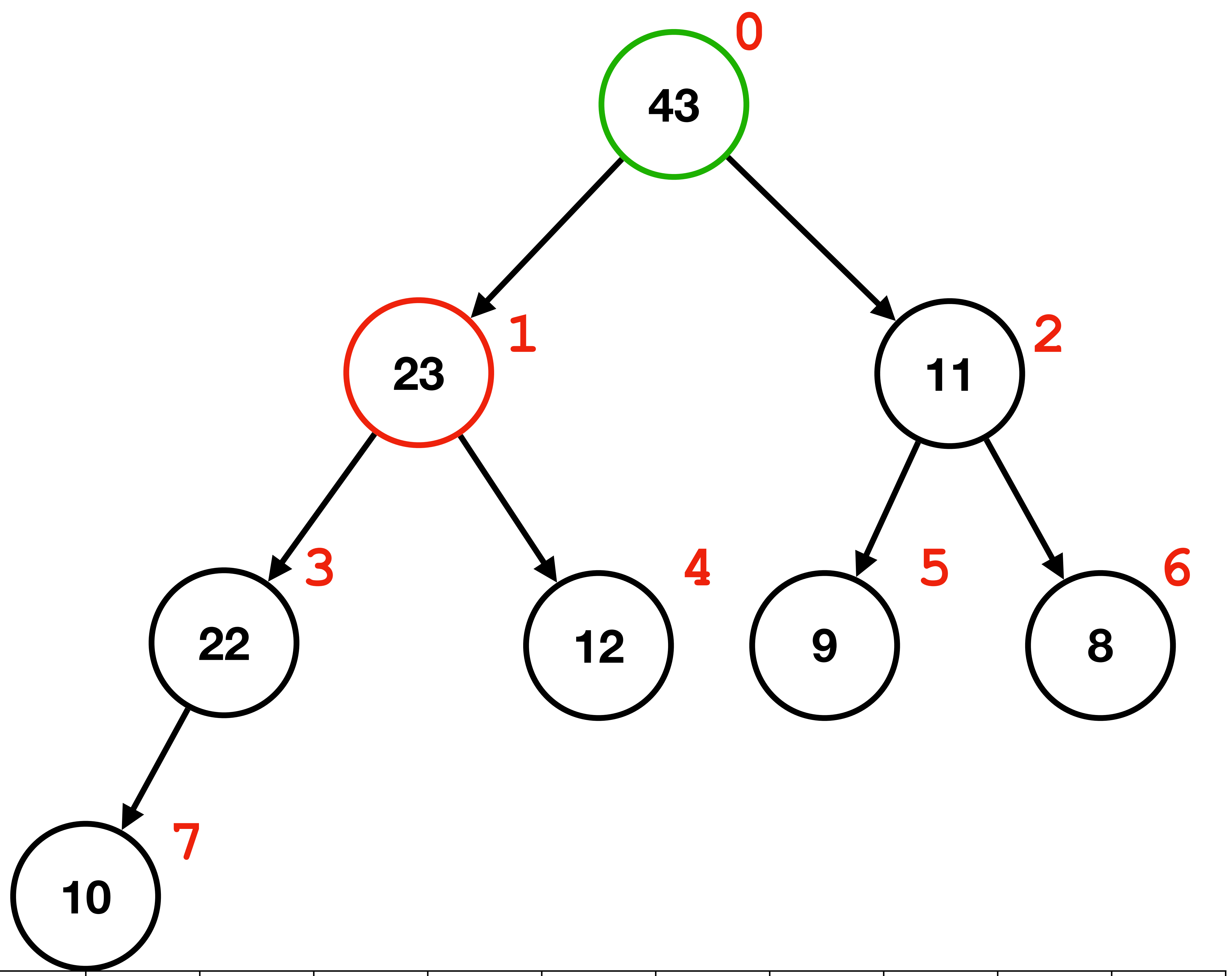
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10			

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;
        }
    }
}
```



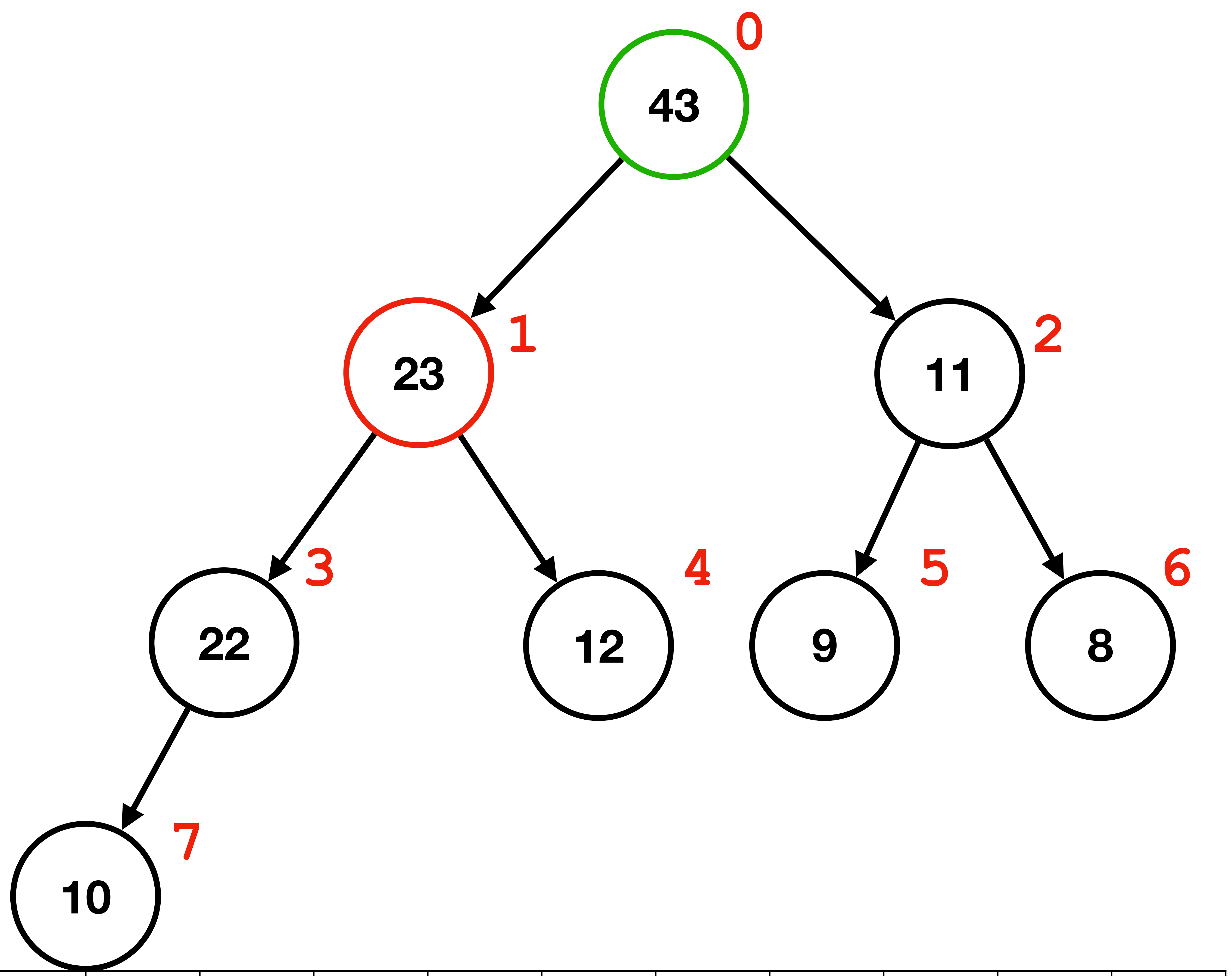
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10			

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;
        }
    }
}
```



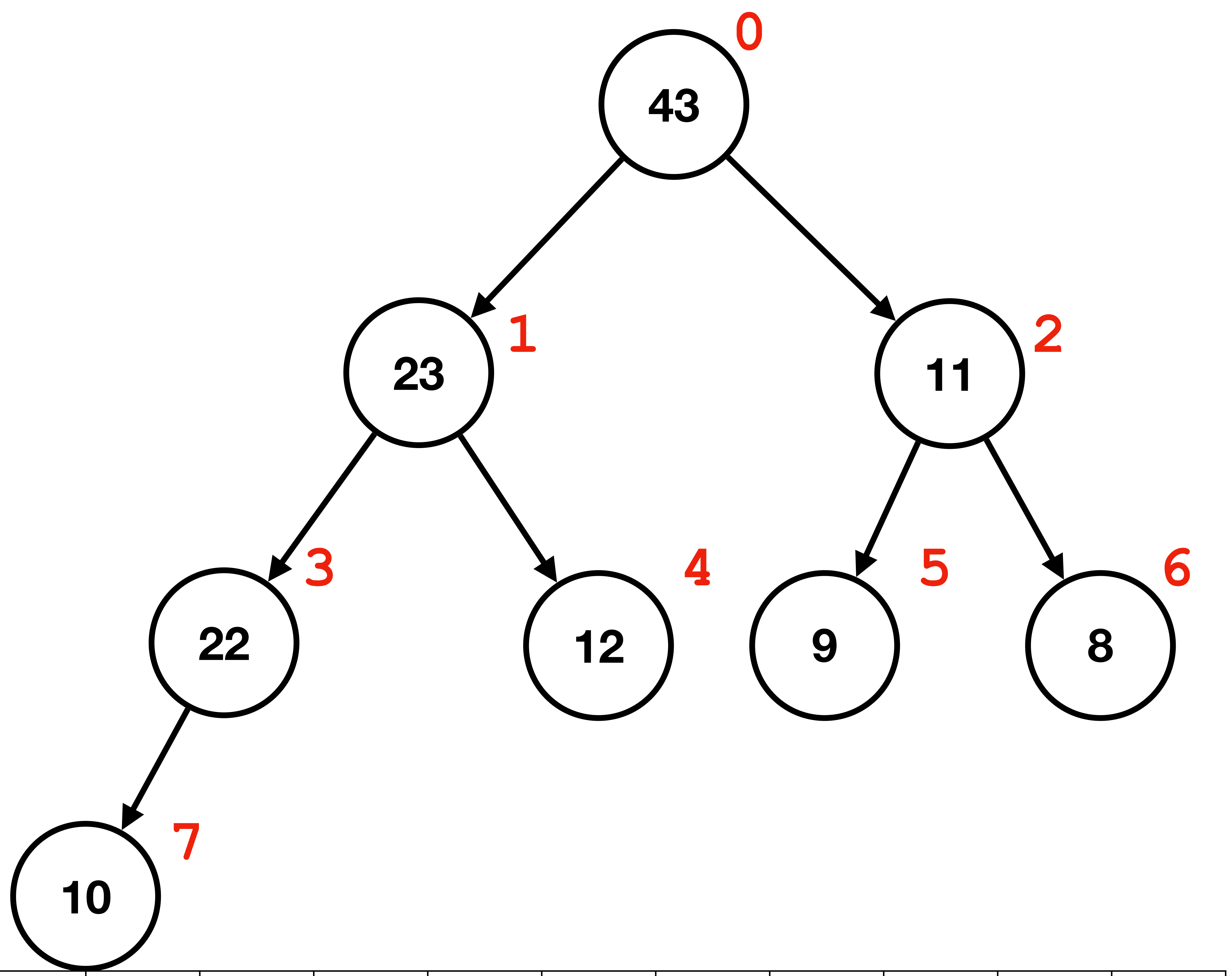
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10			

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;
        }
    }
}
```



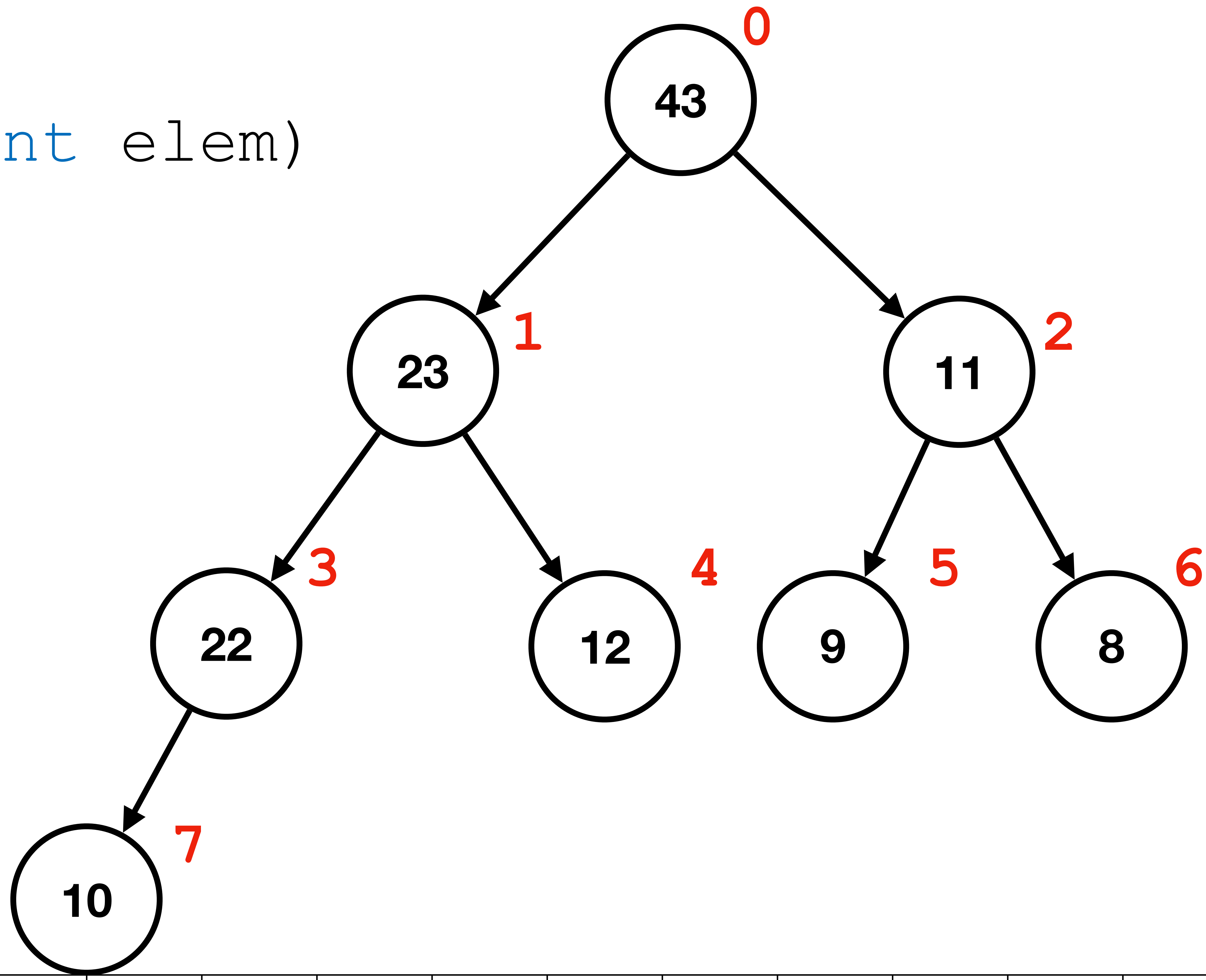
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10			

Heap

Bubble Up

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

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



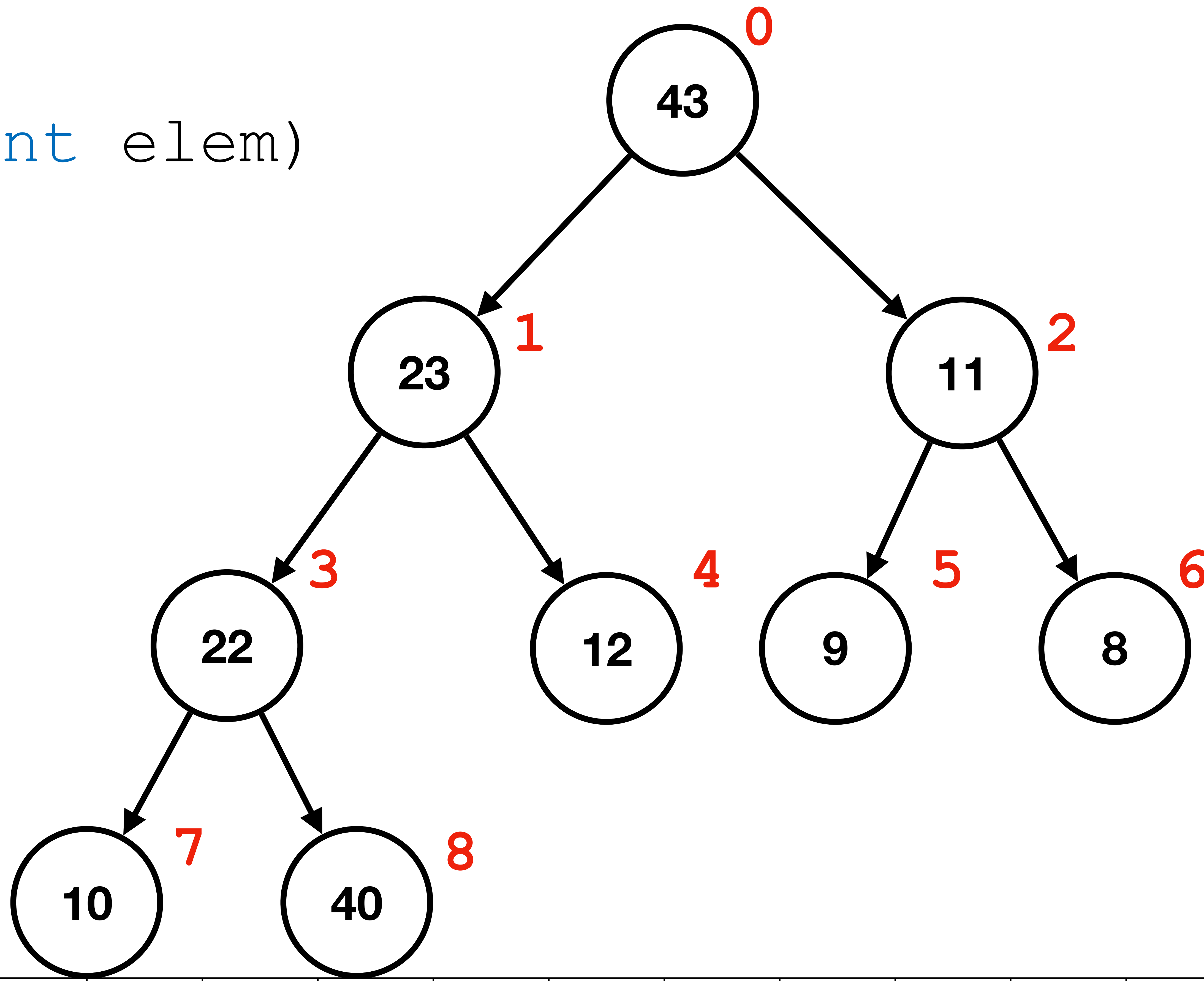
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10			

Heap

Bubble Up

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

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



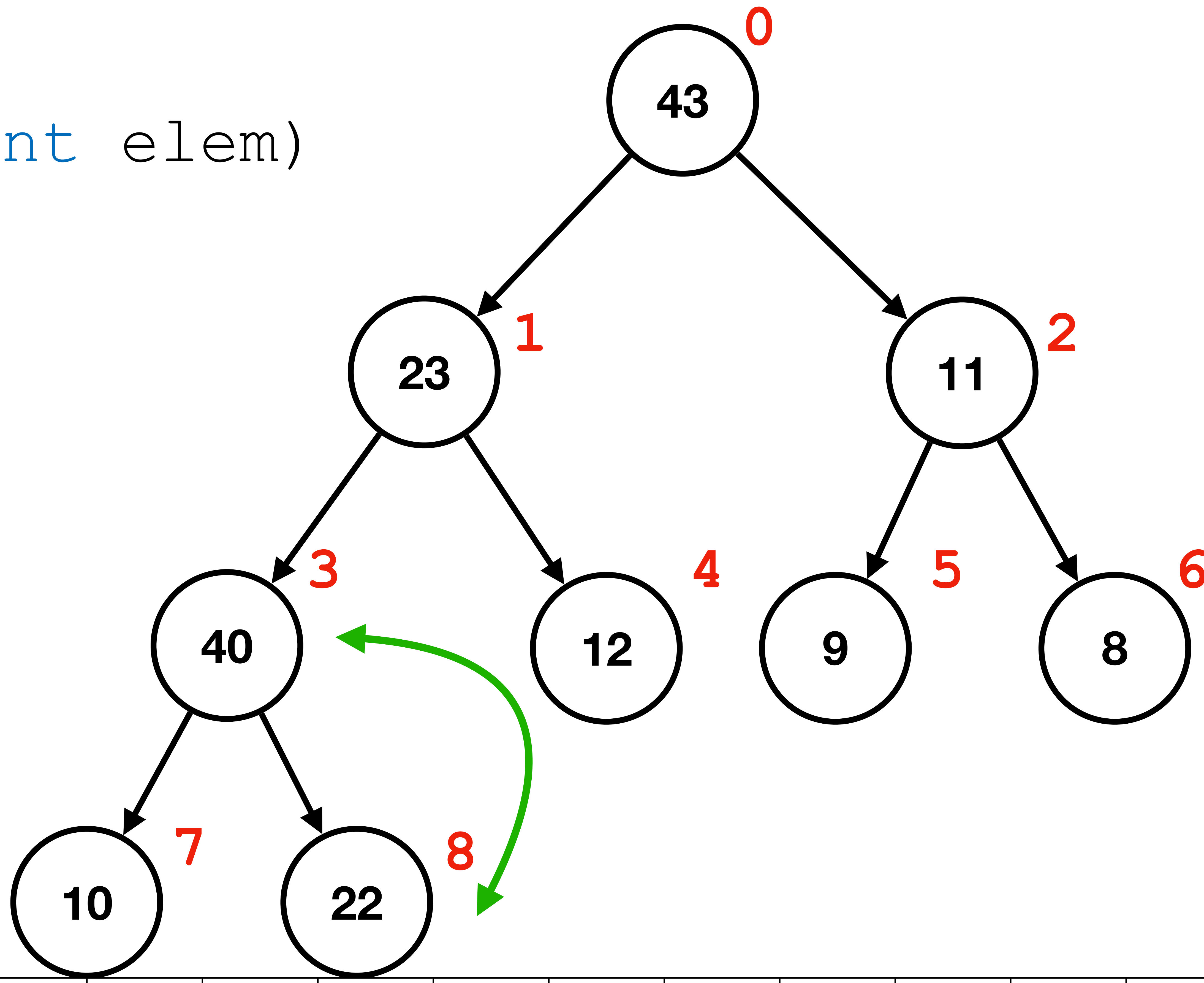
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	22	12	9	8	10	40		

Heap

Bubble Up

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

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



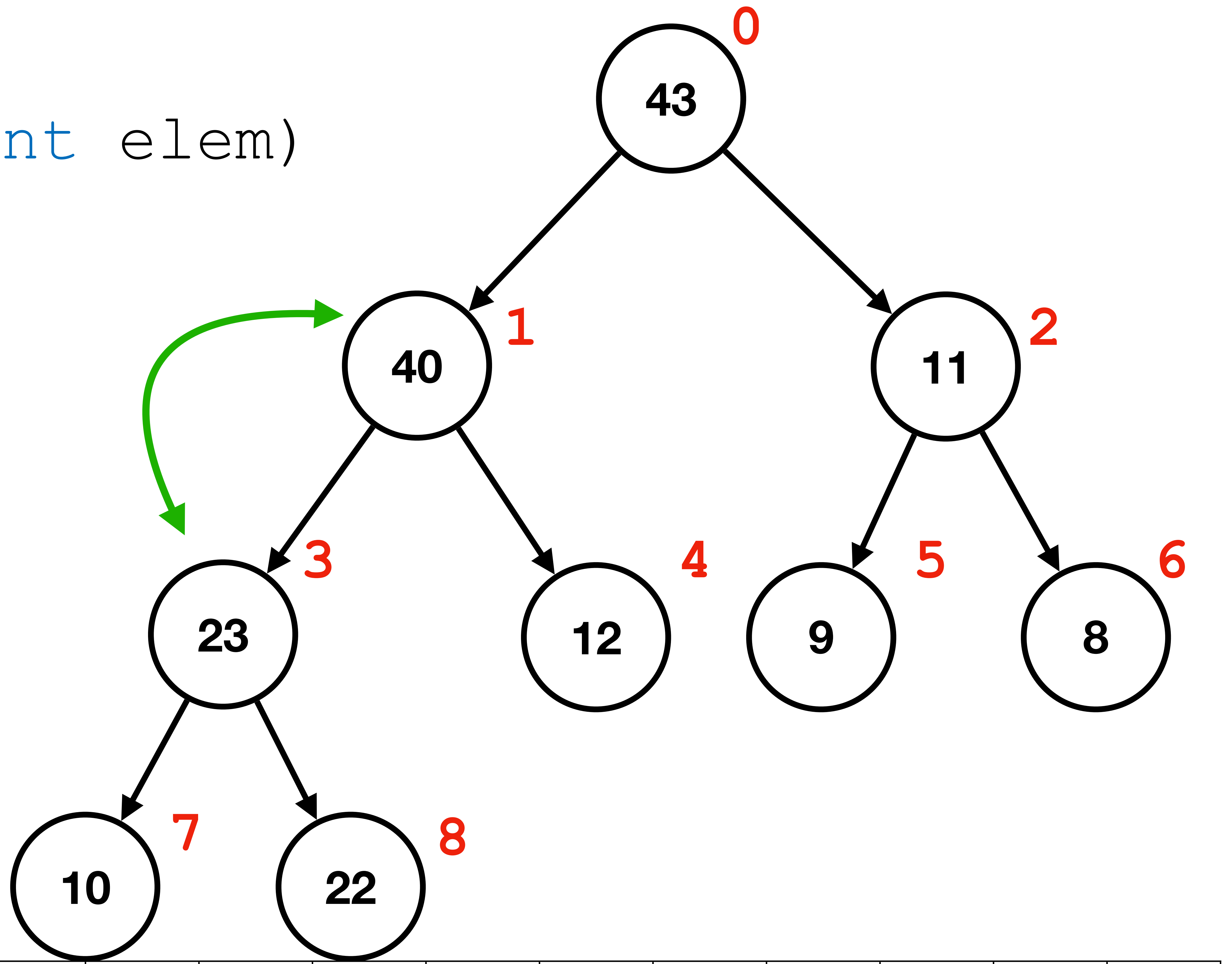
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	23	11	40	12	9	8	10	22		

Heap

Bubble Up

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

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



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
43	40	11	23	12	9	8	10	22		

Heap

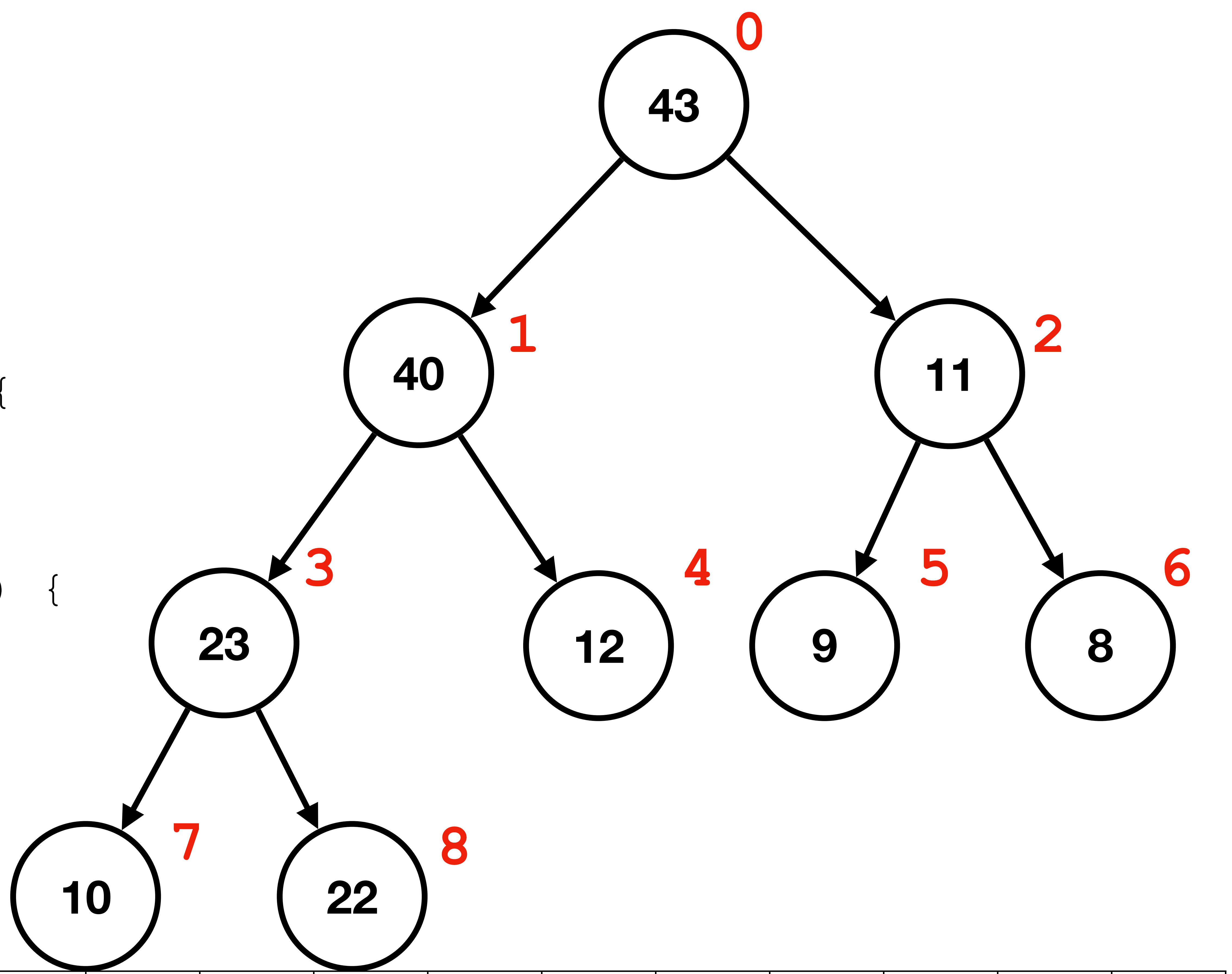
Bubble Down

```
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]
43	40	11	23	12	9	8	10	22		

Heap

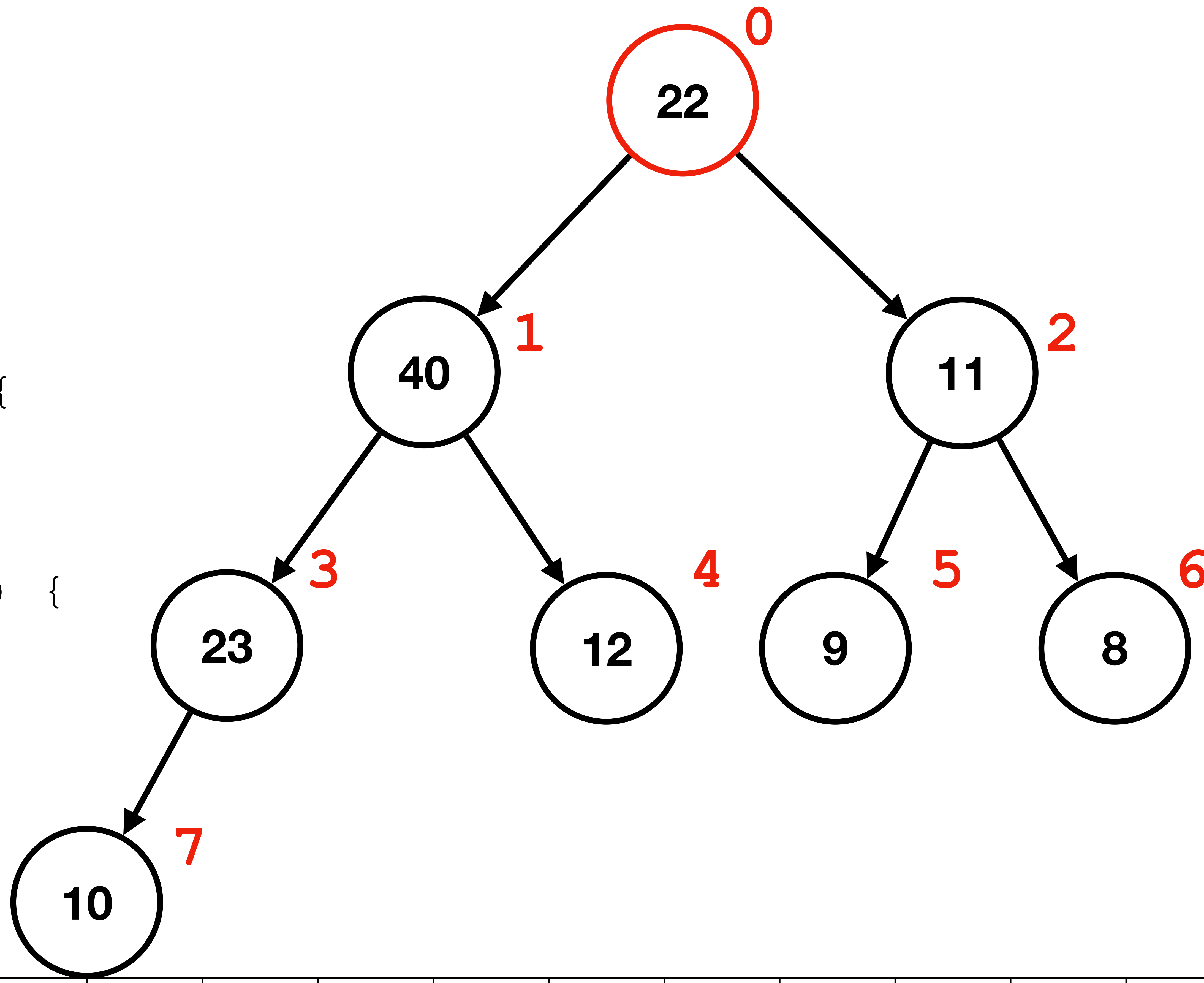
Bubble Down

```c
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]
22	40	11	23	12	9	8	10			

Heap

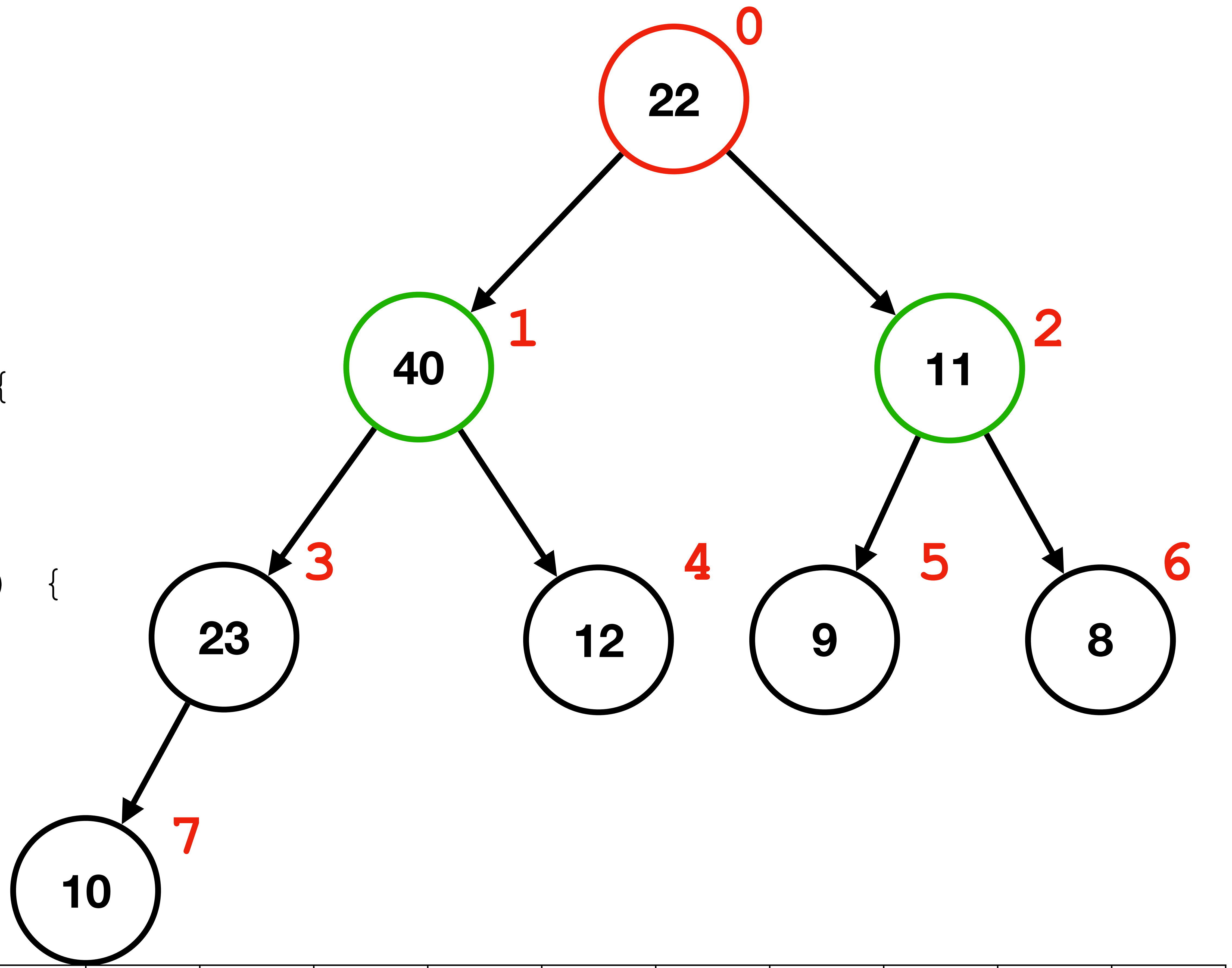
Bubble Down

```
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;
        }
    }
}
```



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

Heap

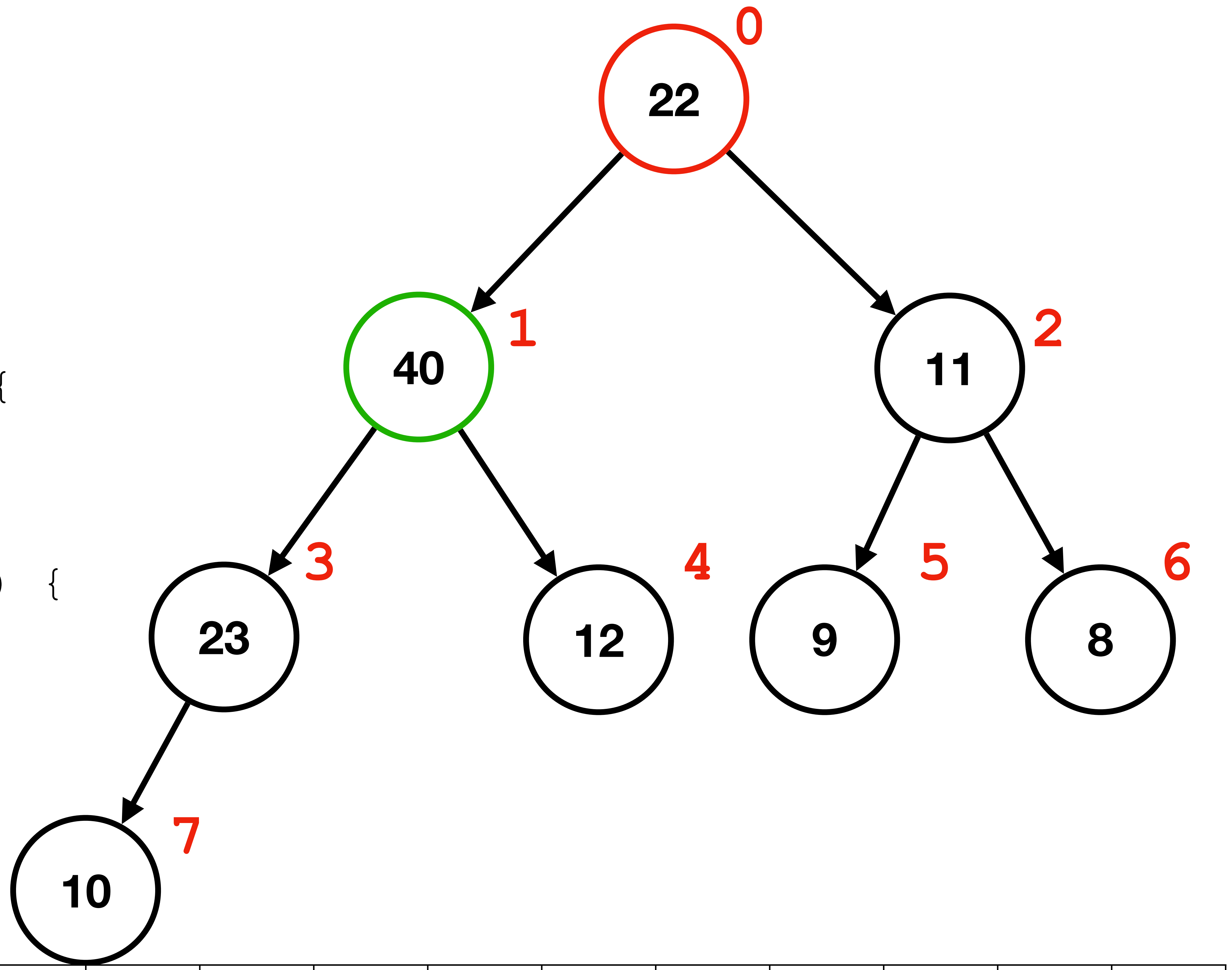
Bubble Down

```
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;
        }
    }
}
```



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

Heap

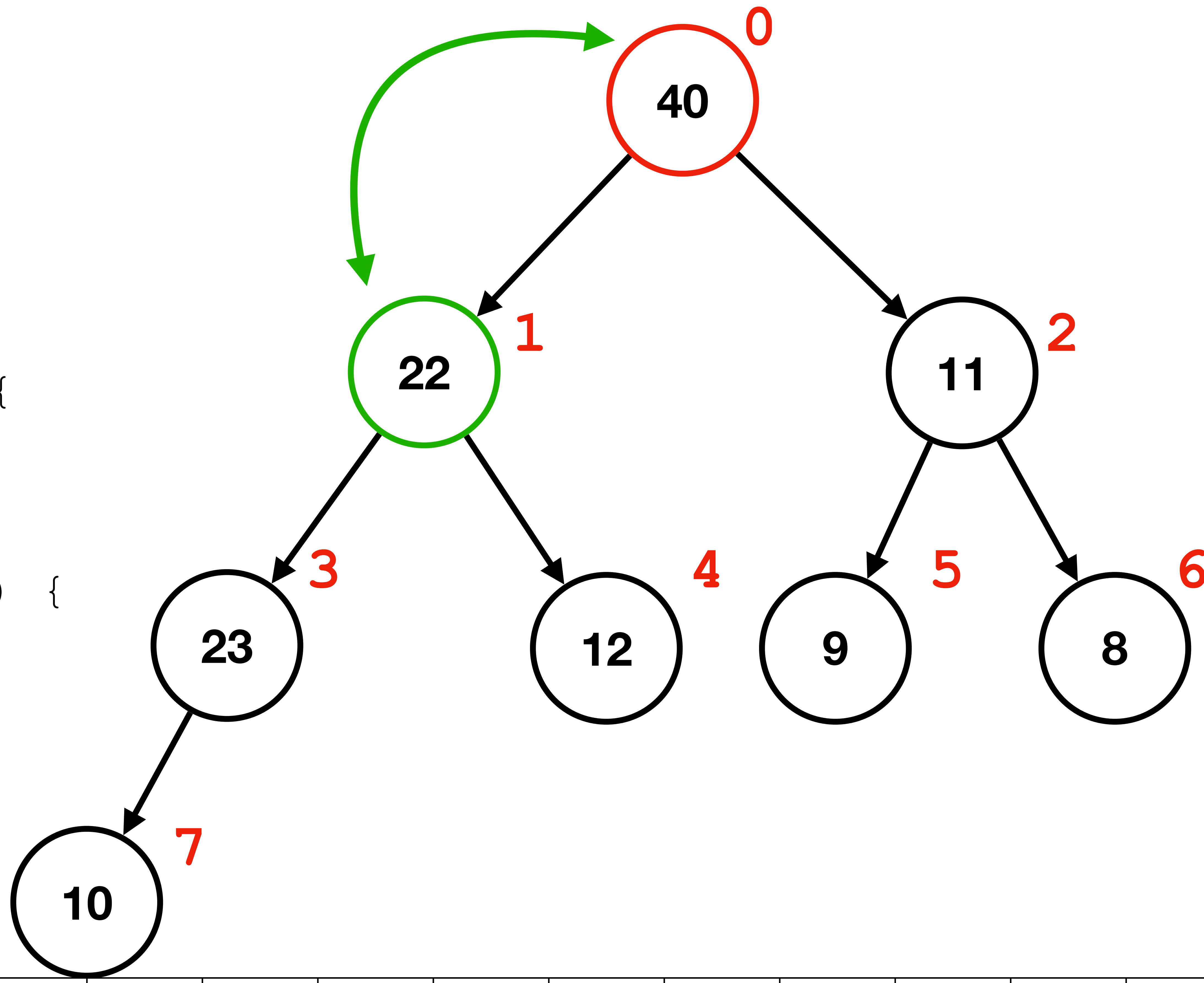
Bubble Down

```
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;
        }
    }
}
```



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

Heap

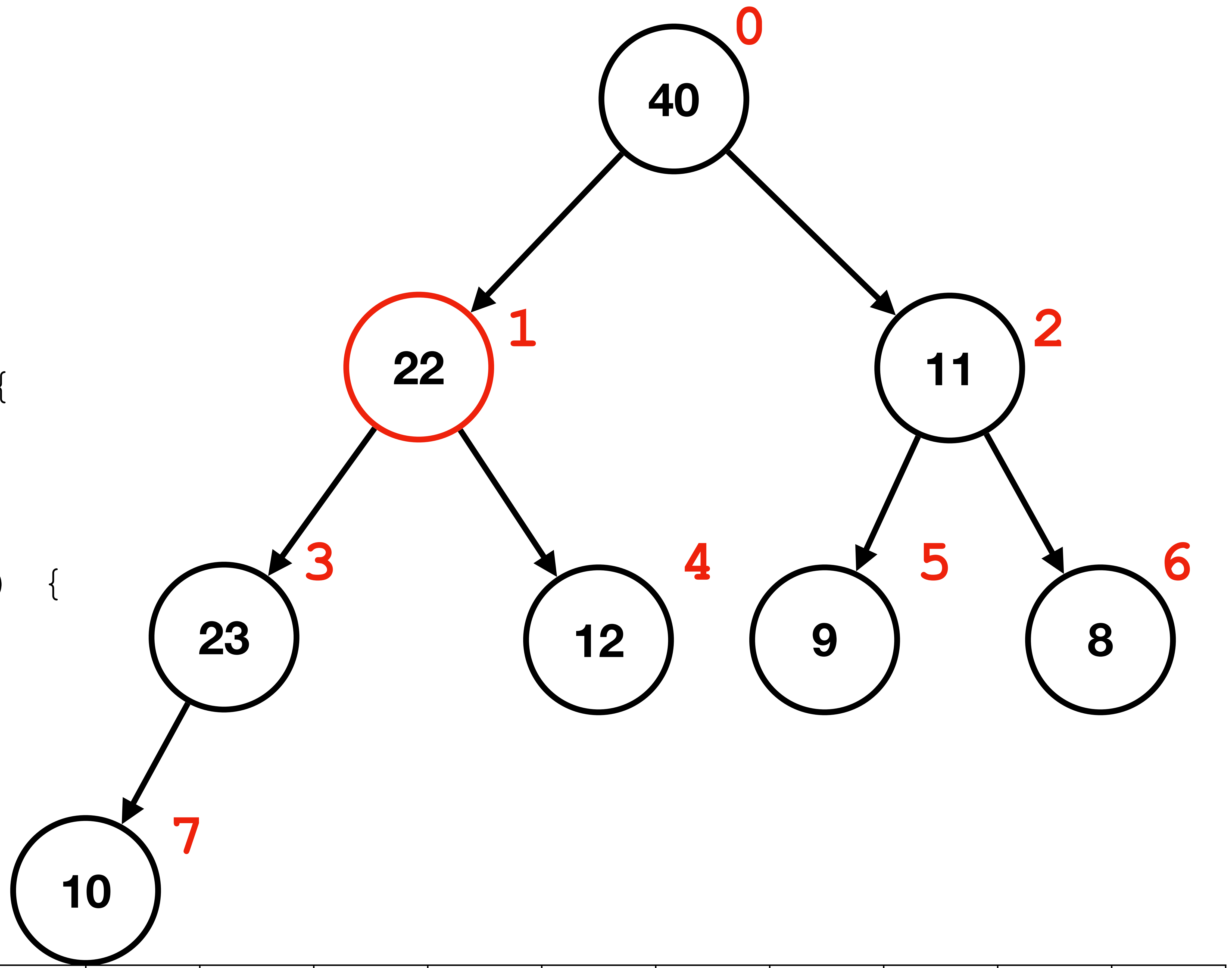
Bubble Down

```
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;
        }
    }
}
```



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

Heap

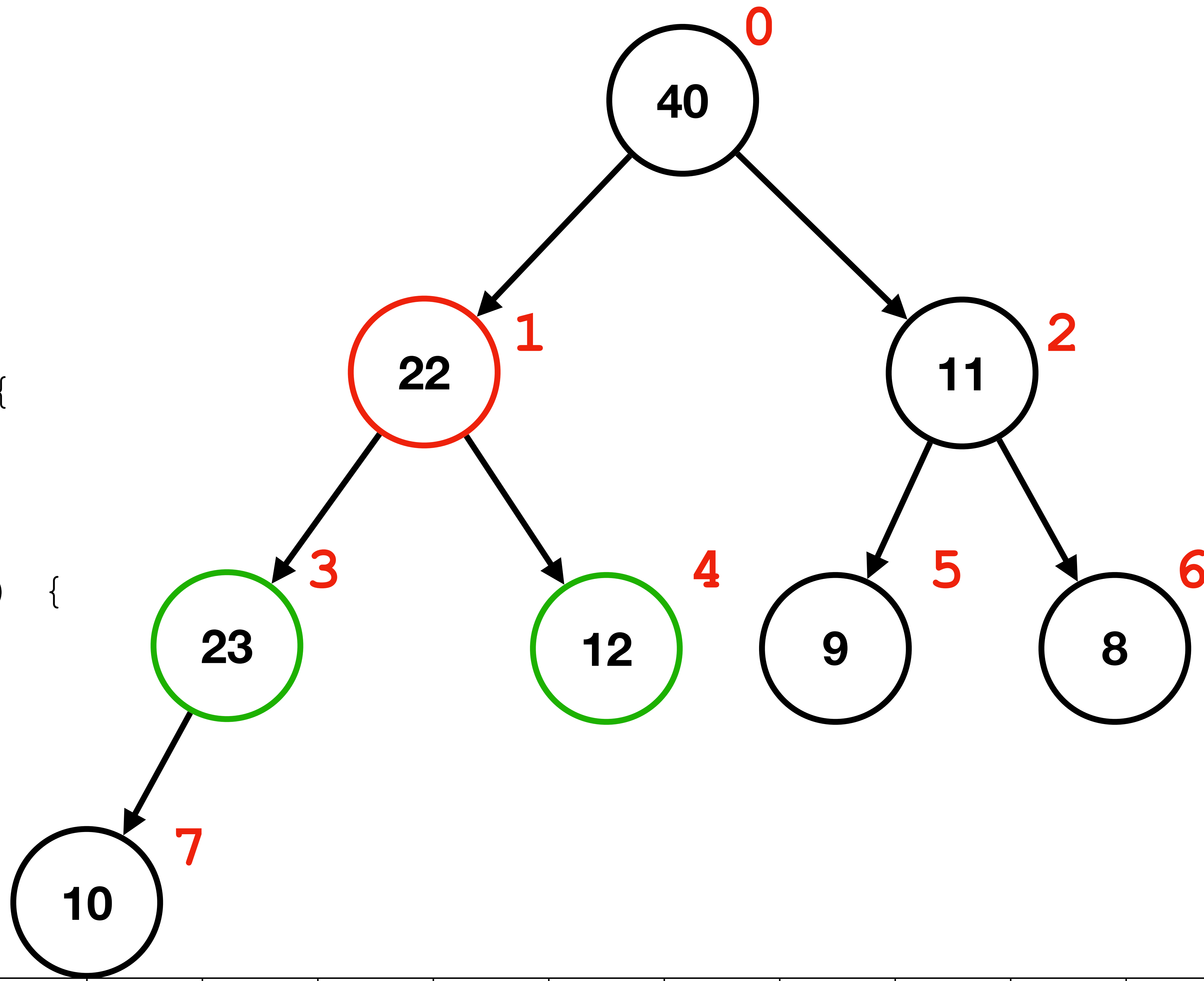
Bubble Down

```
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;
        }
    }
}
```



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

Heap

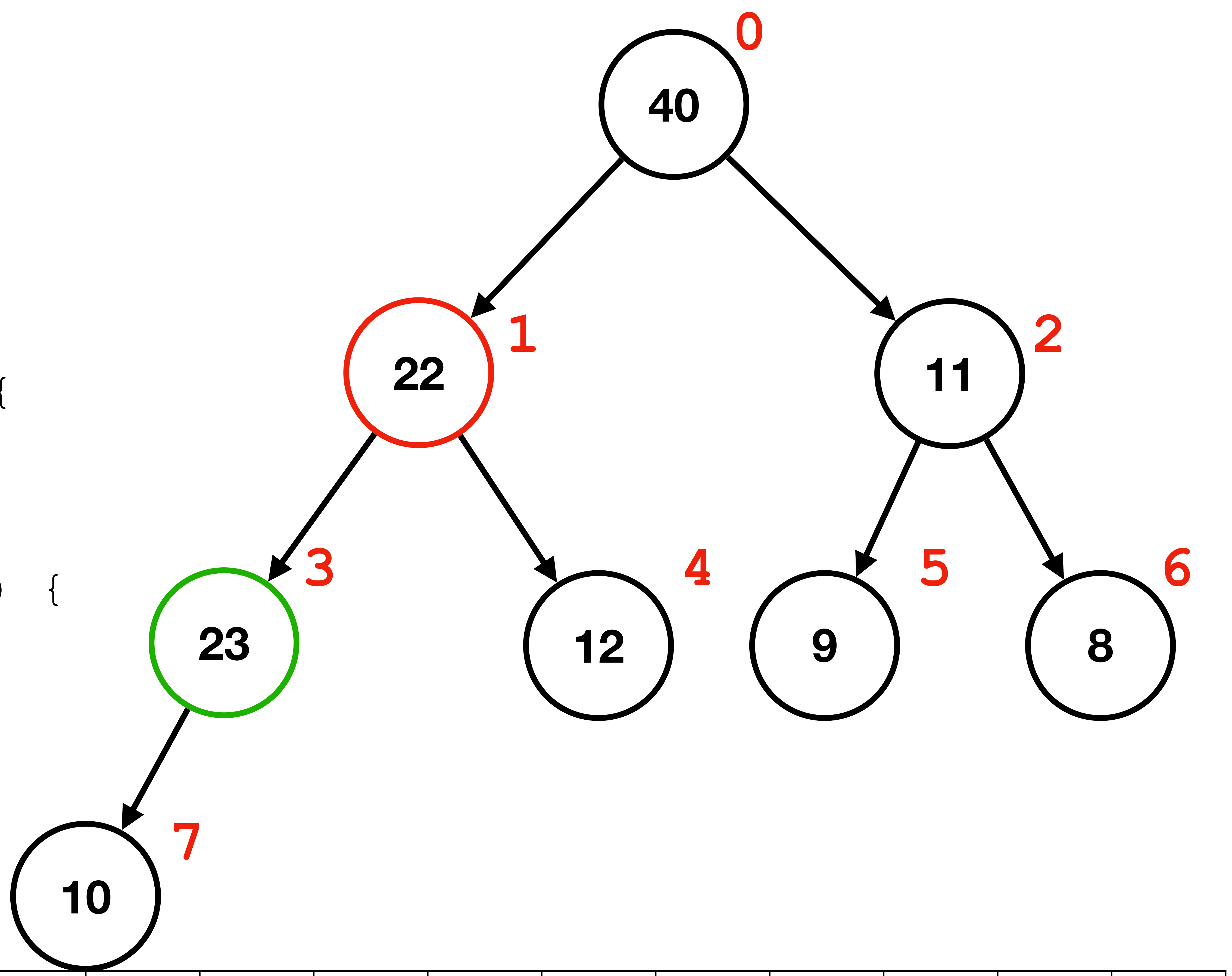
Bubble Down

```c
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	22	11	23	12	9	8	10			

Heap

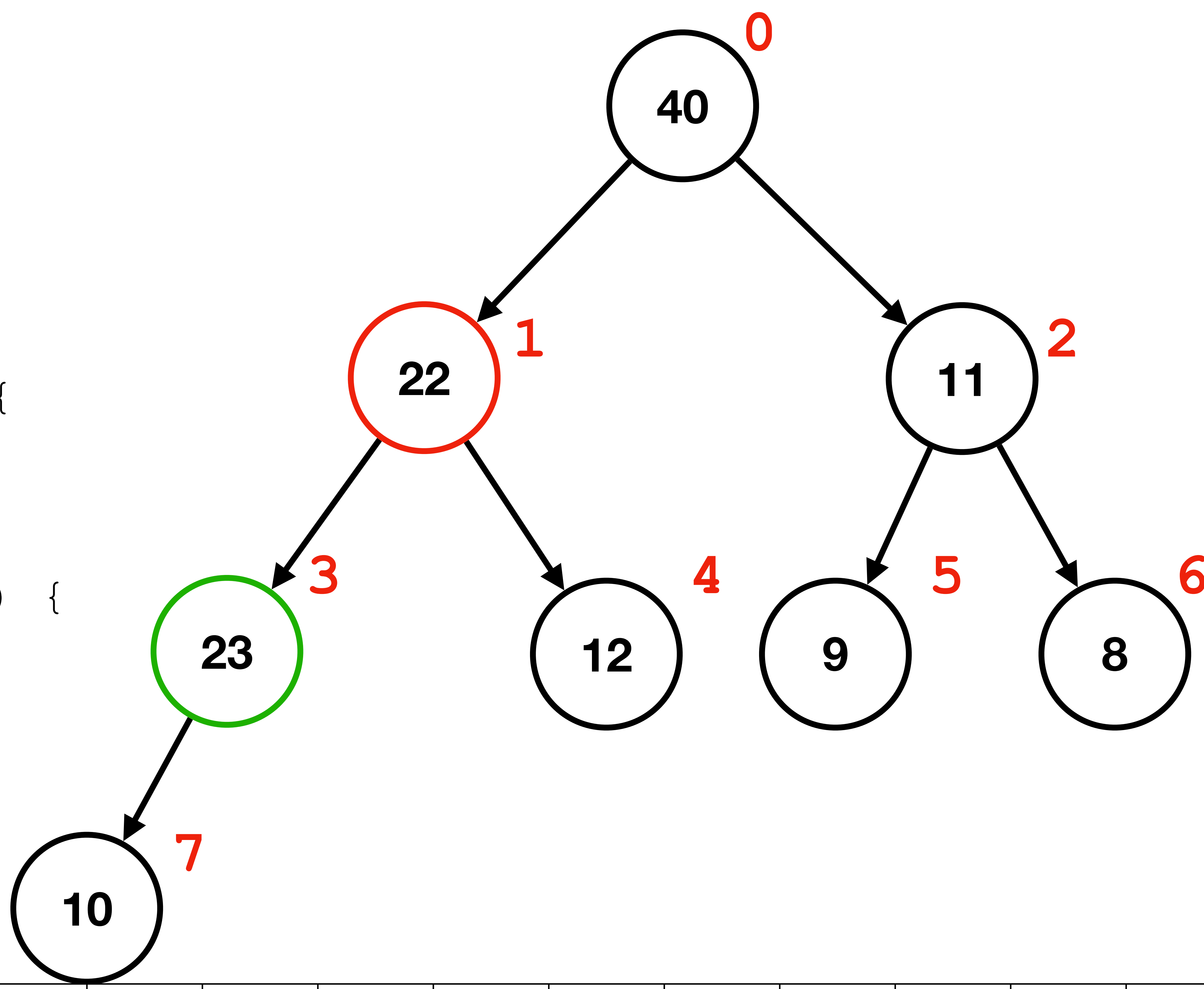
Bubble Down

```
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;
        }
    }
}
```



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

Heap

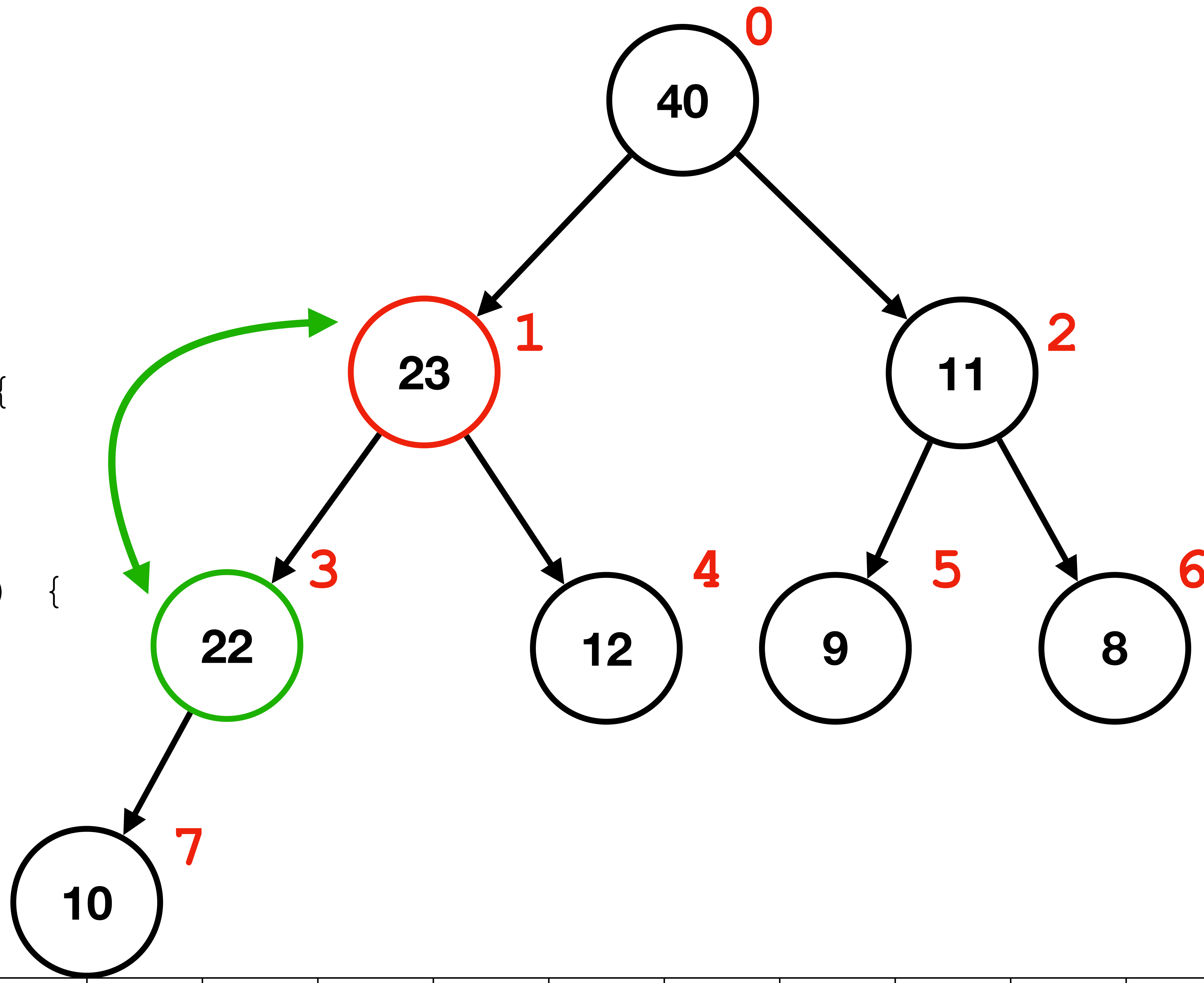
Bubble Down

```c
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

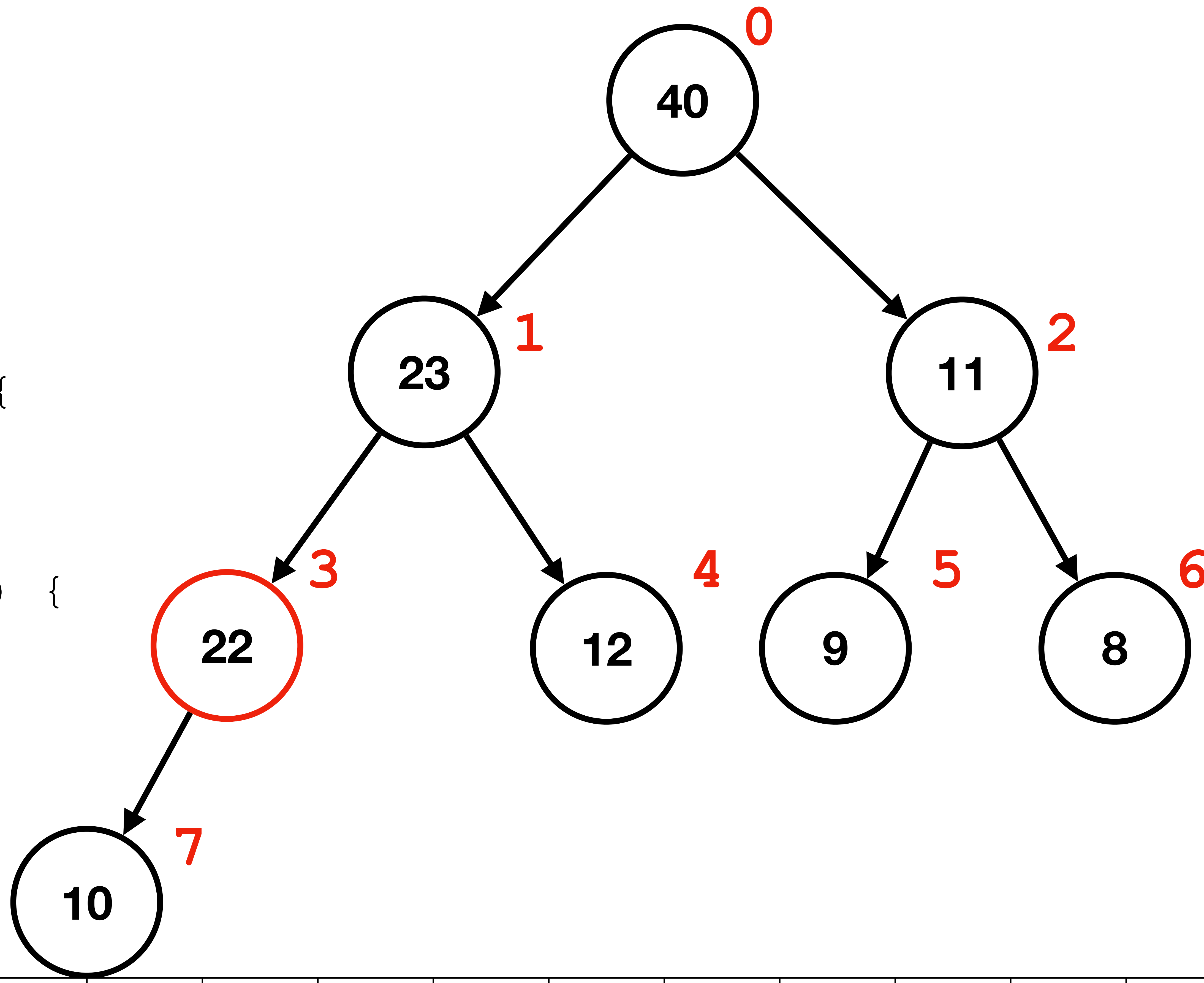
Bubble Down

```
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;
        }
    }
}
```



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

Heap

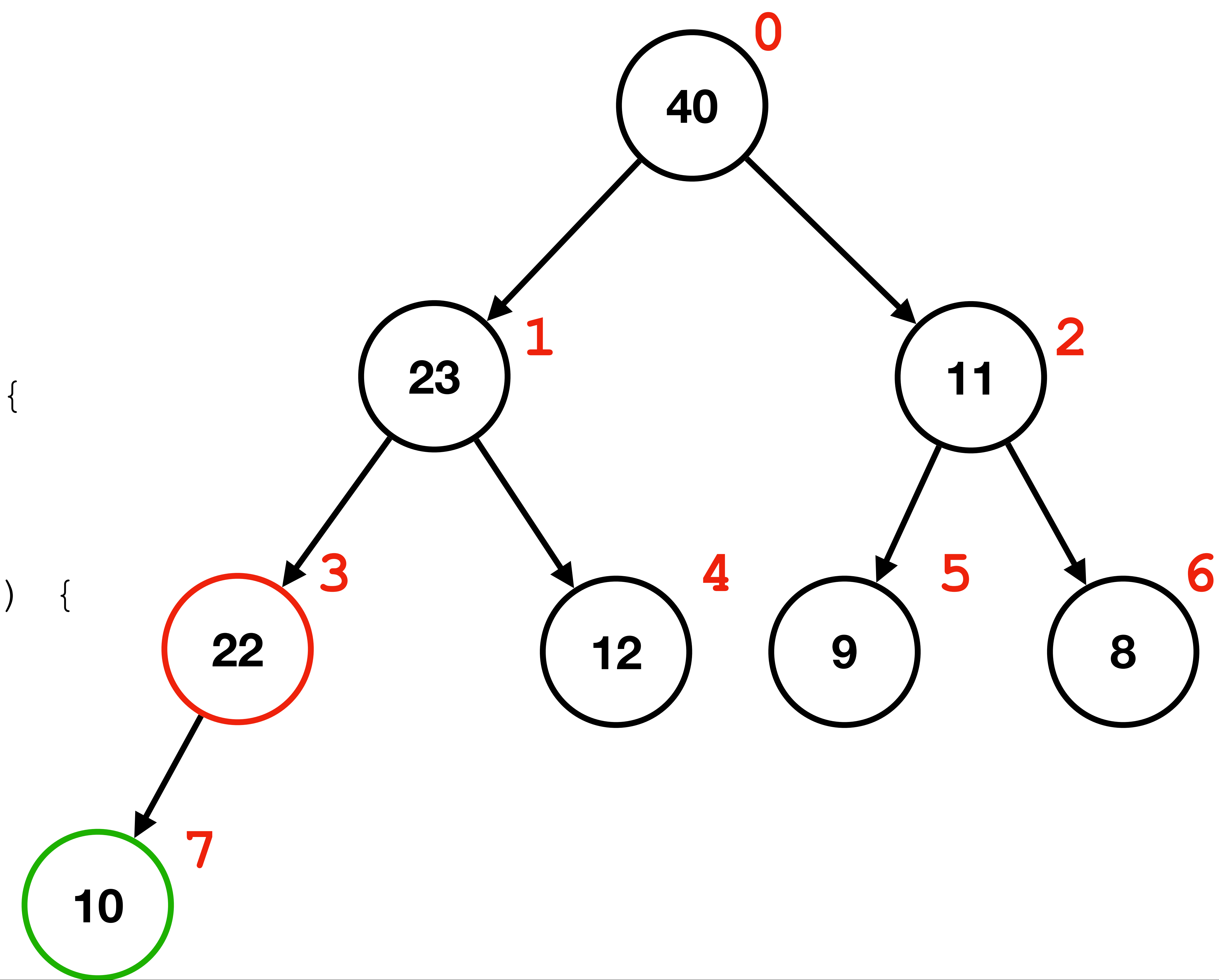
Bubble Down

```
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;
        }
    }
}
```



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

Heap

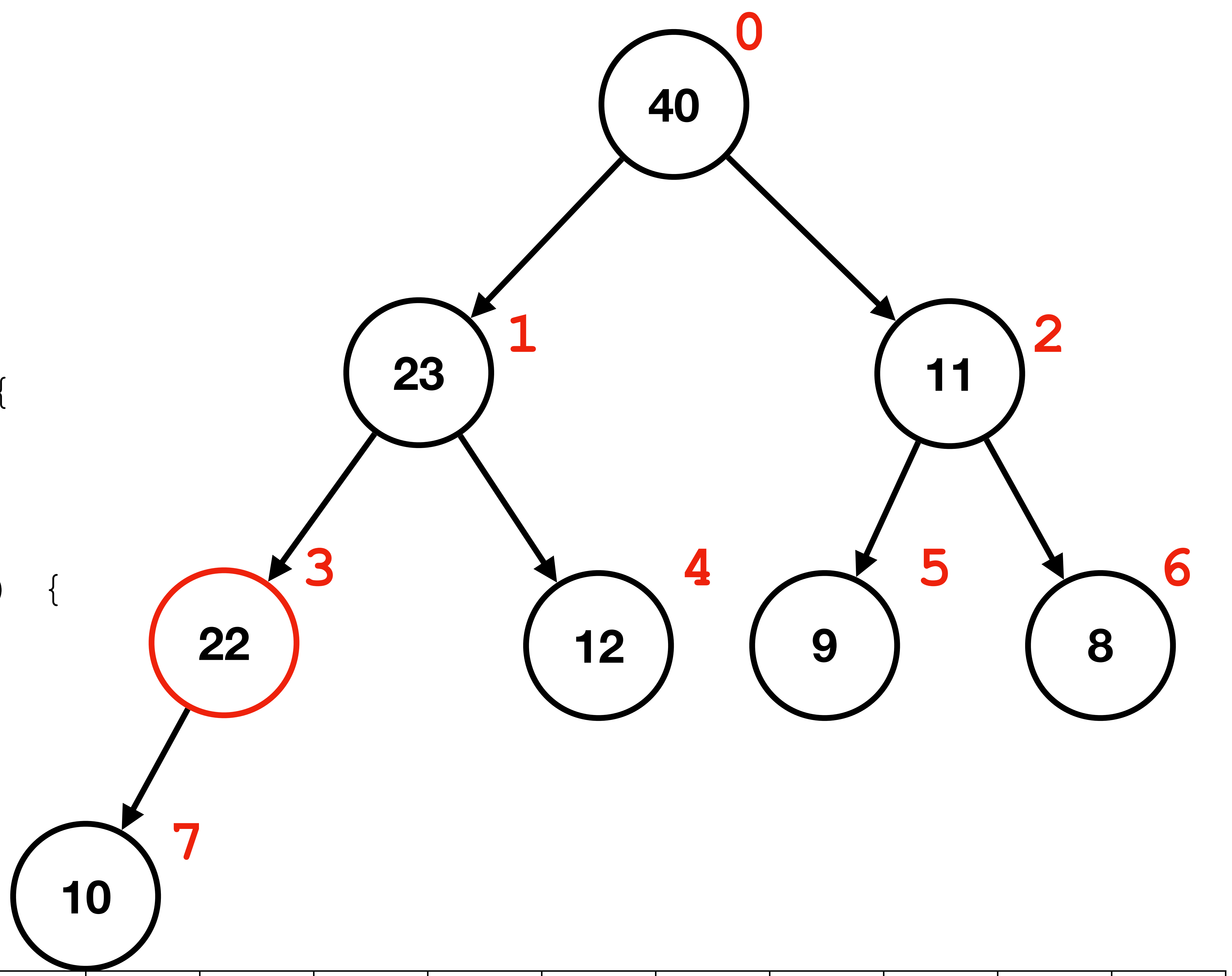
Bubble Down

```
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;
        }
    }
}
```



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

Heap

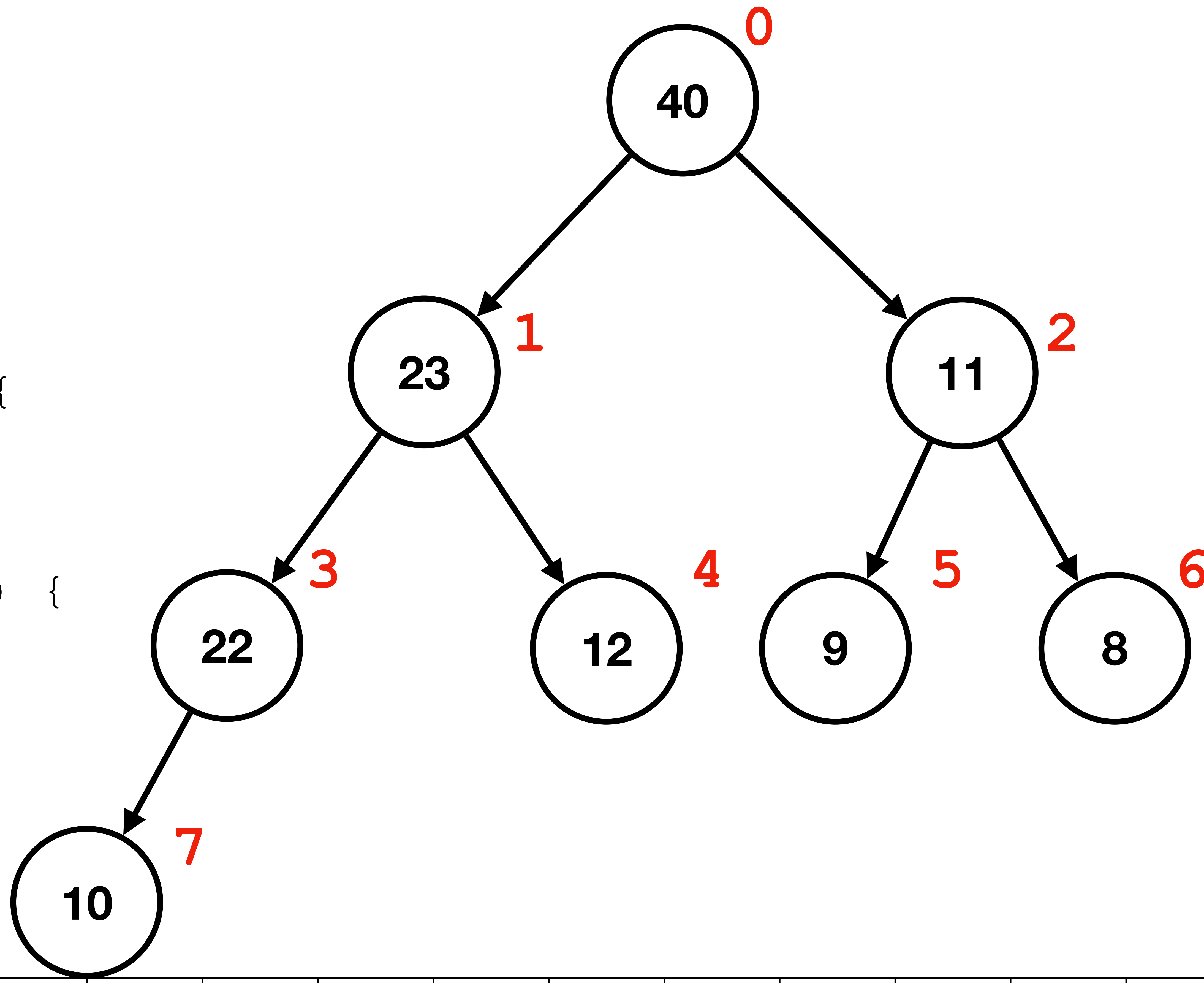
Bubble Down

```
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;
        }
    }
}
```



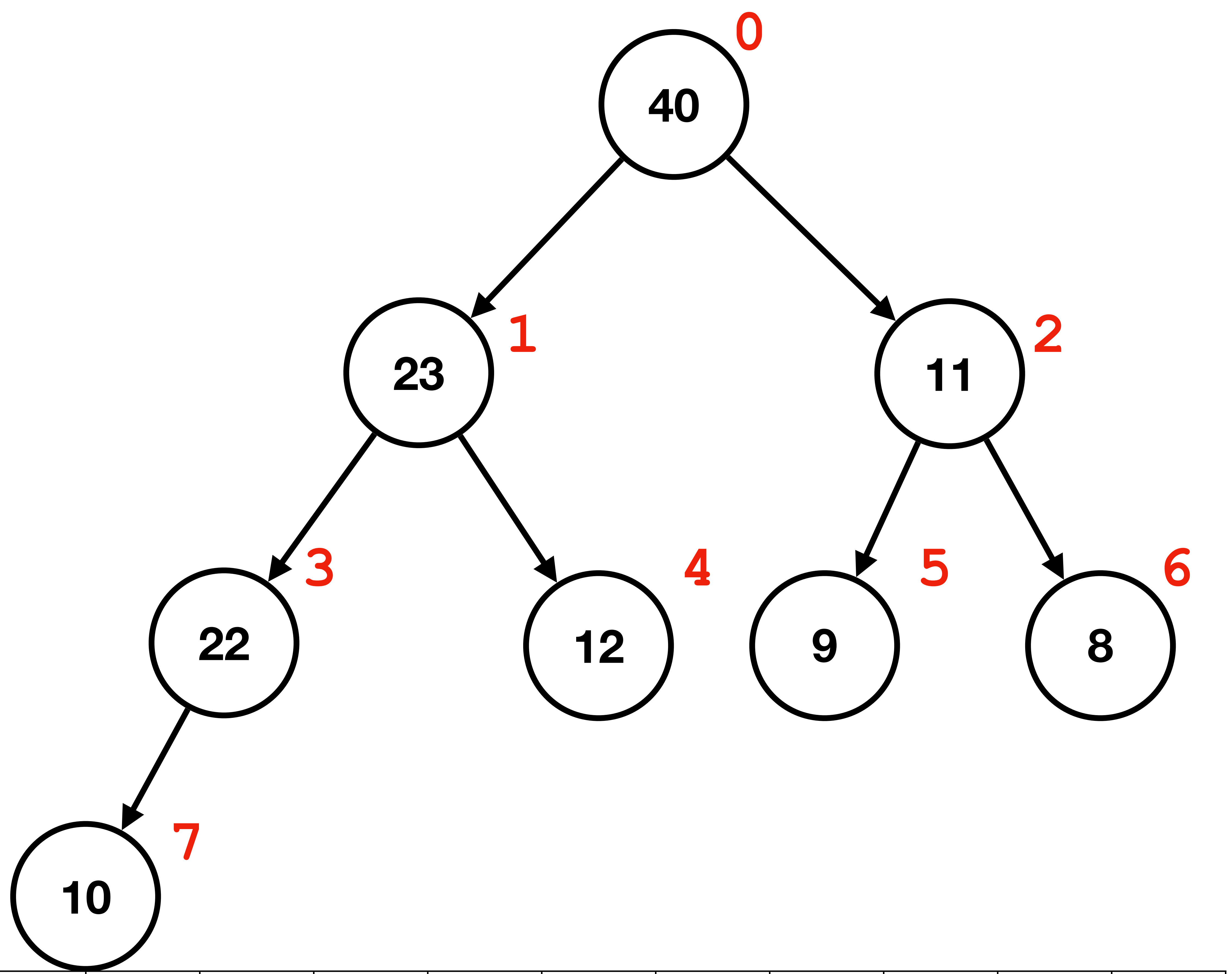
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
40	23	11	22	12	9	8	10			

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);
}
```



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

Priority Queue

Implementations

	`insert`	`get_top`	`remove_top`
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

- 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!

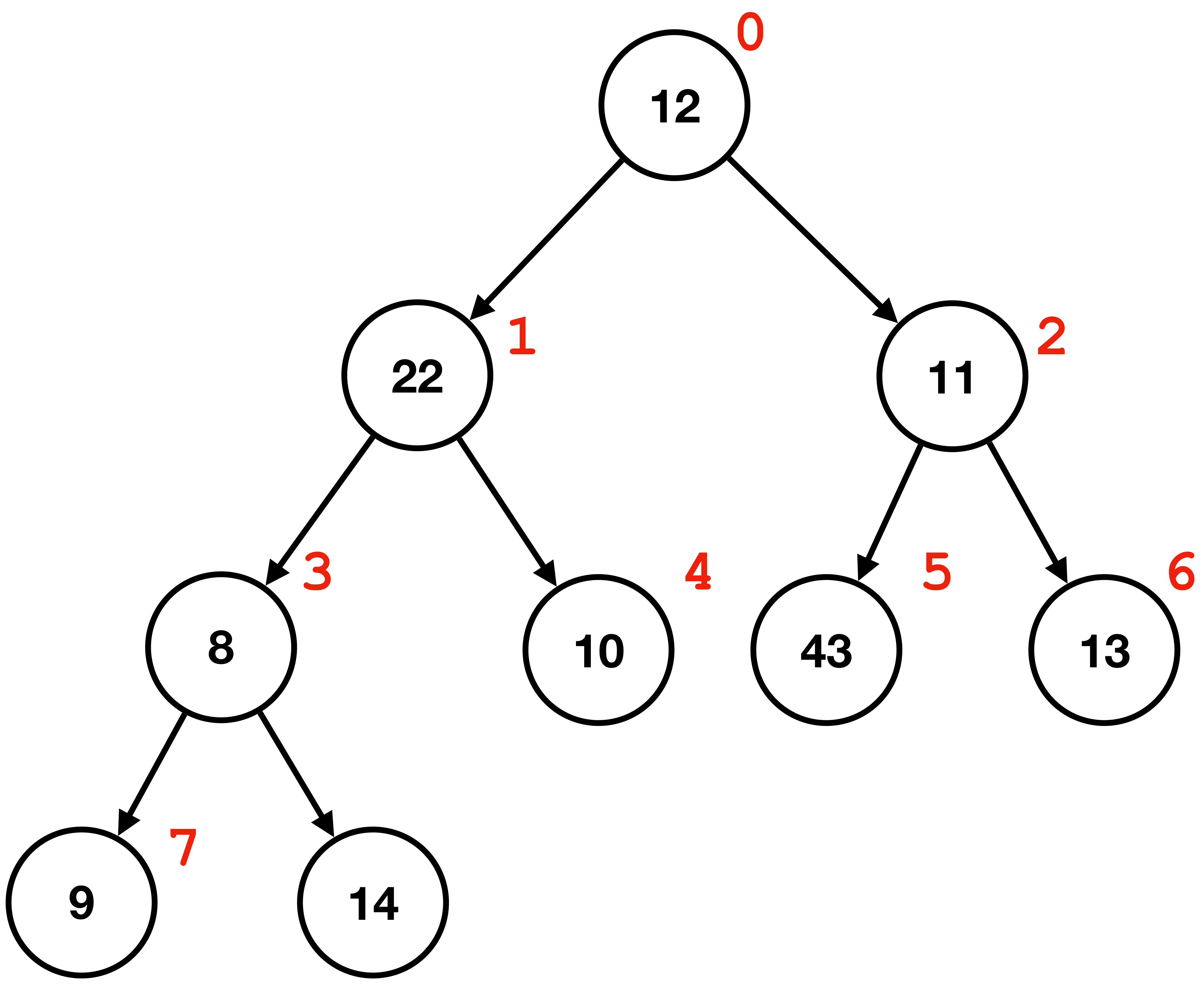
Heap

Heapify

- Given an unsorted array:

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

- What if we just call this a *heap?*
  - <u>*Shape property*</u> is satisfied
  - <u>*Value property*</u> is not, but...
    - All leaves satisfy the value property (they have no children)



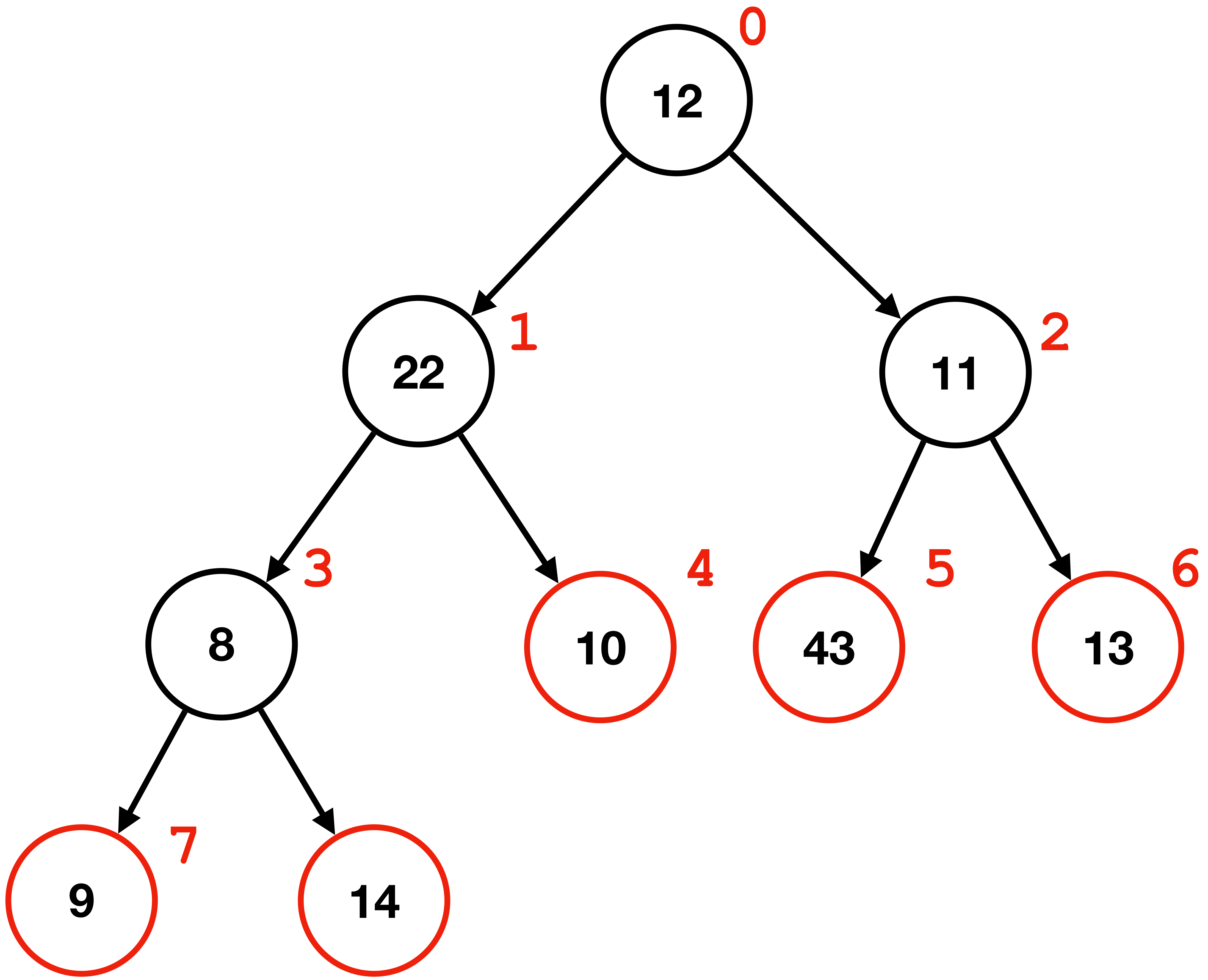
Heap

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

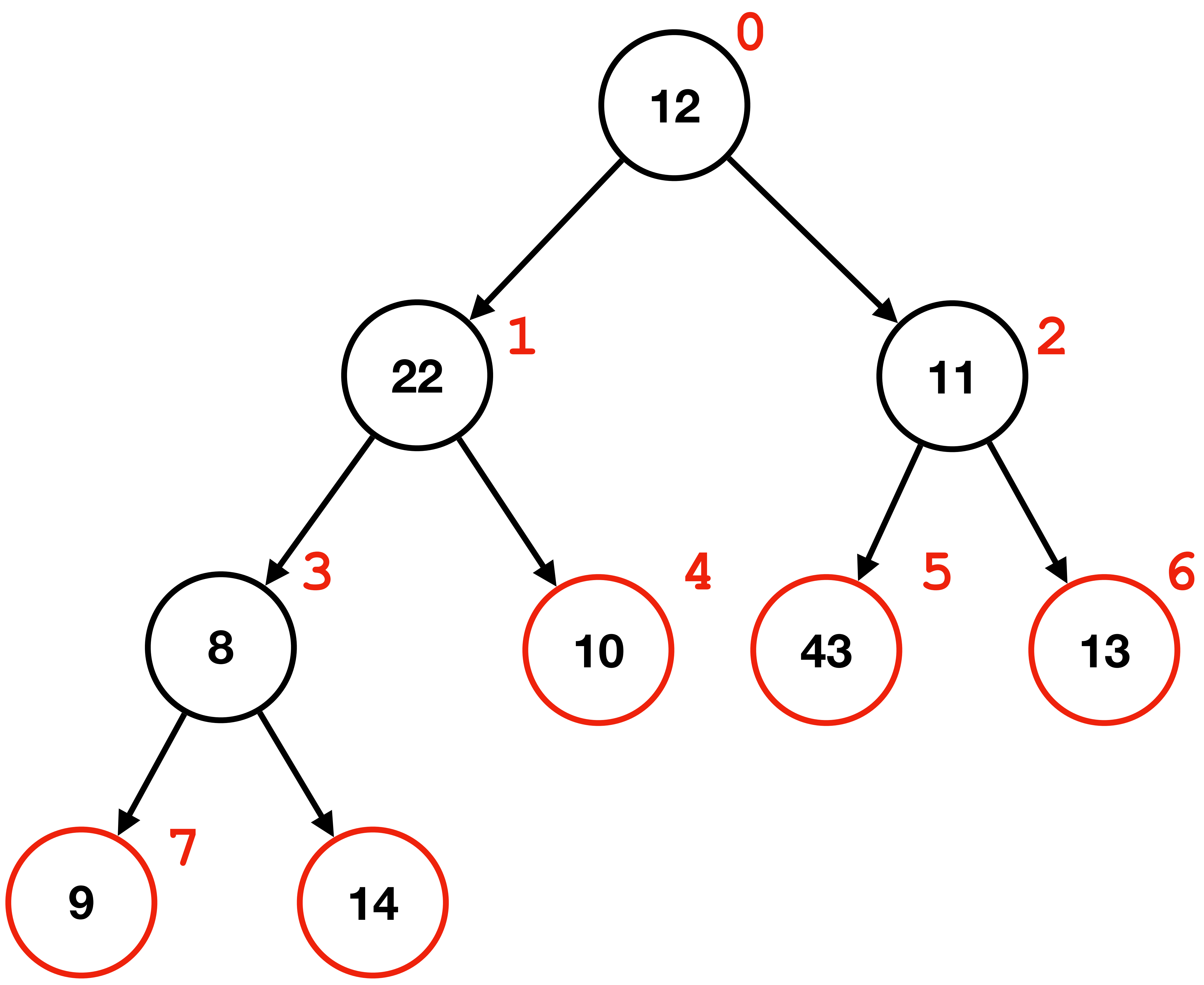
- Given an unsorted array:

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

- What if we just call this a *heap?*

- *Insight*: To make non-leaves satisfy the value property, perform bubble-down on every non-leaf node, from bottom up.

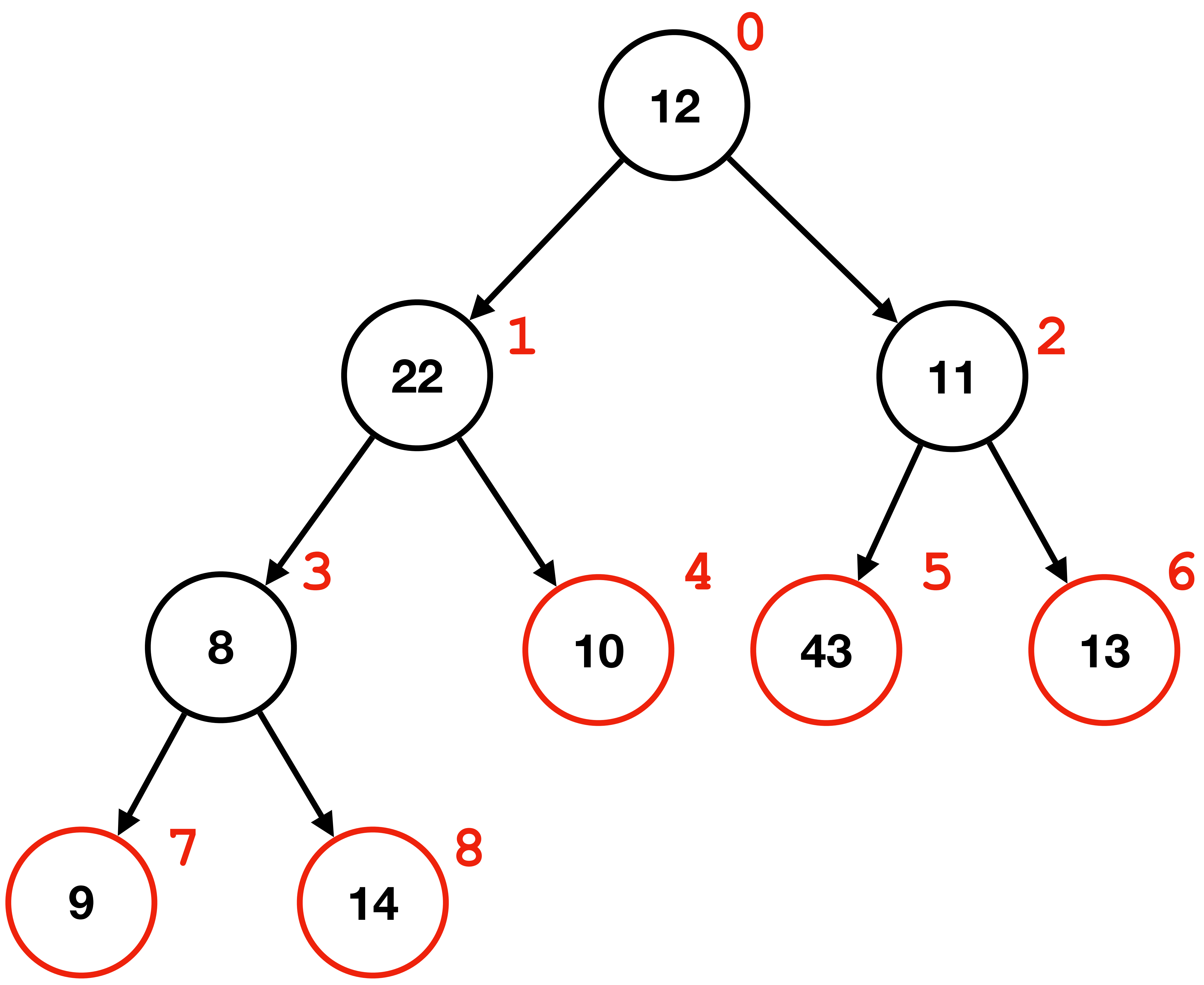
- Bubble down assumes that both subtrees are valid heaps.



Heap

Heapify

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



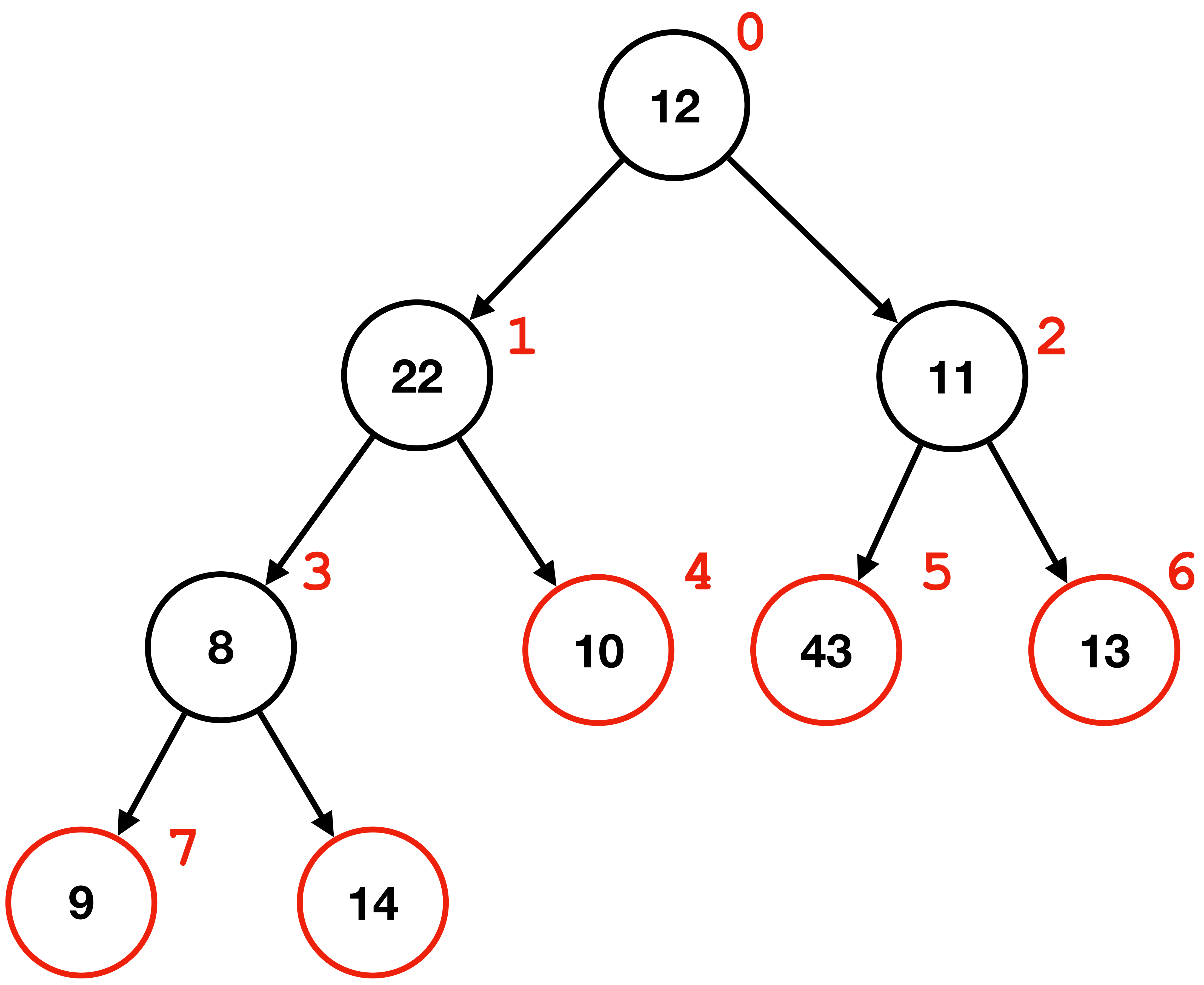
Heap

Heapify

- Given an unsorted array:

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

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



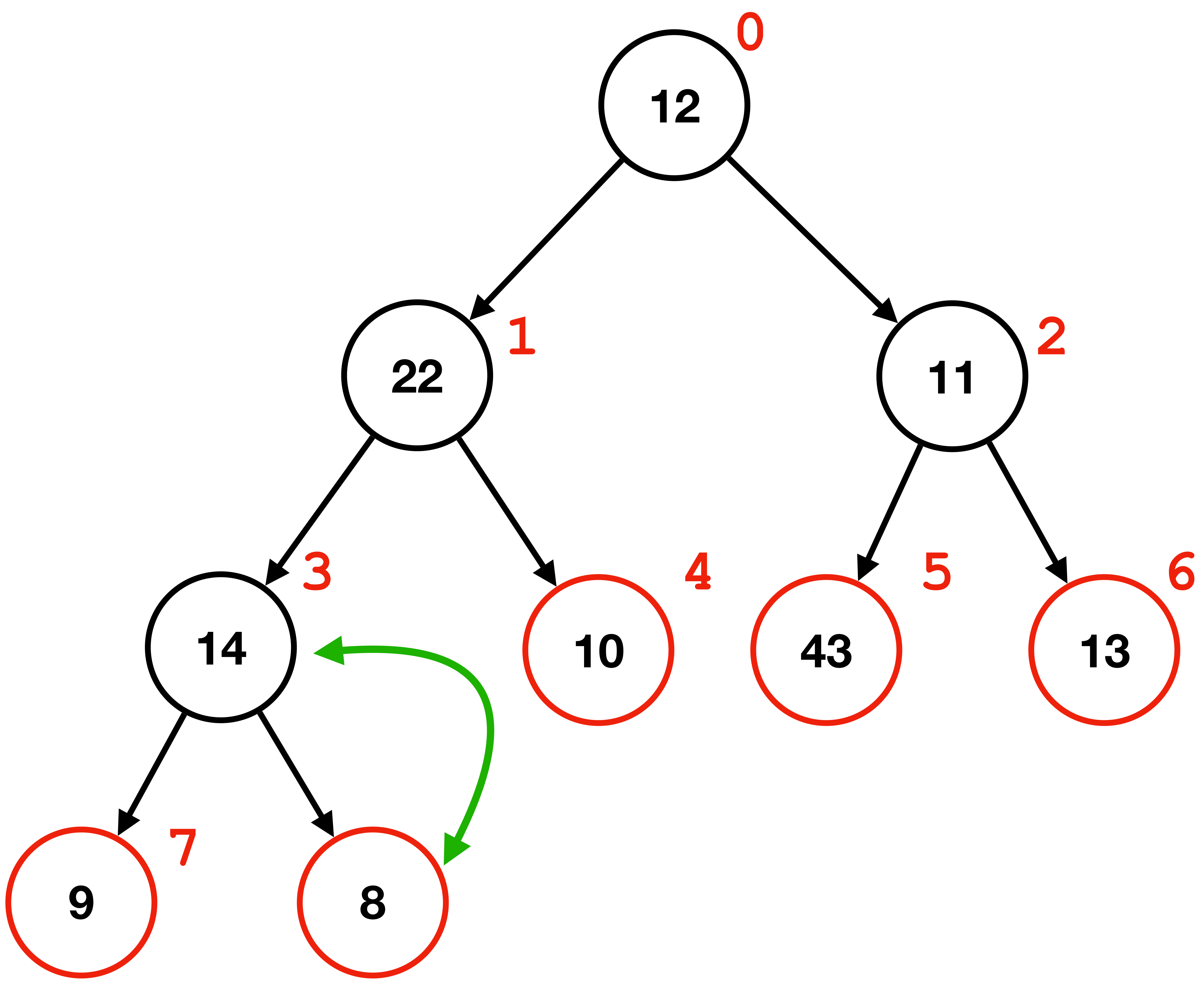
Heap

Heapify

- Given an unsorted array:

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

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



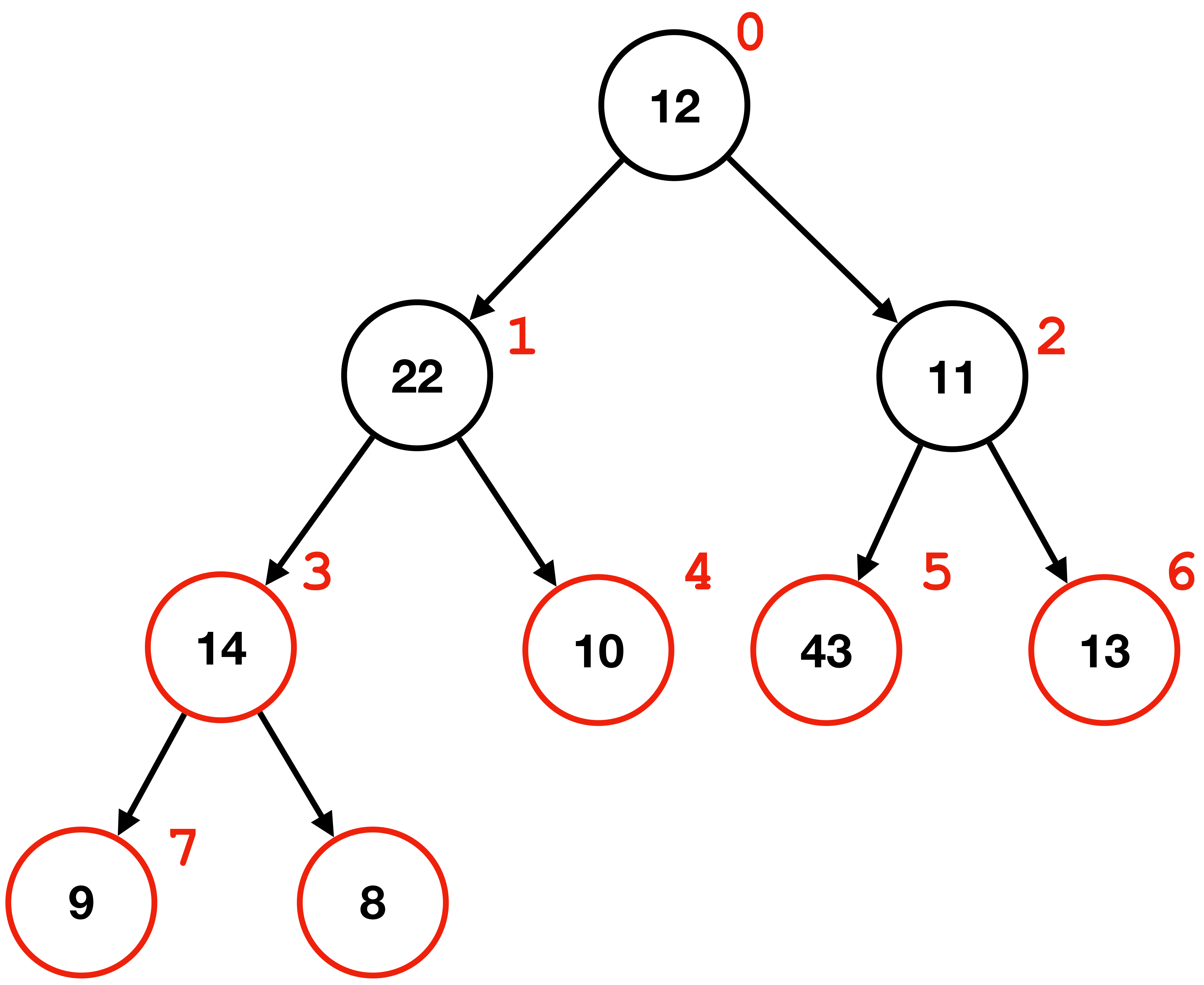
Heap

Heapify

- Given an unsorted array:

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

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



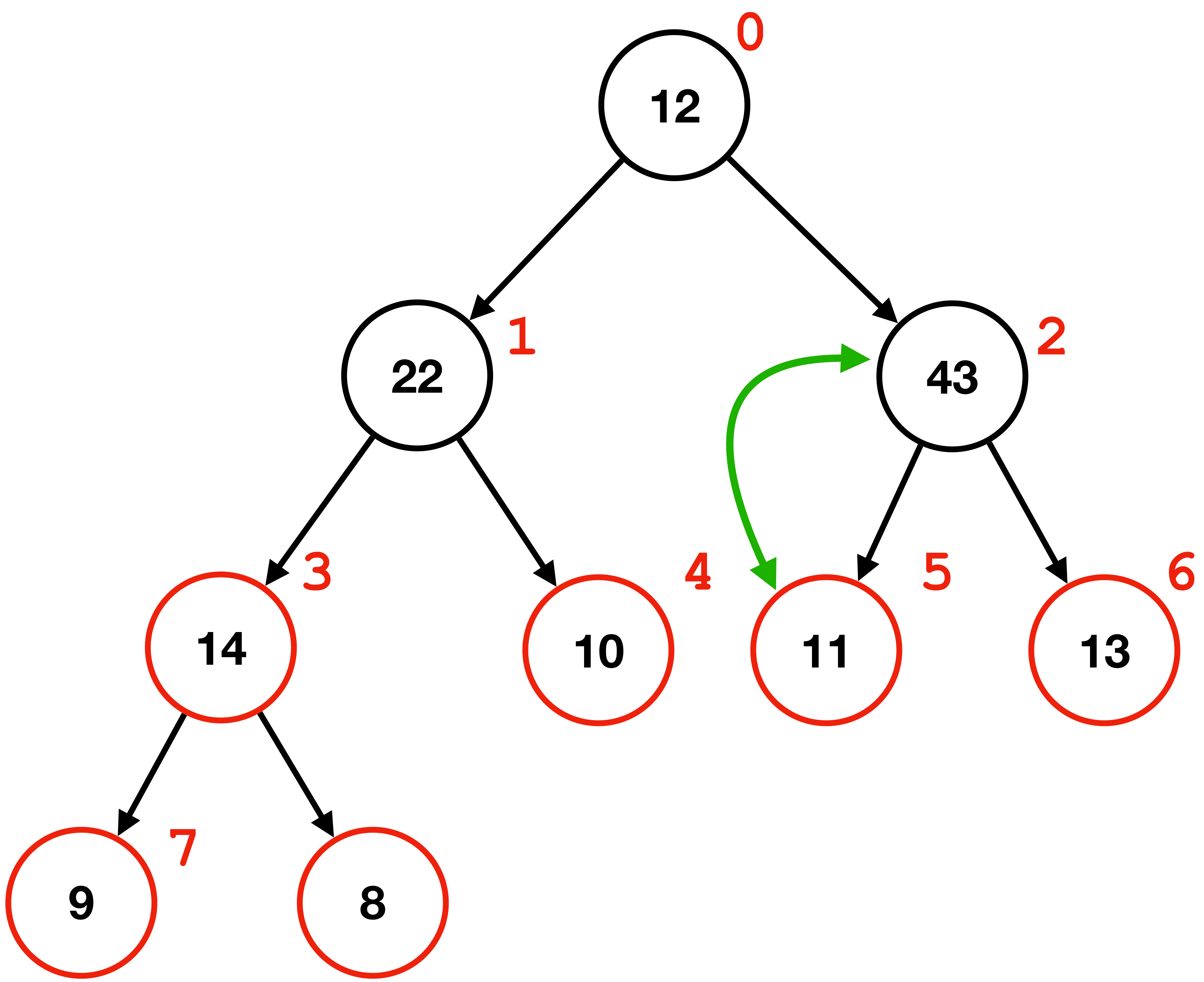
Heap

Heapify

- Given an unsorted array:

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

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



Heap

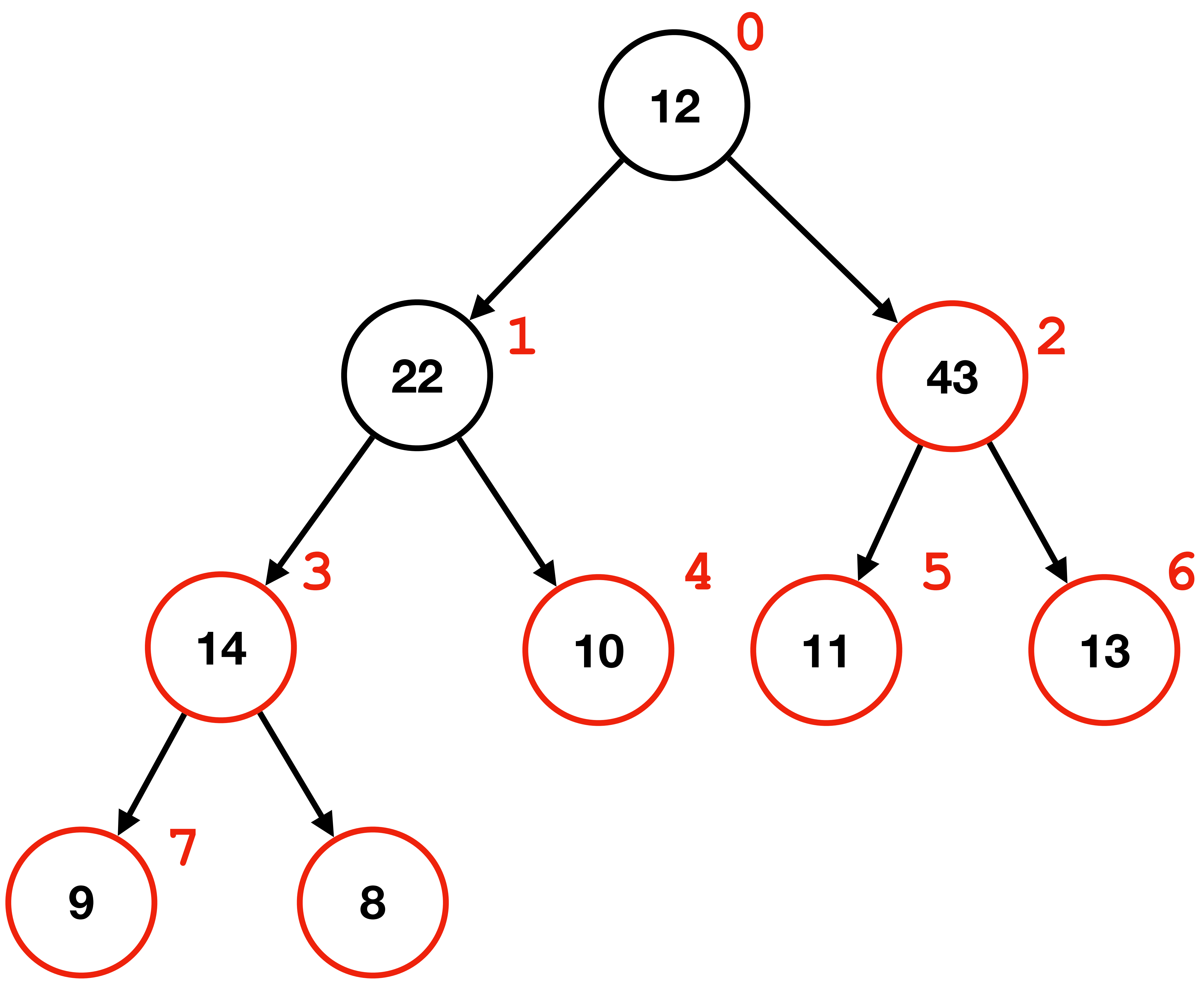
Heapify

- Given an unsorted array:

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

1. Call this a heap.

2. Starting at index `heap_size / 2 - 1` and moving backwards: perform `bubble_down` on every non-leaf node



Heap

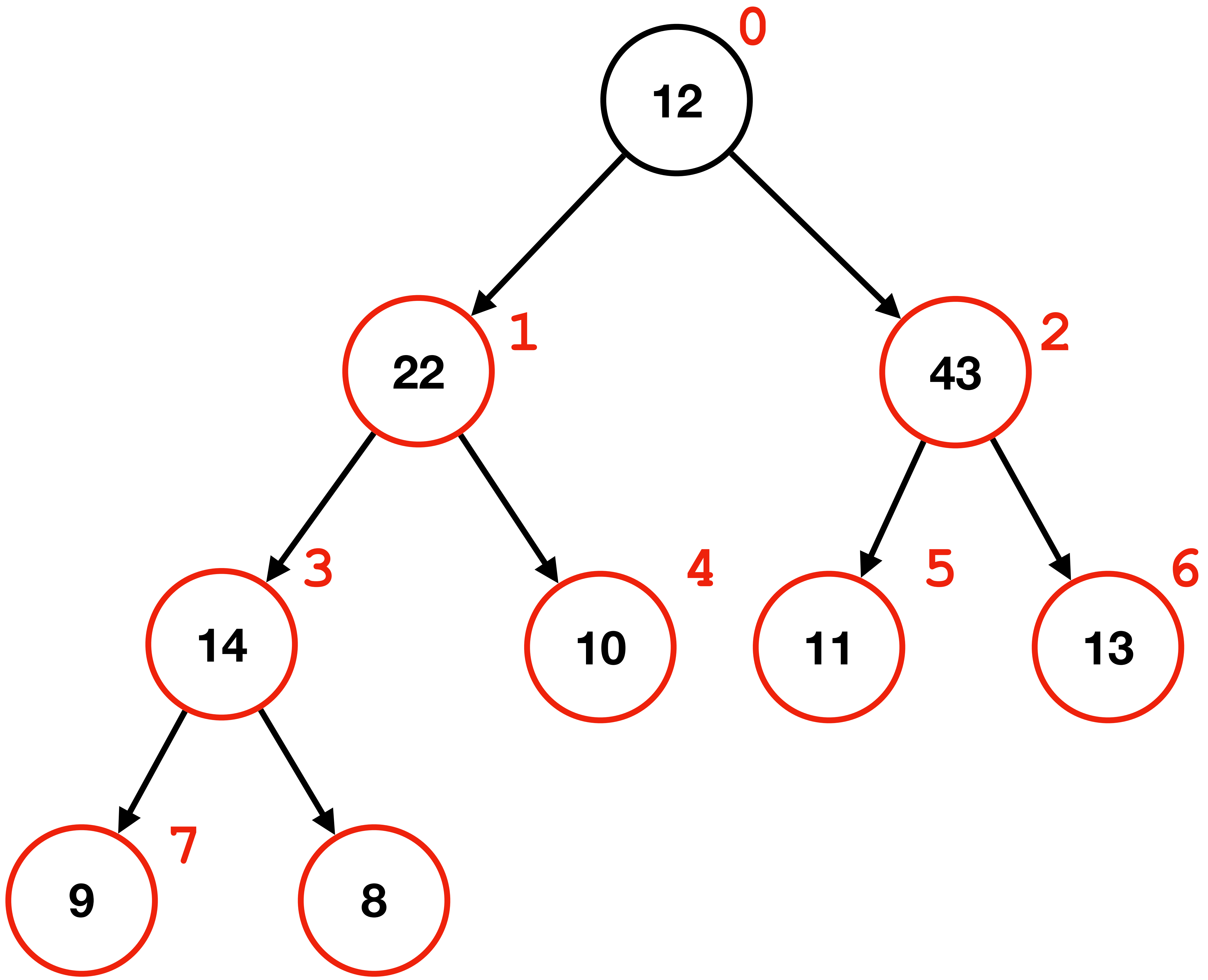
Heapify

- Given an unsorted array:

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

1. Call this a heap.

2. Starting at index `heap_size / 2 - 1` and moving backwards: perform `bubble_down` on every non-leaf node



Heap

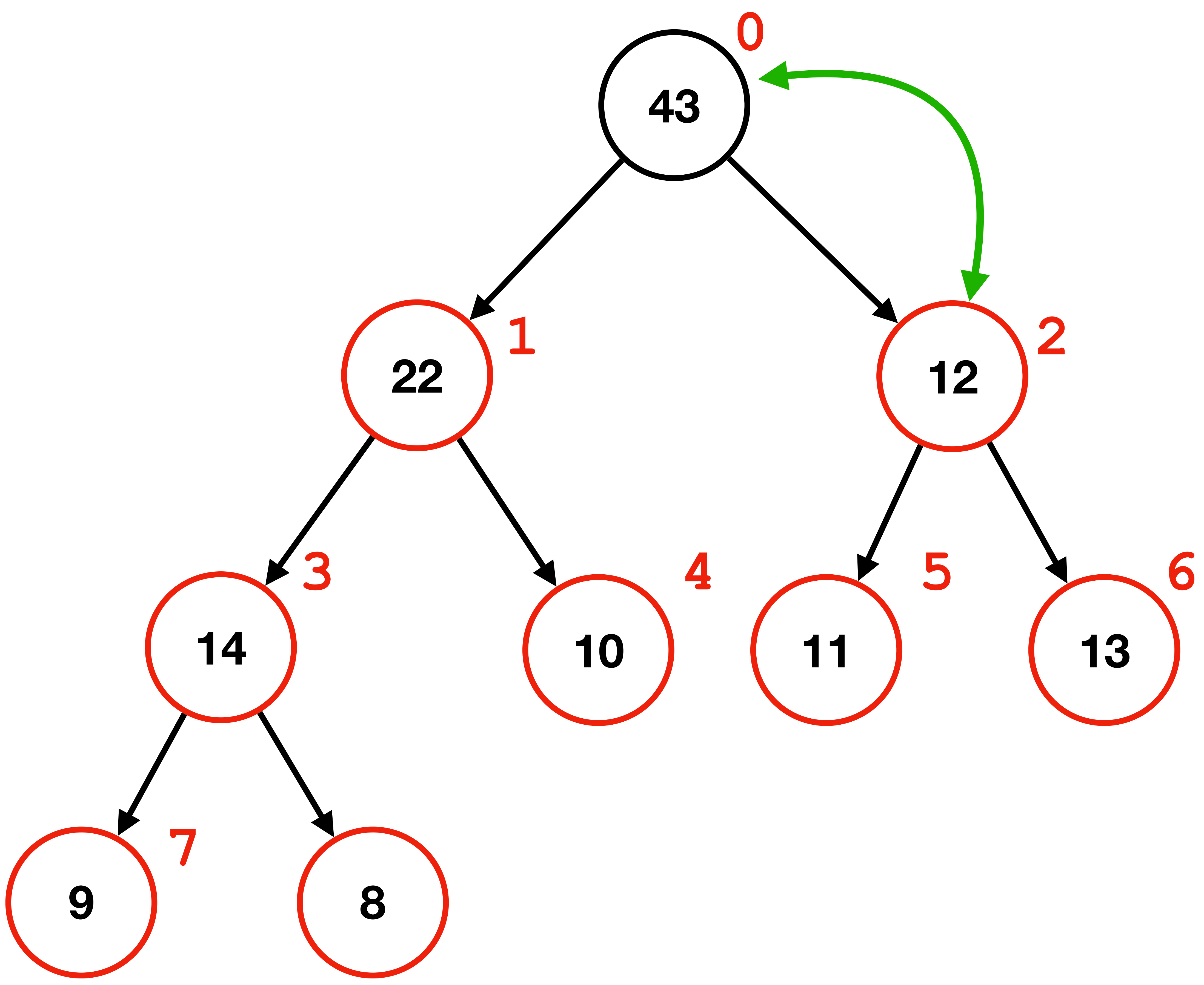
Heapify

- Given an unsorted array:

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

1. Call this a heap.

2. Starting at index `heap_size / 2 - 1` and moving backwards: perform `bubble_down` on every non-leaf node



Heap

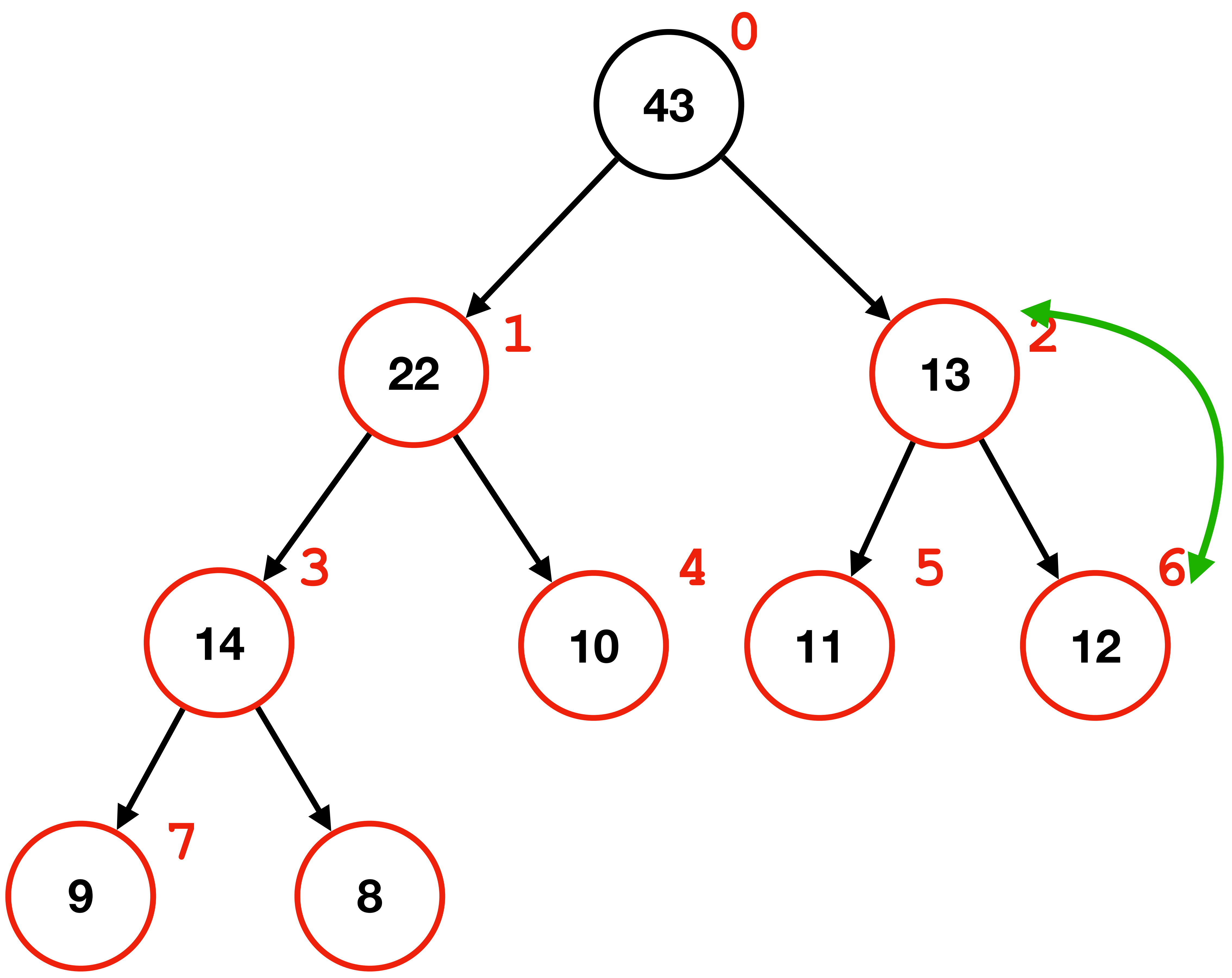
Heapify

- Given an unsorted array:

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

1. Call this a heap.

2. Starting at index `heap_size / 2 - 1` and moving backwards: perform `bubble_down` on every non-leaf node



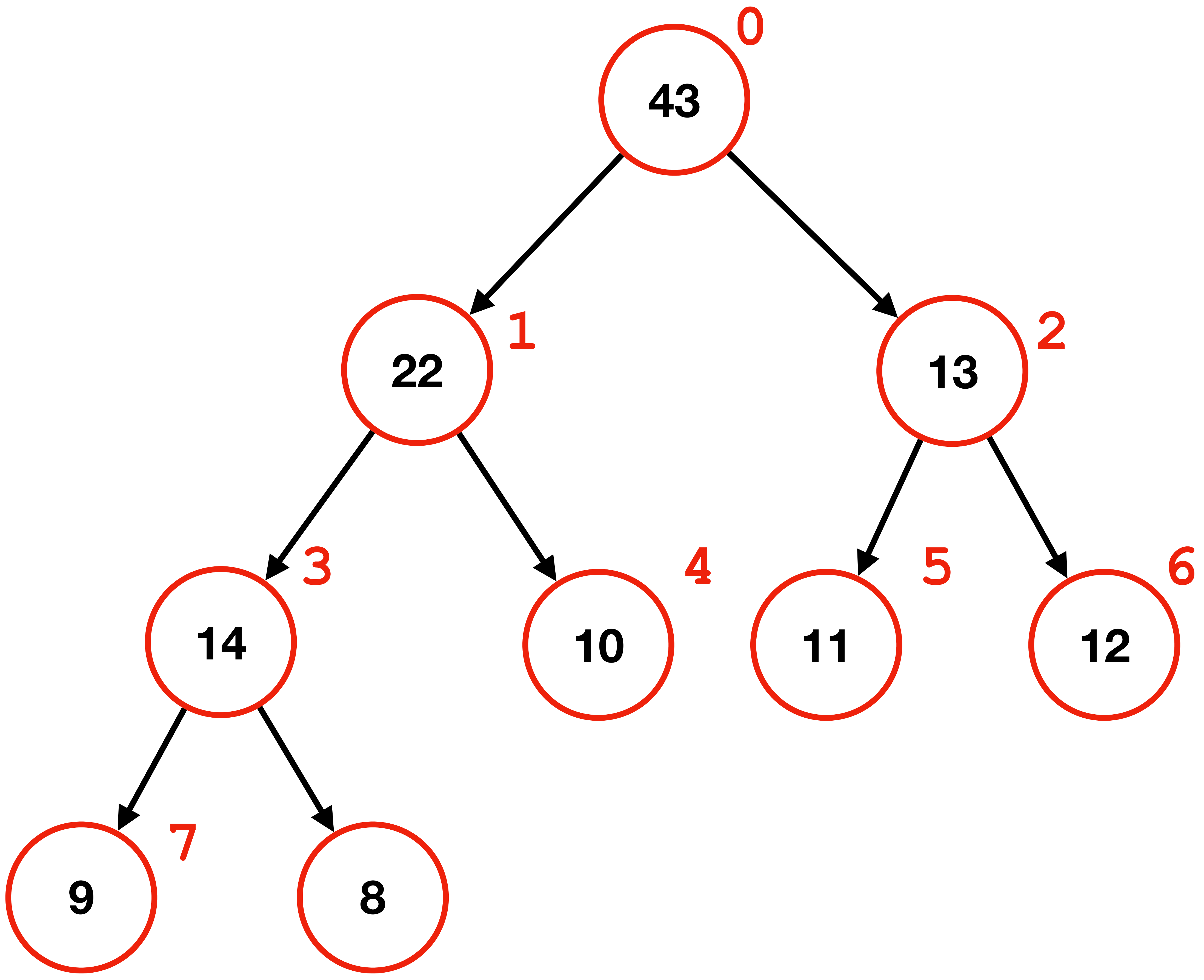
Heap

Heapify

- Given an unsorted array:

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

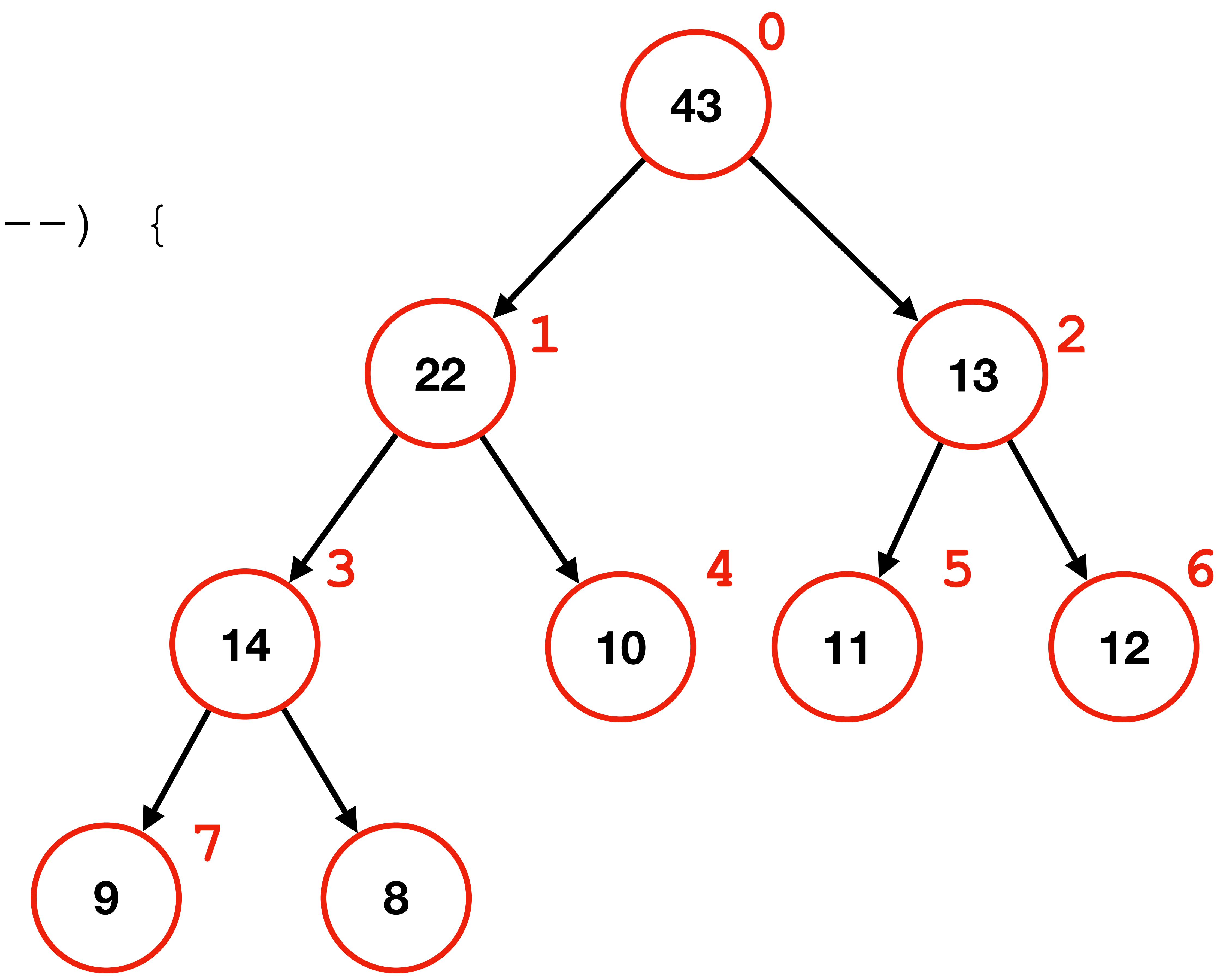
1. Call this a heap.
2. Starting at index `heap_size / 2 - 1` and moving backwards: perform `bubble_down` on every non-leaf node



Heap

Heapify

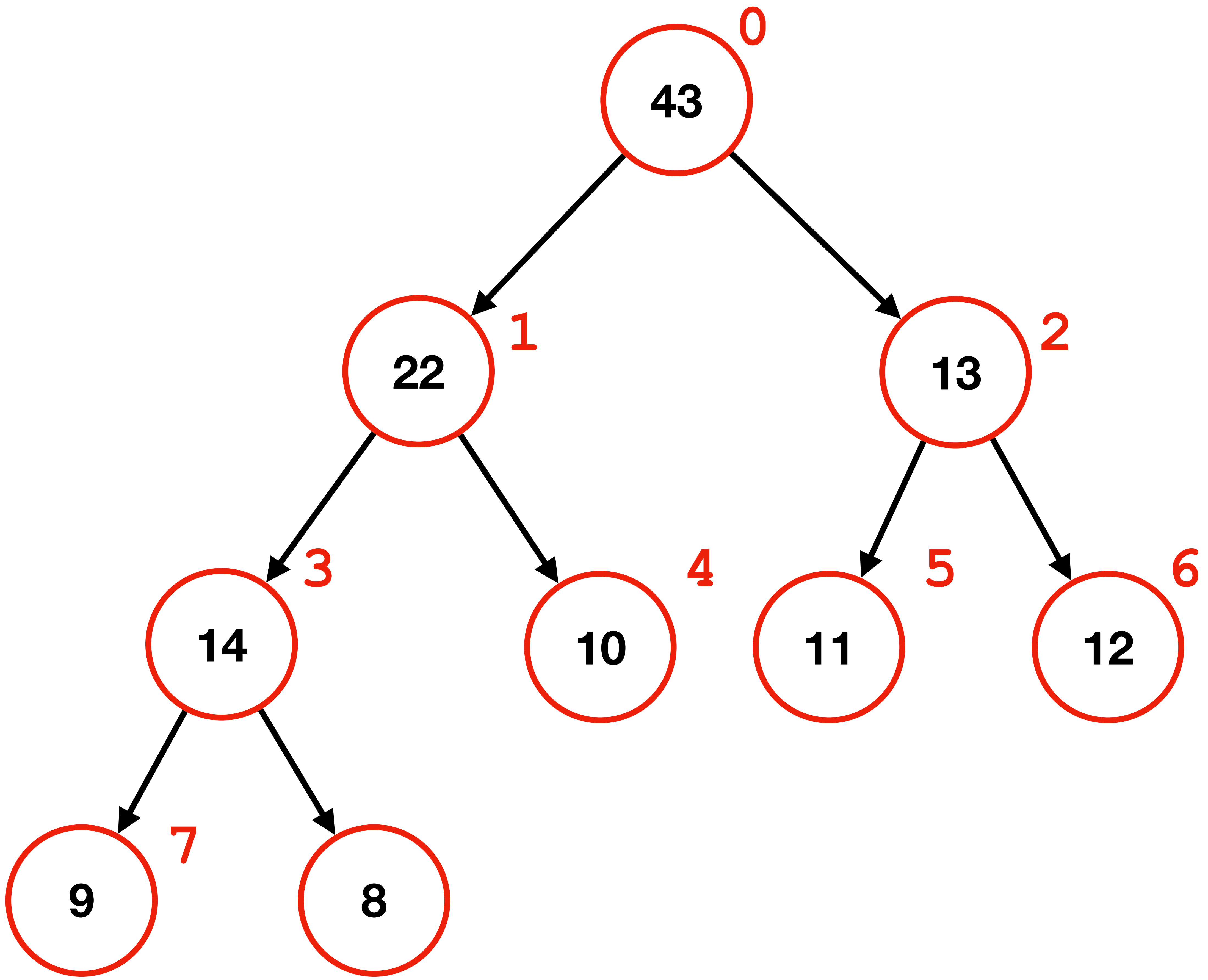
```
void heapify(int *arr, int len)
{
    for (int i = len / 2 - 1; i >= 0; i--) {
        bubble_down(arr, len, i);
    }
}
```



Heap

Heapify Complexity?

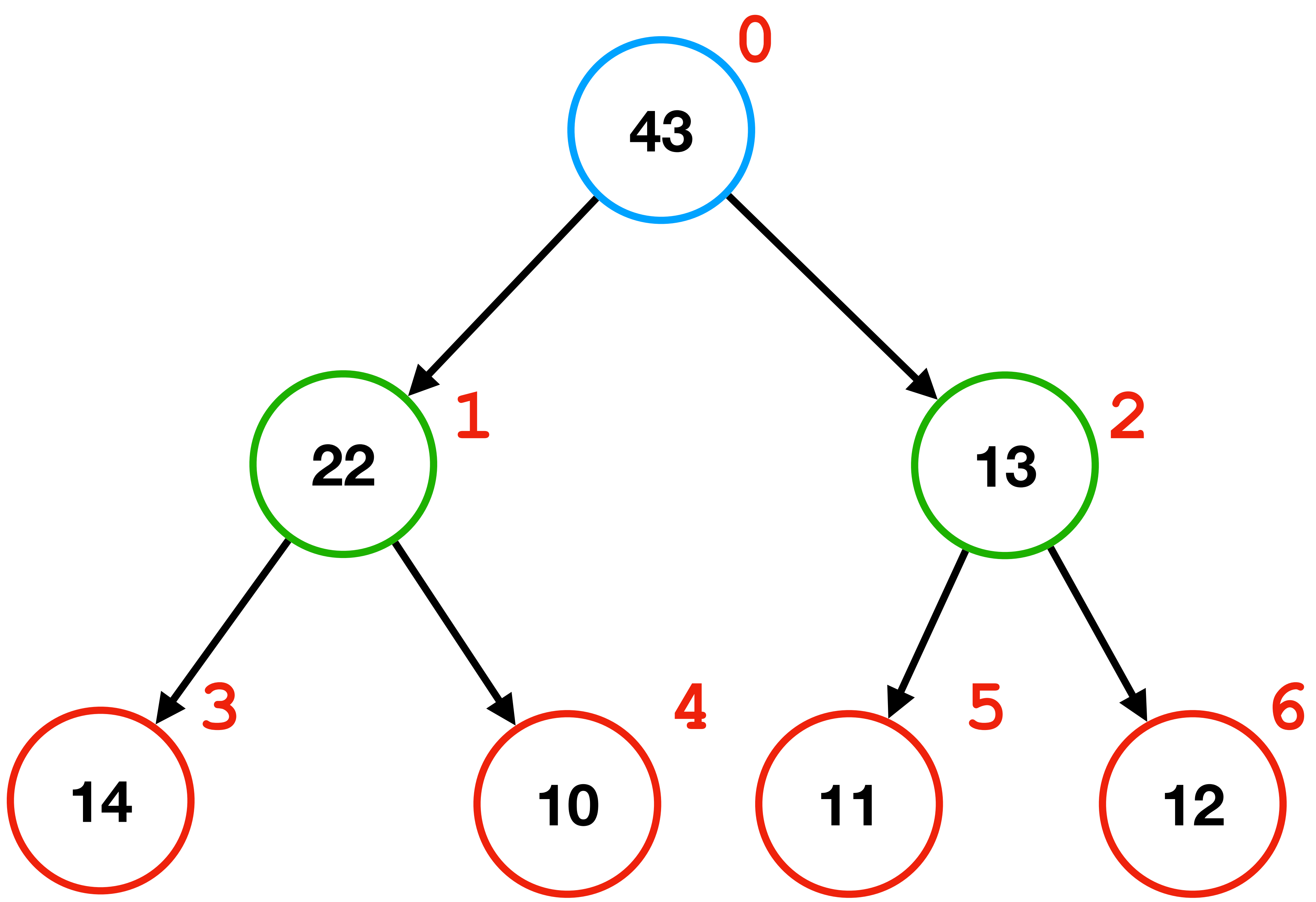
- CMSC 27200.
- Informal analysis:



Heap

Heapify Complexity?

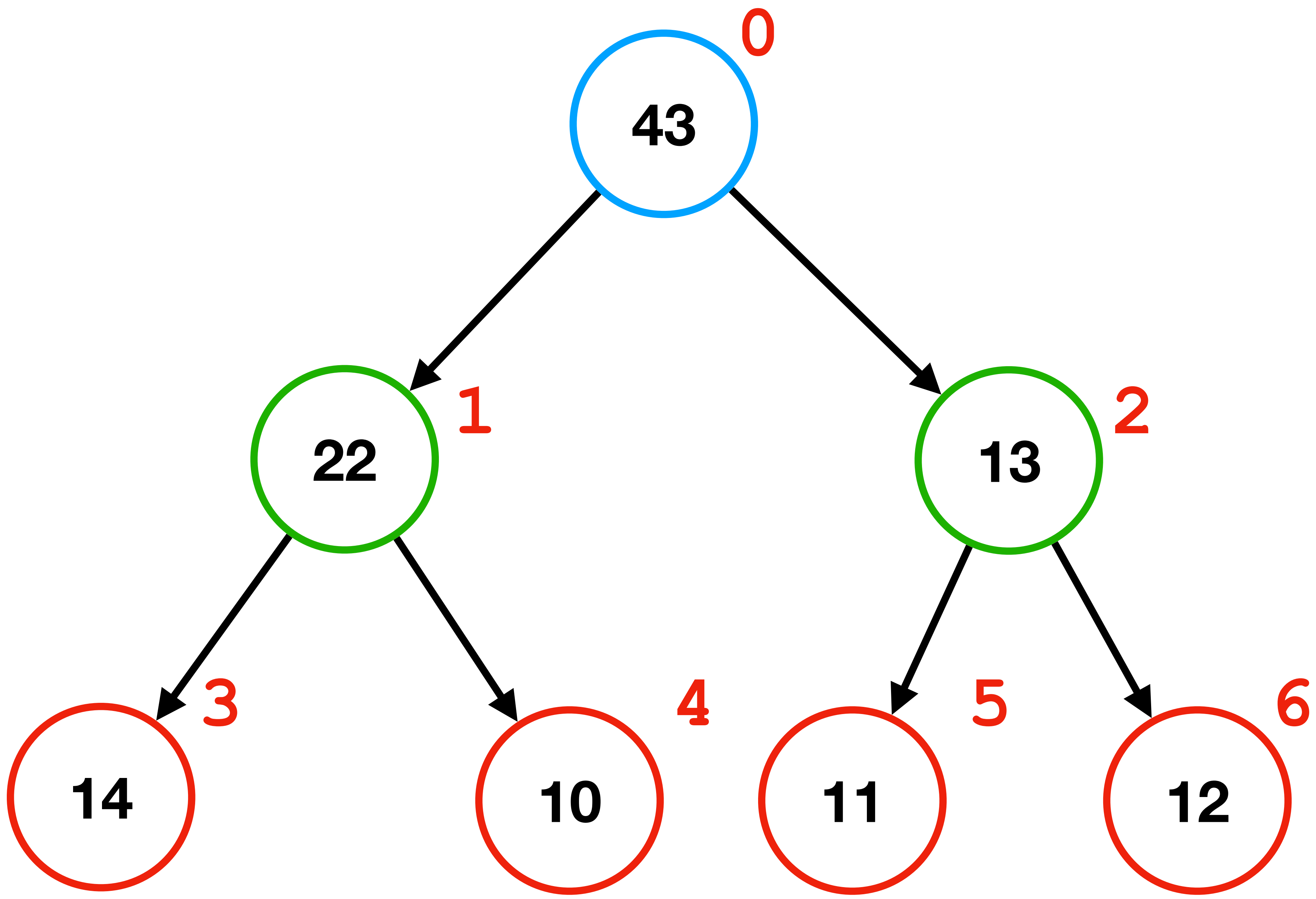
- CMSC 27200.
- Informal analysis:
  - Define *level* to be the maximum distance from the leaves.
    - red: level 0
    - green: level 1
    - blue: level 2



Heap

Heapify Complexity?

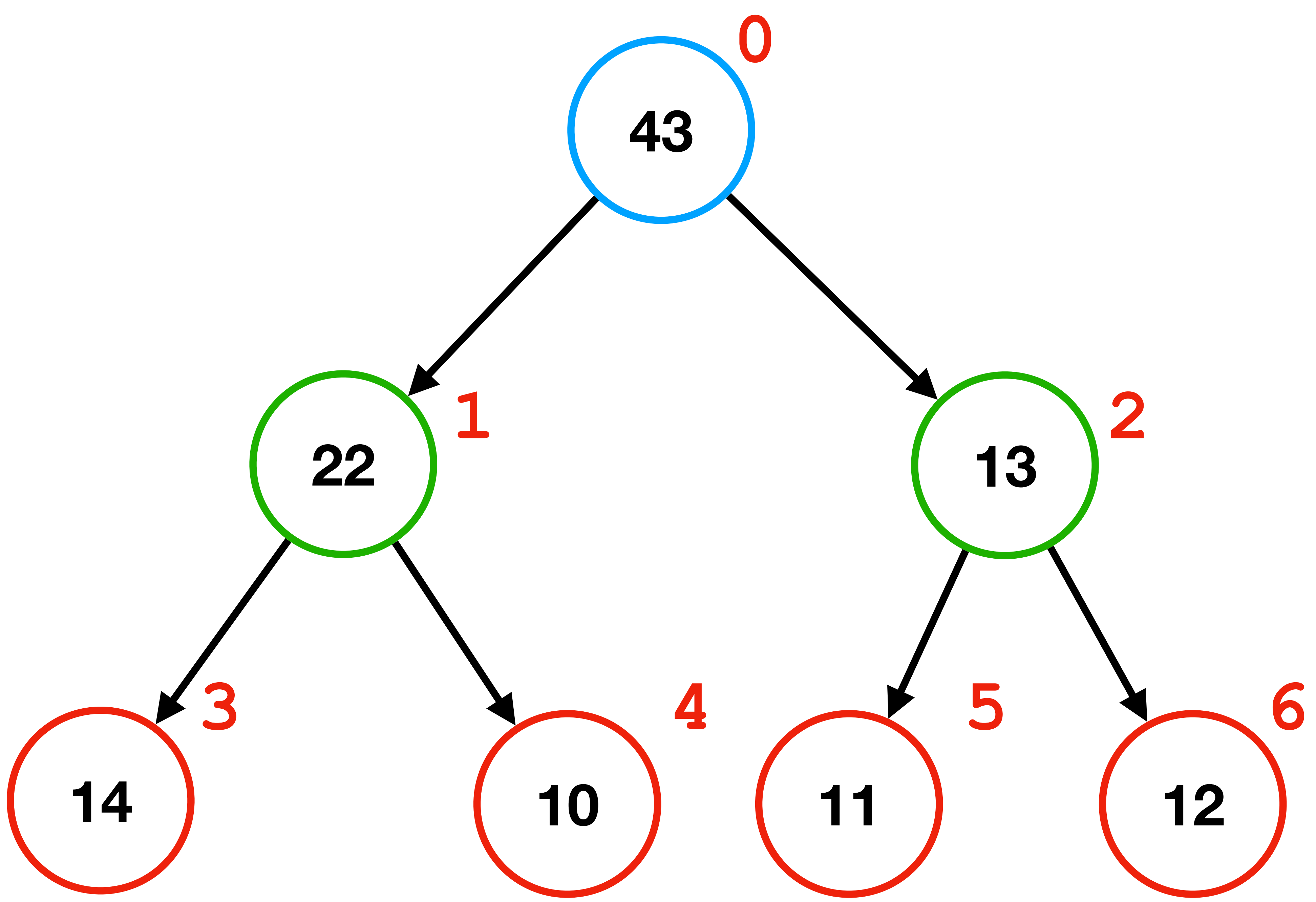
- CMSC 27200.
- Informal analysis:
  - Define *level* to be the maximum distance from the leaves.
  - If n is `heap_size`, there are at most...
    - $\lceil n/2^1 \rceil$ level 0 nodes (7 / 2 $\approx$ 4)
    - $\lceil n/2^2 \rceil$ level 1 nodes (7 / 4 $\approx$ 2)
    - $\lceil n/2^3 \rceil$ level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



Heap

Heapify Complexity?

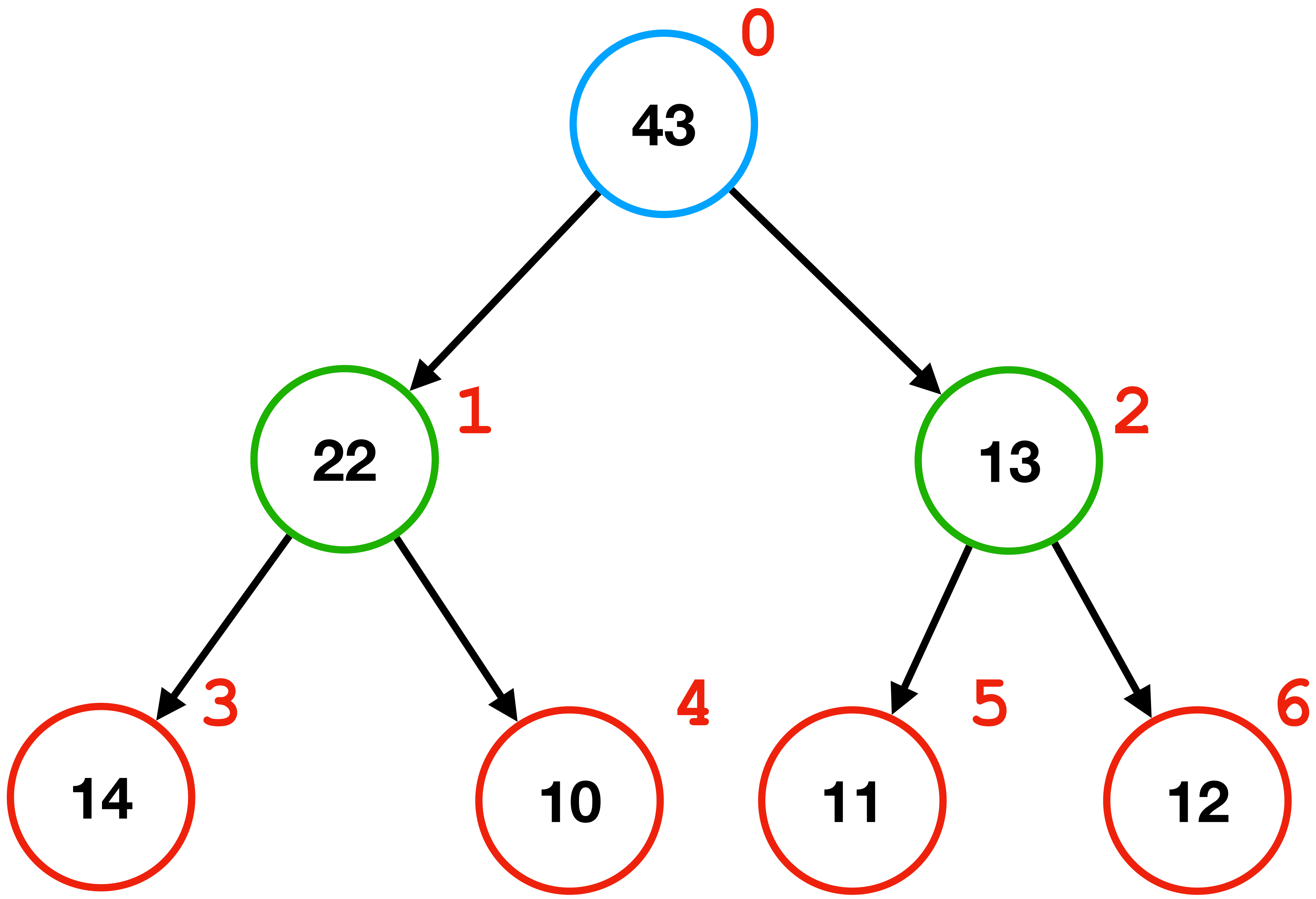
- CMSC 27200.
- Informal analysis:
  - Define *<u>level</u>* to be the maximum distance from the leaves.
  - If n is `heap_size`, there are at most...
    - $n/2^{l+1}$ nodes for level l



Heap

Heapify Complexity?

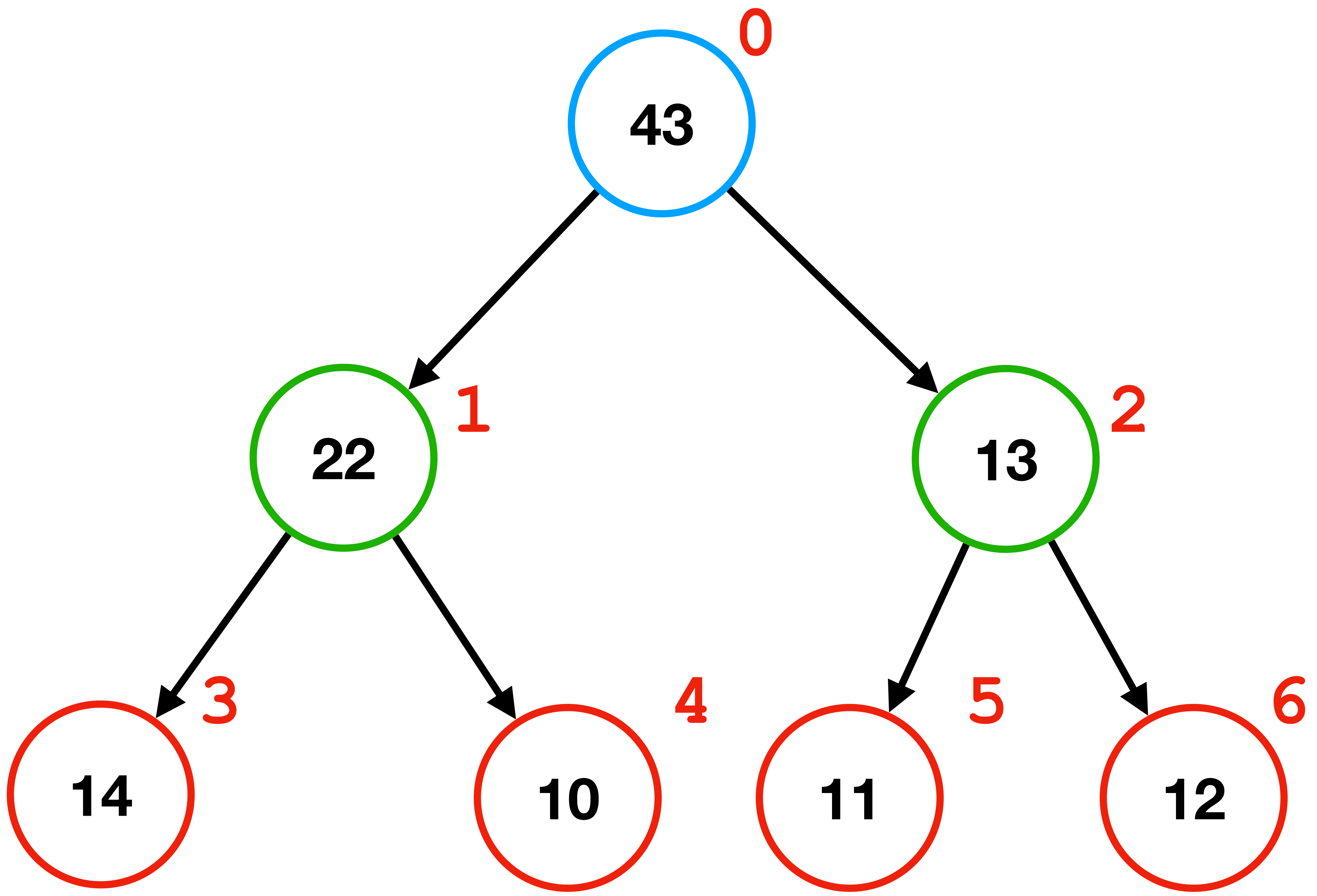
- CMSC 27200.
- Informal analysis:
  - Define *level* to be the maximum distance from the leaves.
  - If n is `heap_size`, there are at most...
    - $n/2^{l+1}$ nodes for level l
    - For nodes with level l, there can be at most l swaps bubbling down.



Heap

Heapify Complexity?

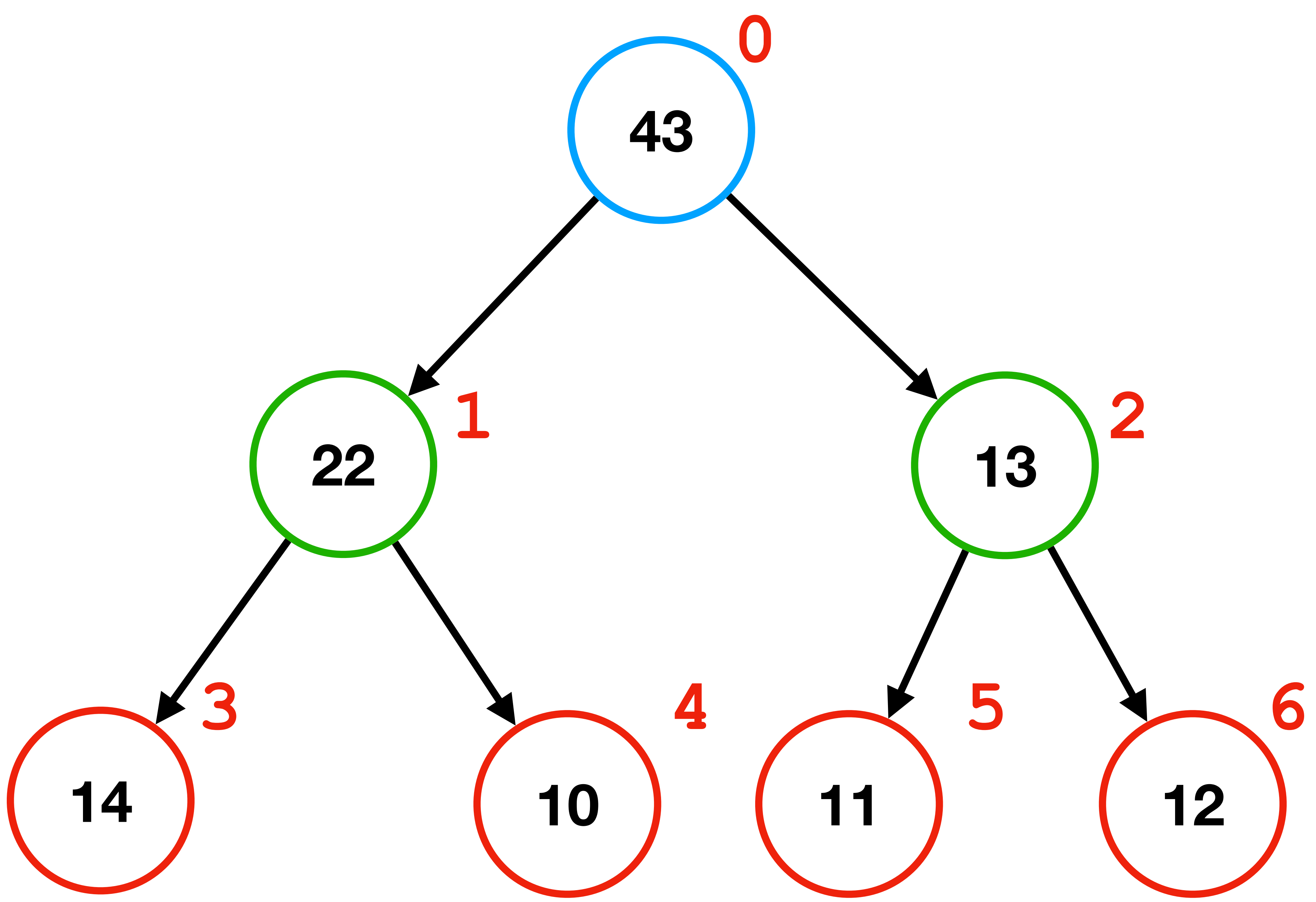
- CMSC 27200.
- Informal analysis:
  - Total works: $\sum_{l=0}^{\log n} \frac{n}{2^{l+1}} \cdot l$
  - $n\left(\sum_{l=0}^{\log n} \frac{l}{2^{l+1}}\right)$
  - $\sum_{l=0}^{\log n} \frac{l}{2^{l+1}} < \sum_{l=0}^{\infty} \frac{l}{2^{l+1}} = 1$



Heap

Heapify Complexity?

- CMSC 27200.
- Informal analysis:
    - Total works: $\sum_{l=0}^{\log n} \frac{n}{2^{l+1}} \cdot l < n = O(n)$
    - 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--) {   Heapify
        bubble_down(arr, len, i);
    }

    for (int i = len - 1; i >= 1; i--) {      Extract top and put at the end
        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: $O(n \log n)$
- Space complexity: in-place

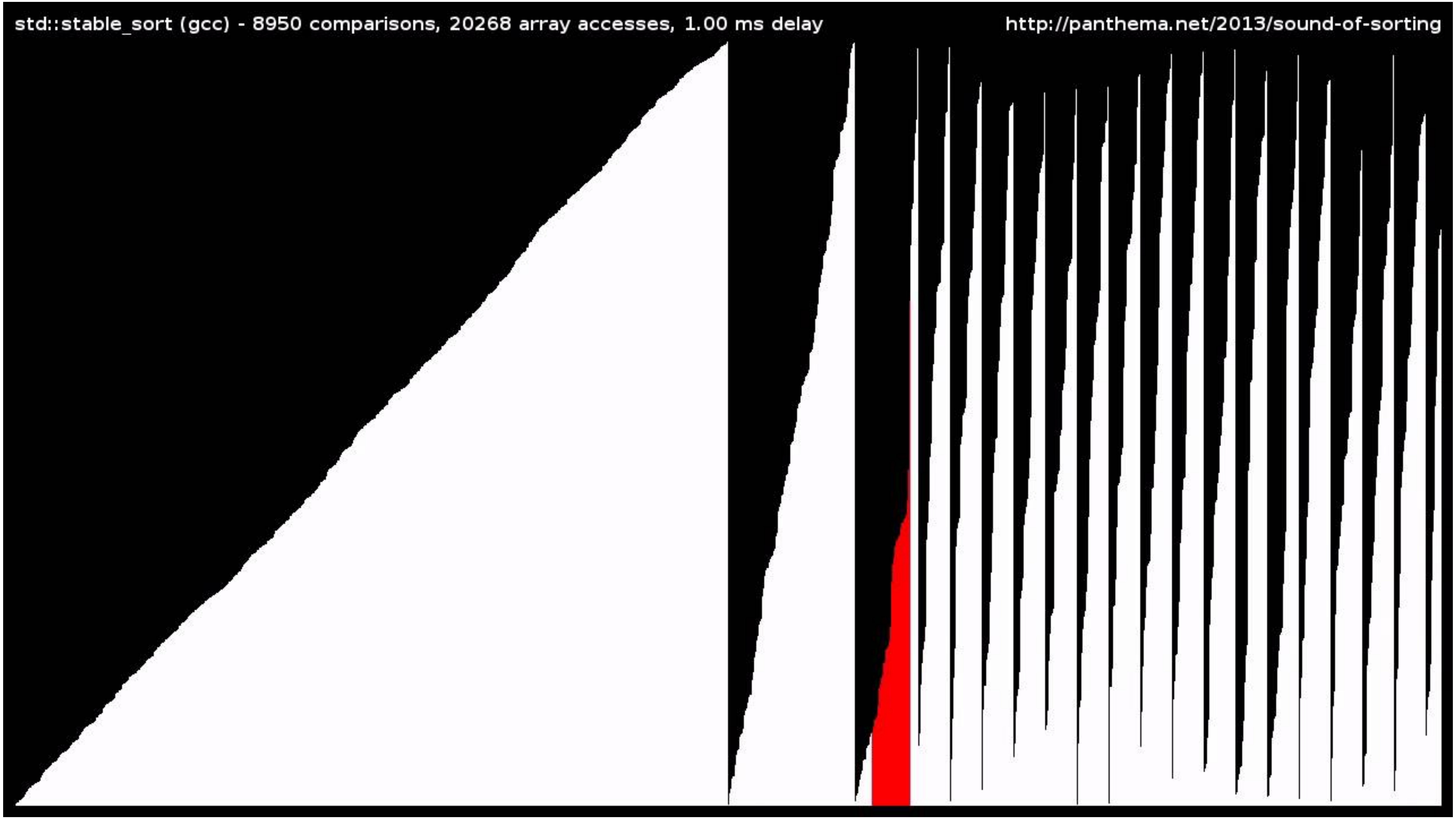
Sorting

- $O(n^2)$: 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