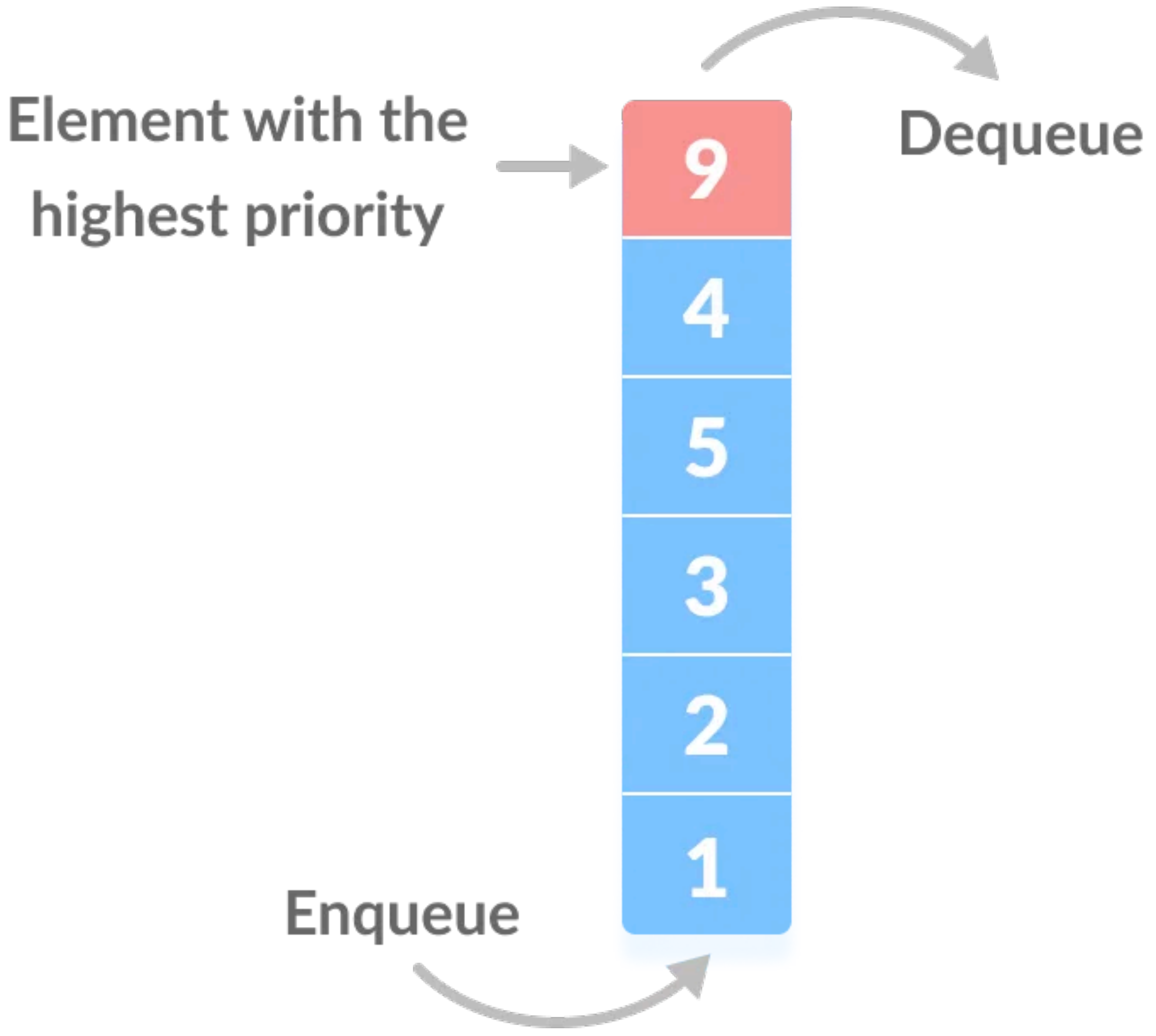
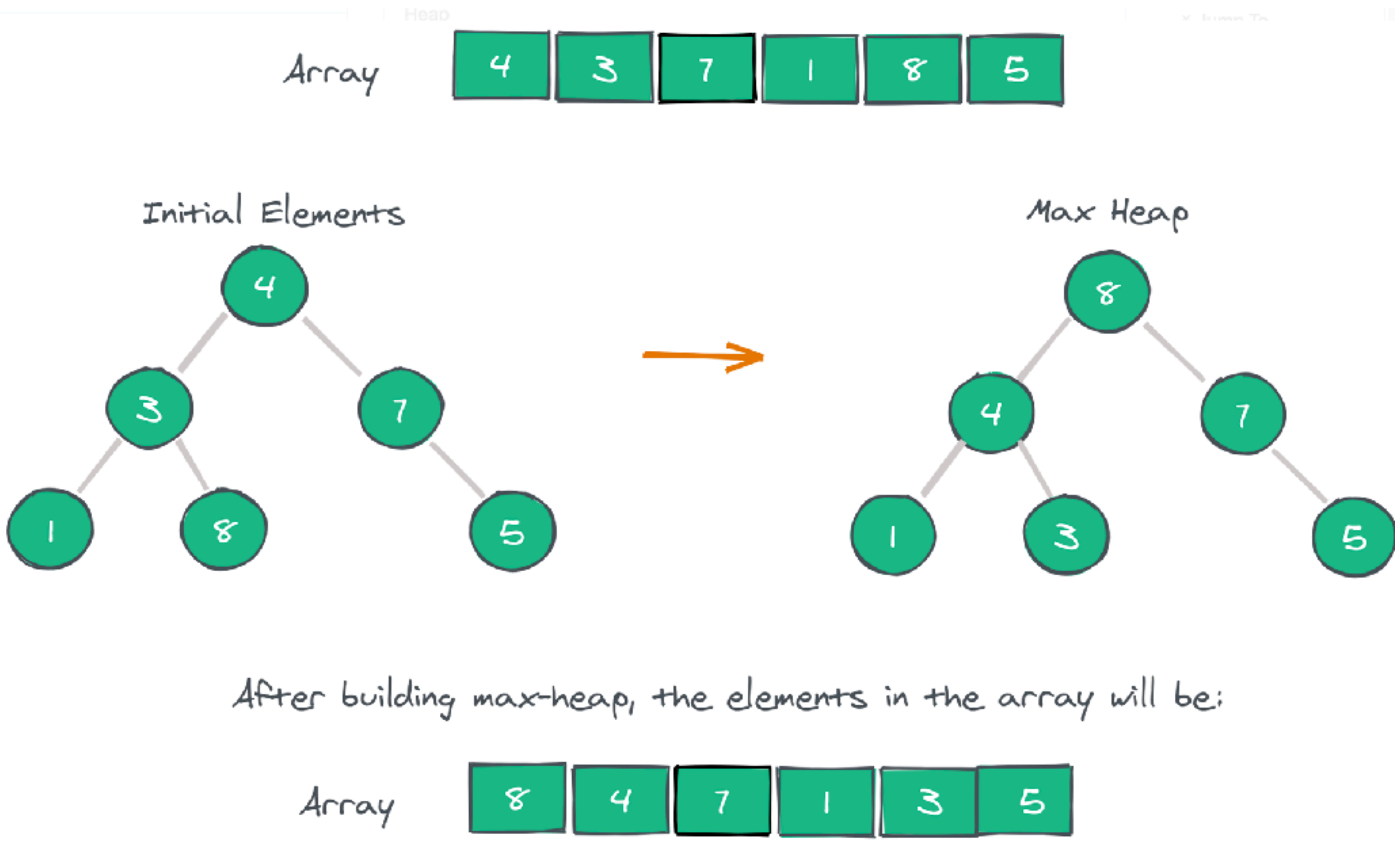
CS62 Class 16: Priority Queues & Heapsort

Sorting



Priority queue: another representation of a binary heap



Heapsort: sorting using a binary heap

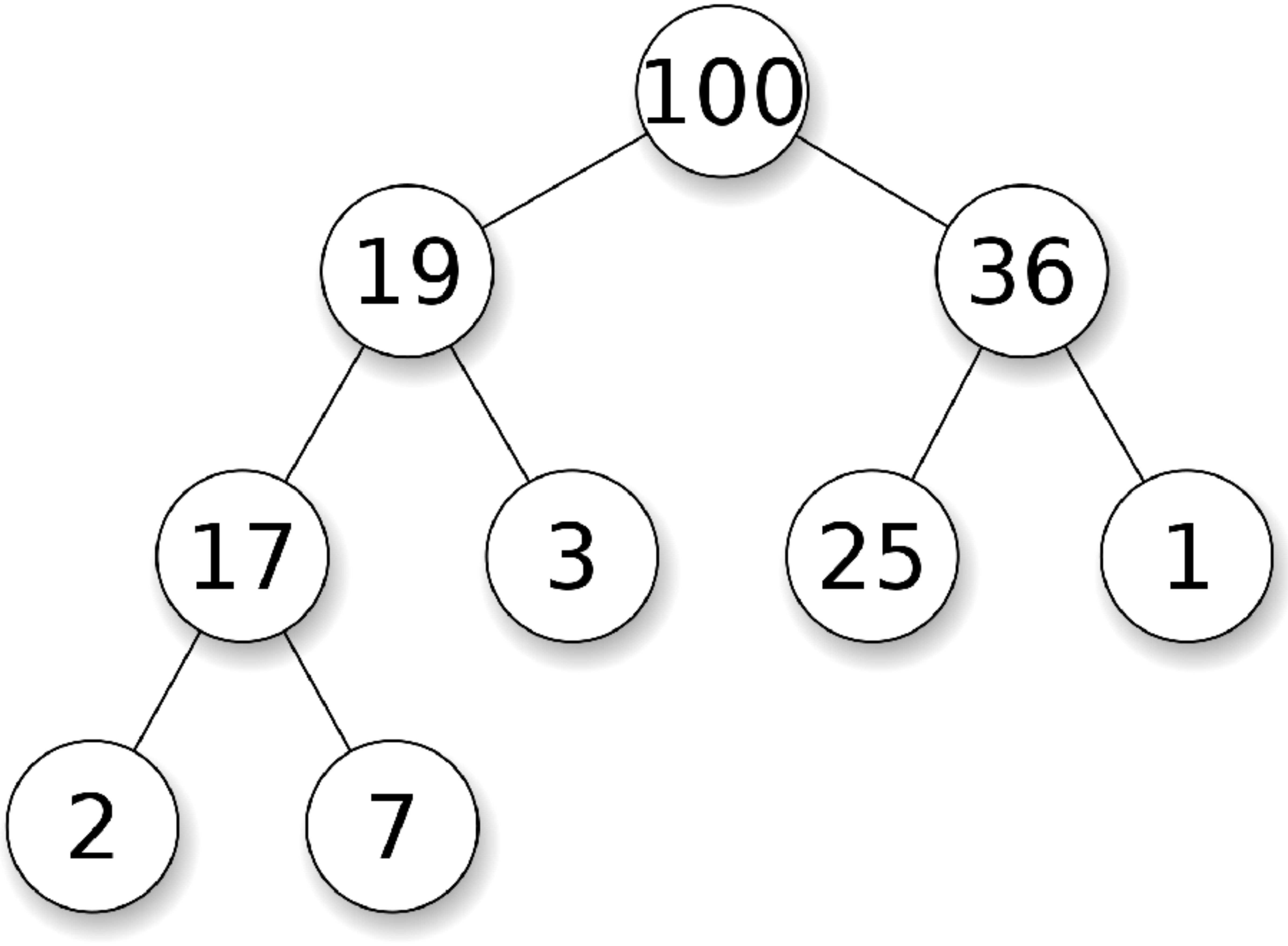
Agenda

- From last time: Binary Heaps
- Priority Queues
- Heapsort
- Heapsort Analysis

Binary Heap (pre spring break review)

Heap-ordered binary trees

- The largest key in a heap-ordered binary tree is found at the root!

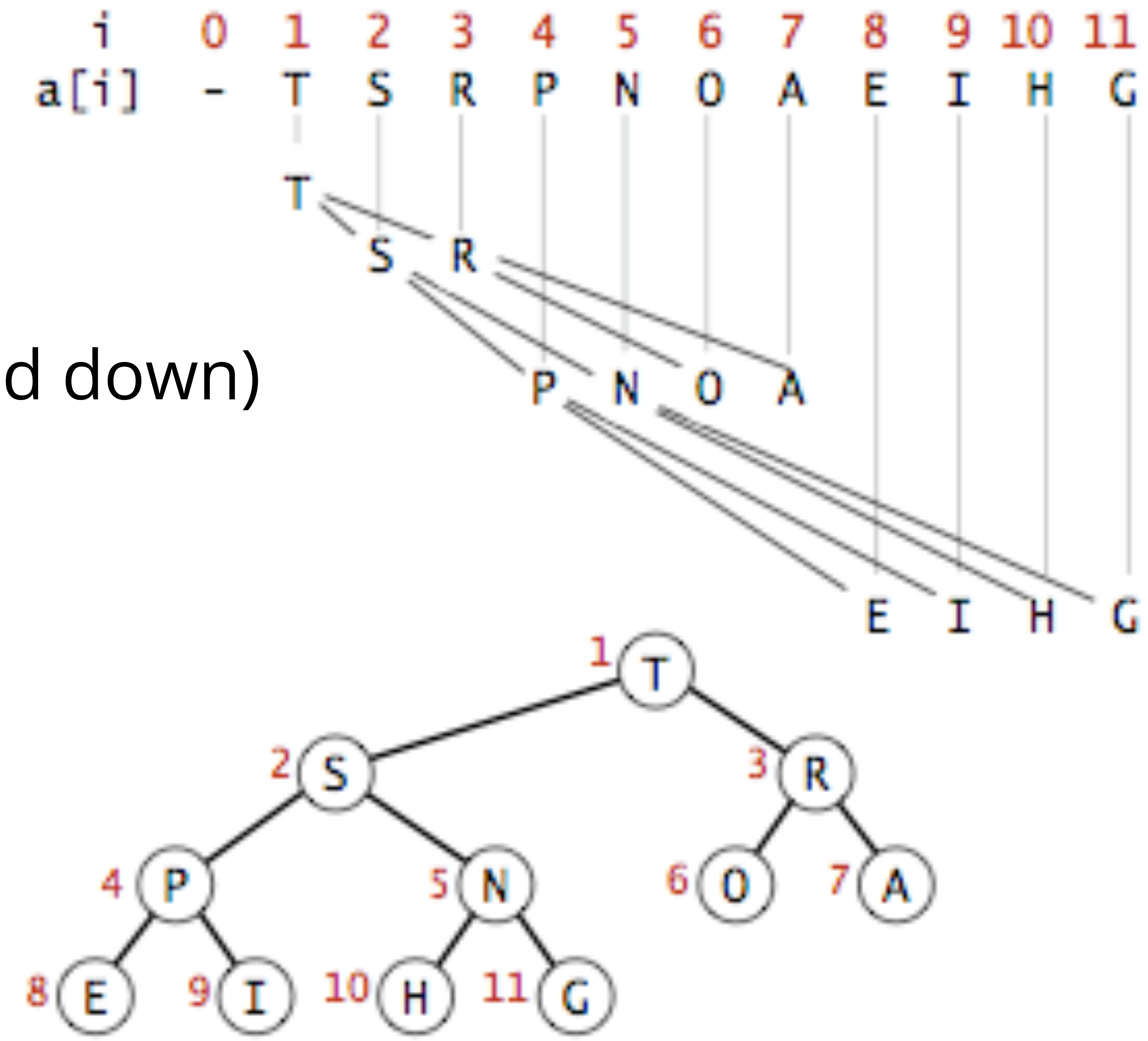


Heap-ordered binary trees

- A binary tree is heap-ordered if the key in each node is larger than or equal to the keys in that node's two children (if any).
- Equivalently, the key in each node of a heap-ordered binary tree is smaller than or equal to the key in that node's parent (if any).
- No assumption of which child is smaller.
- Moving up from any node, we get a non-decreasing sequence of keys.
- Moving down from any node we get a non-increasing sequence of keys.

Array representation of heaps

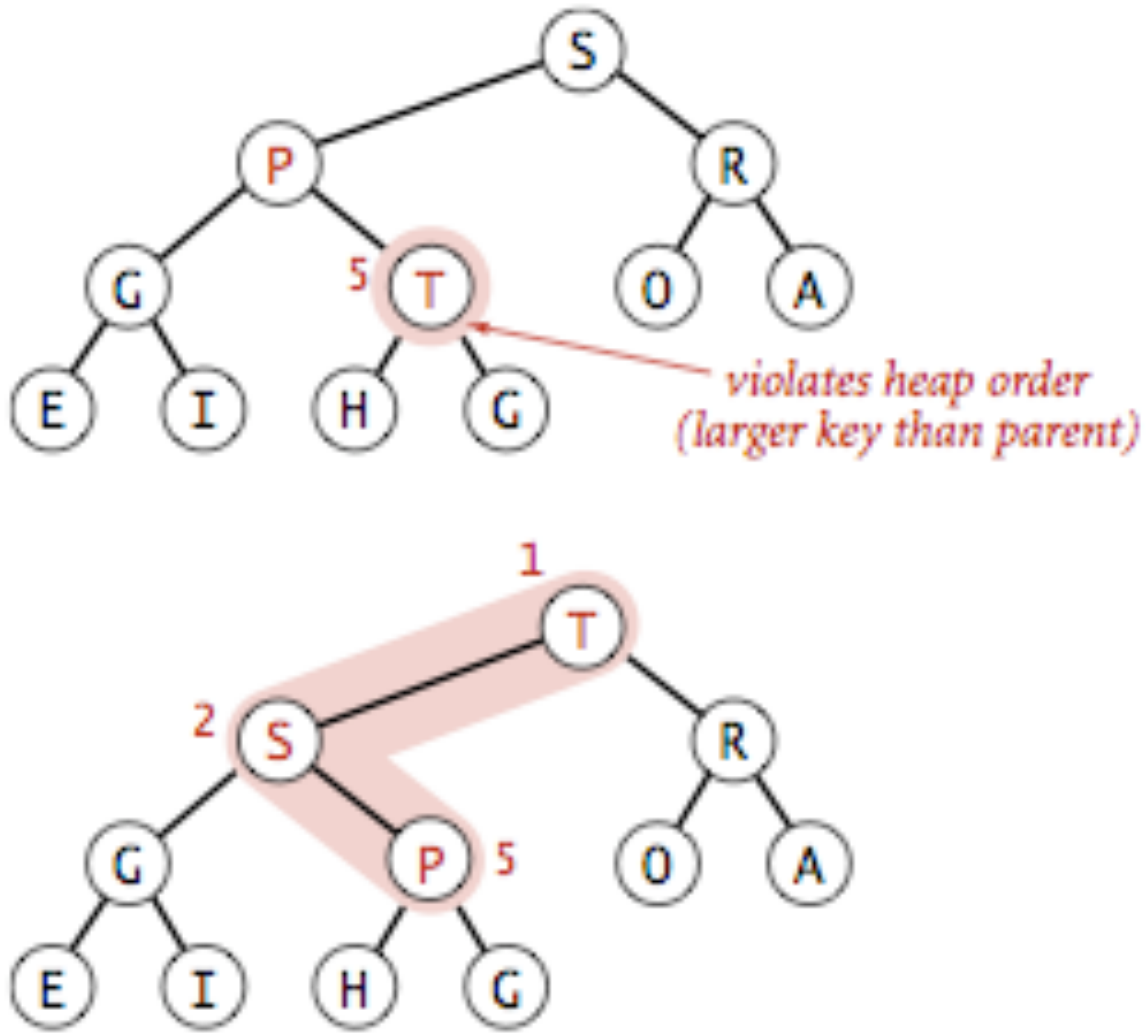
- Nothing is placed at index 0 (for arithmetic convenience).
- Root is placed at index 1.
- Rest of nodes are placed **in level** order.
- Parent of node k: found at index k/2 (round down)
- Children: 2k (left), 2k+1 (right)
- No unnecessary indices and no wasted space because it's complete.



Heap representations

Swim/promote/percolate up: code

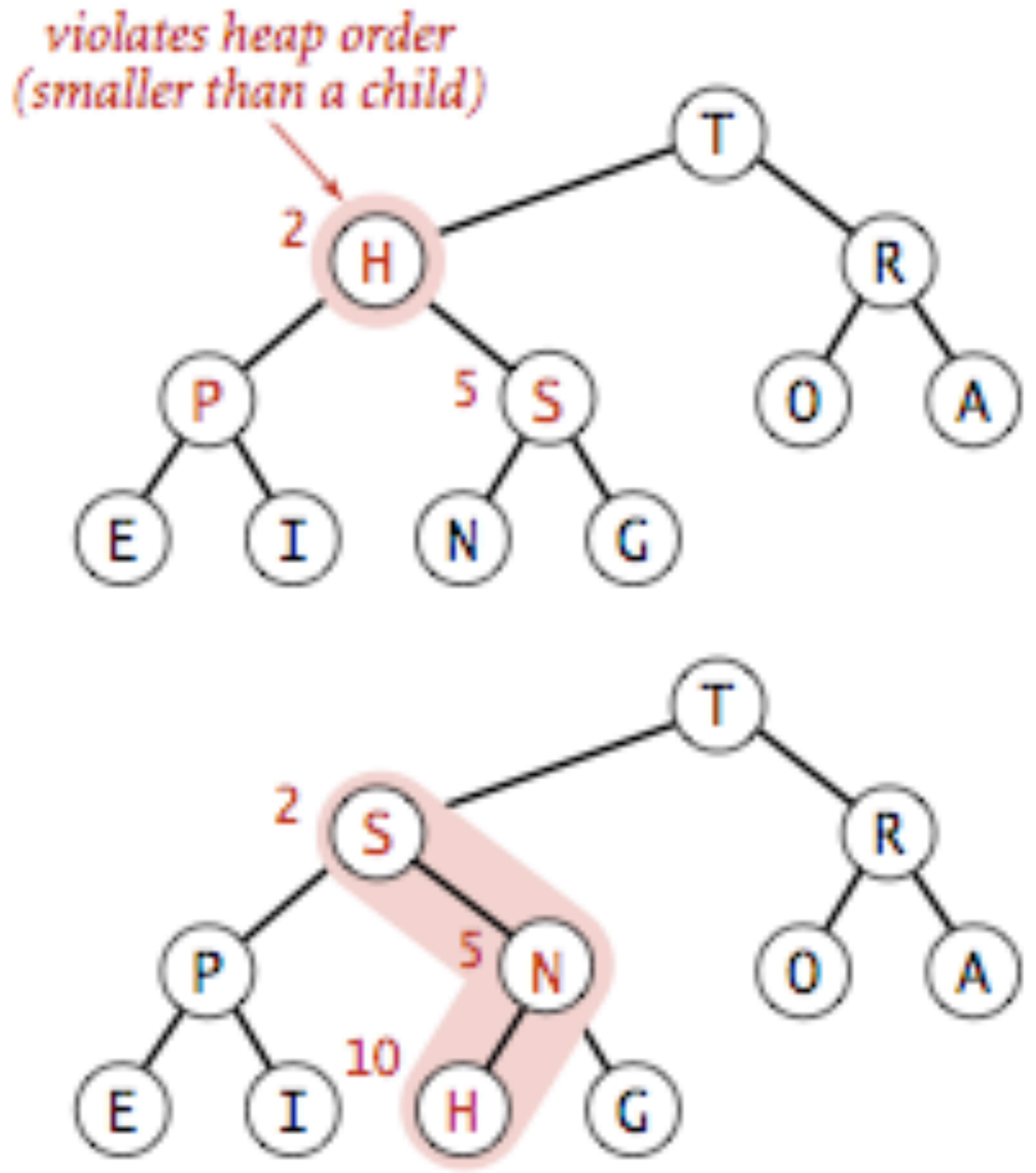
```
private void swim(int k) {
   while (k > 1 && a[k/2].compareTo(a[k])<0) {
      E temp = a[k];
      a[k] = a[k/2];
      a[k/2] = temp;
      k = k/2;
   }
}
```



We **swim large nodes** so they become parents
We do this by swapping with the parent if it’s larger

Sink/demote/top down heapify code

```
private void sink(int k) {
    while (2*k <= n) {
        int j = 2*k;
        if (j < n && a[j].compareTo(a[j+1])<0))
            j++;
        if (a[k].compareTo(a[j])>=0))
            break;
        E temp = a[k];
        a[k] = a[j];
        a[j] = temp;
        k = j;
    }
}
```

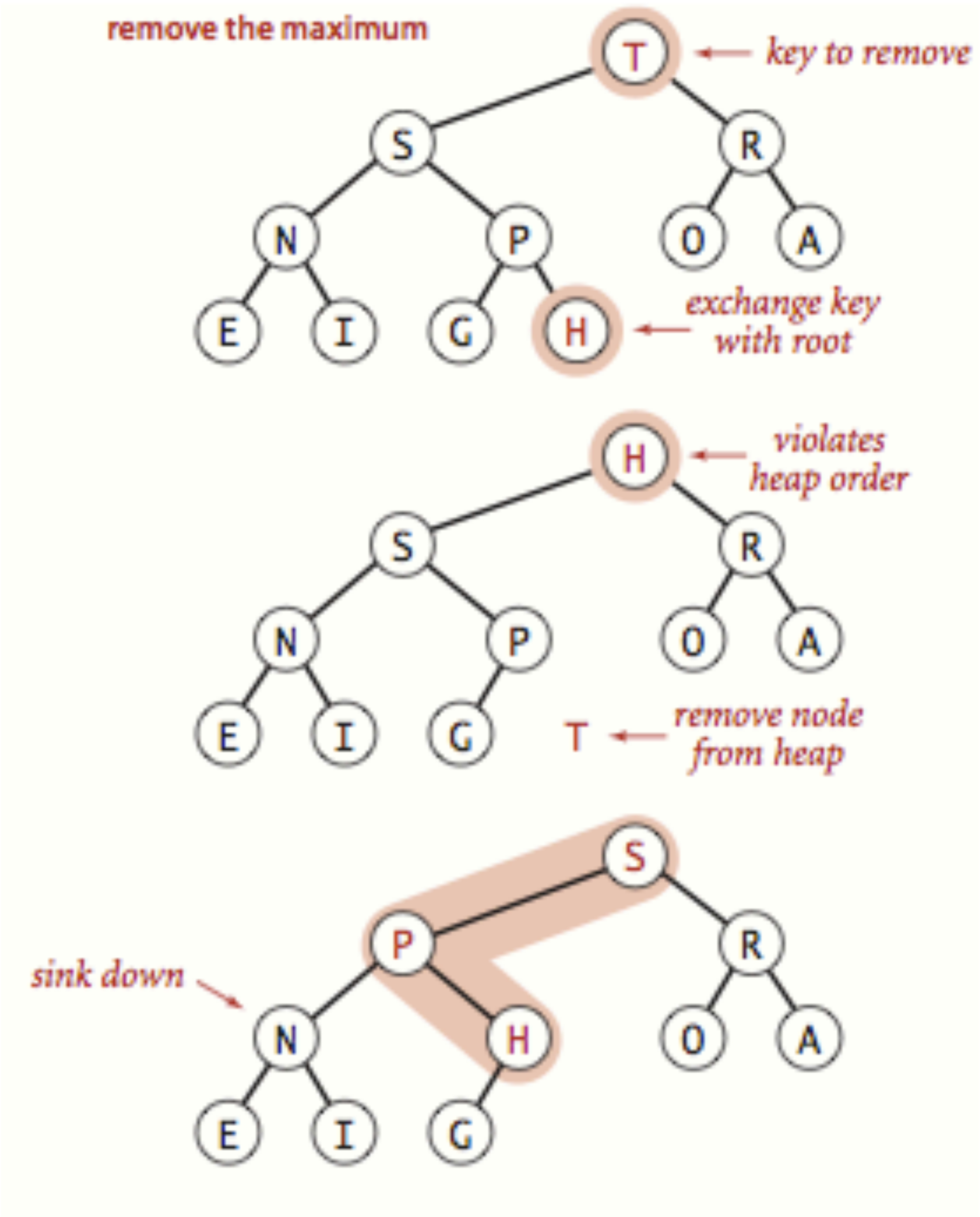


We **sink small nodes** so they become leaves
We do this by swapping with the larger child

Binary Heap (new)

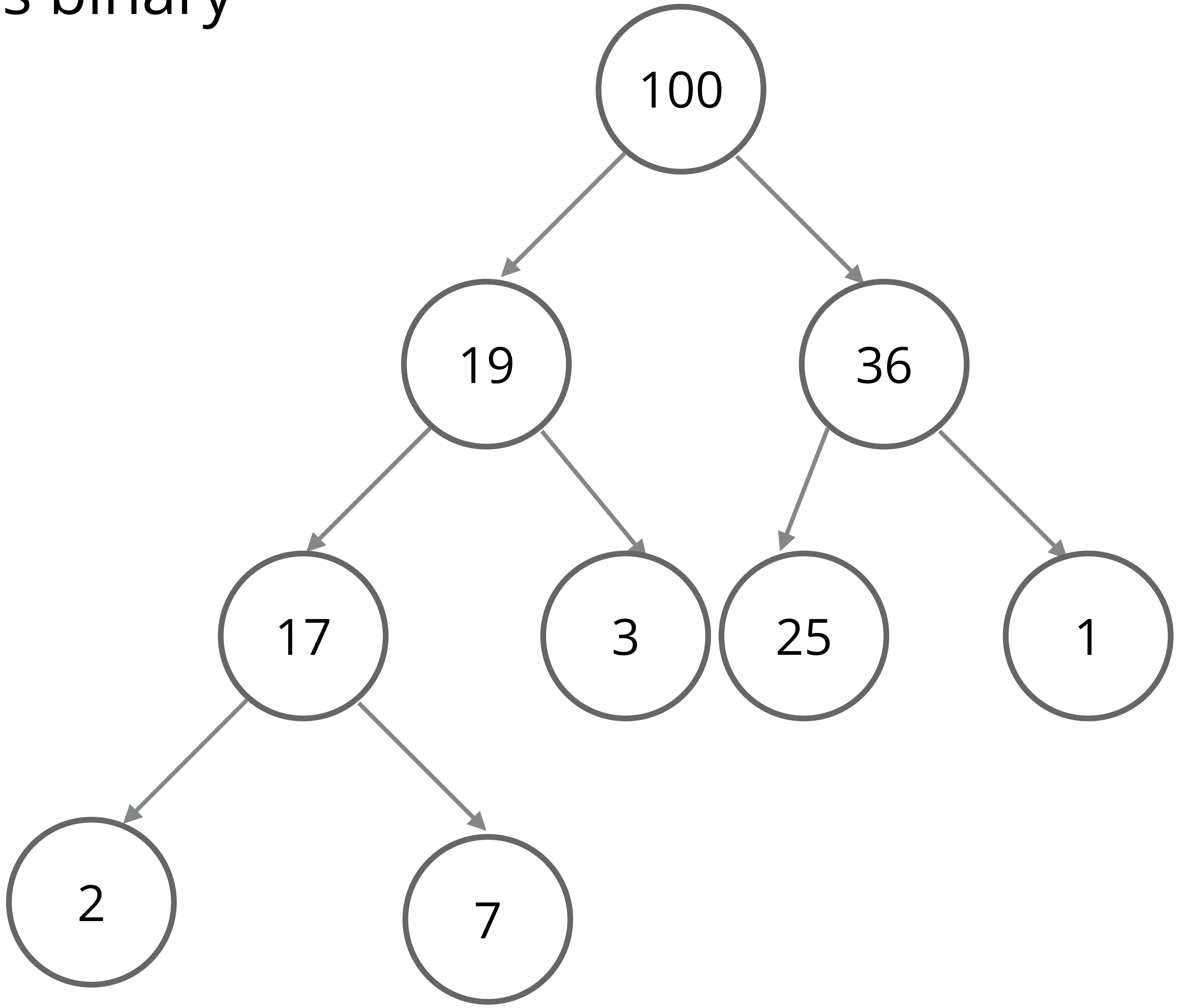
Binary heap: return (and delete) the maximum

- Delete max: Swap the root with the last node (the rightmost child). Return and delete the root. Sink the new root down.
- Cost: At most $2 \log n$ compares.



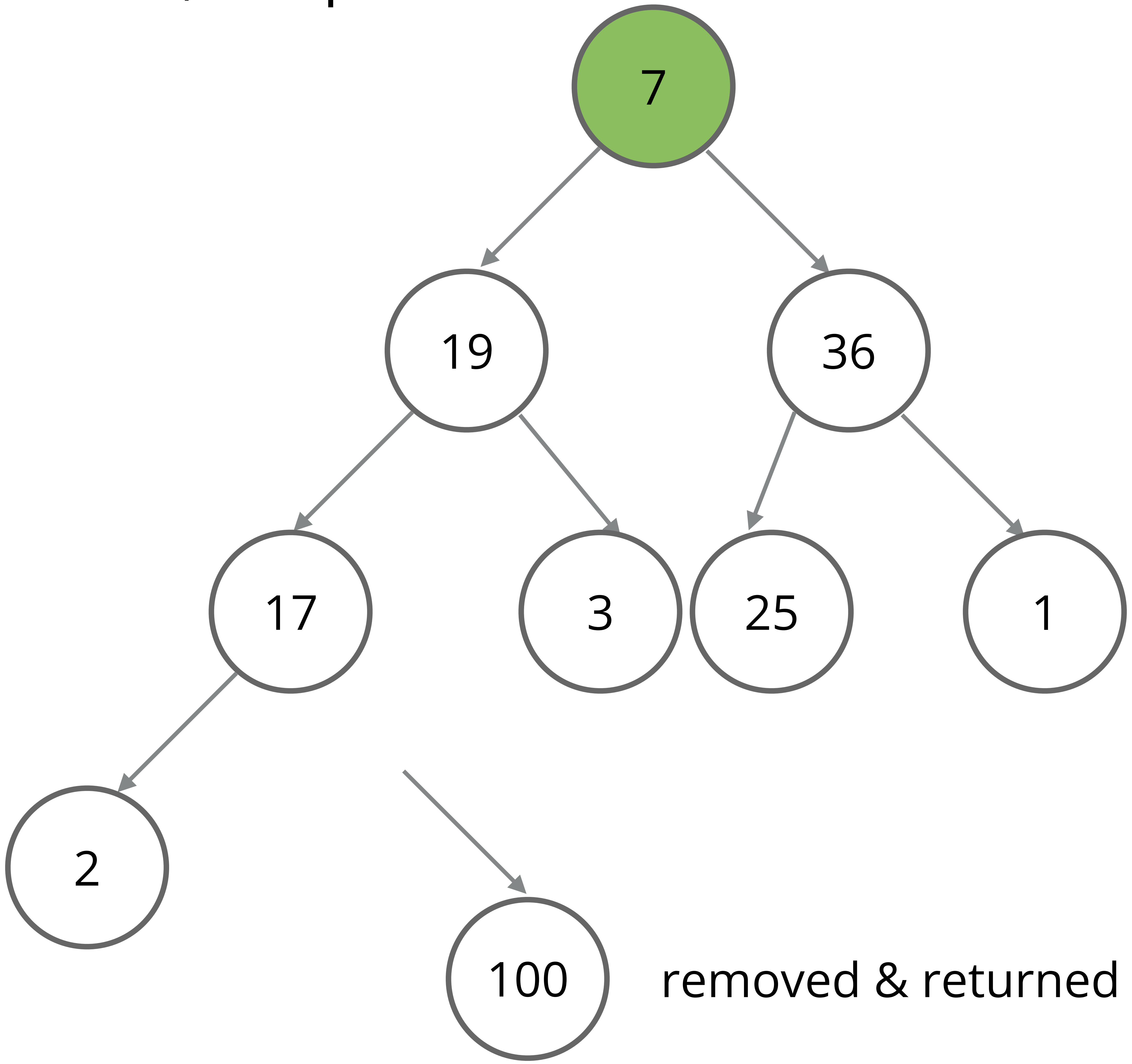
Worksheet time!

- Delete and return the maximum of this binary heap.

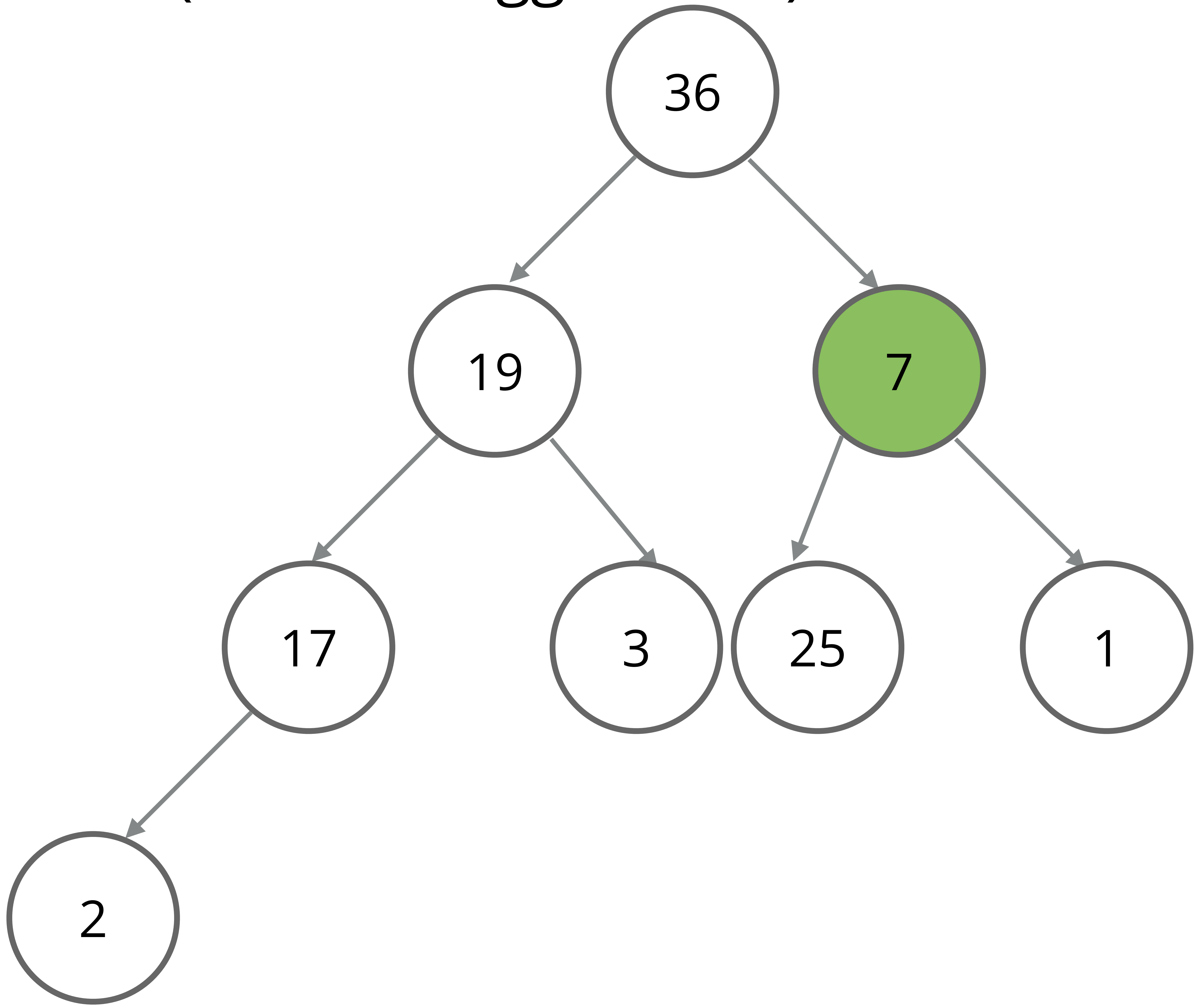


Worksheet answers

- First, swap with 7

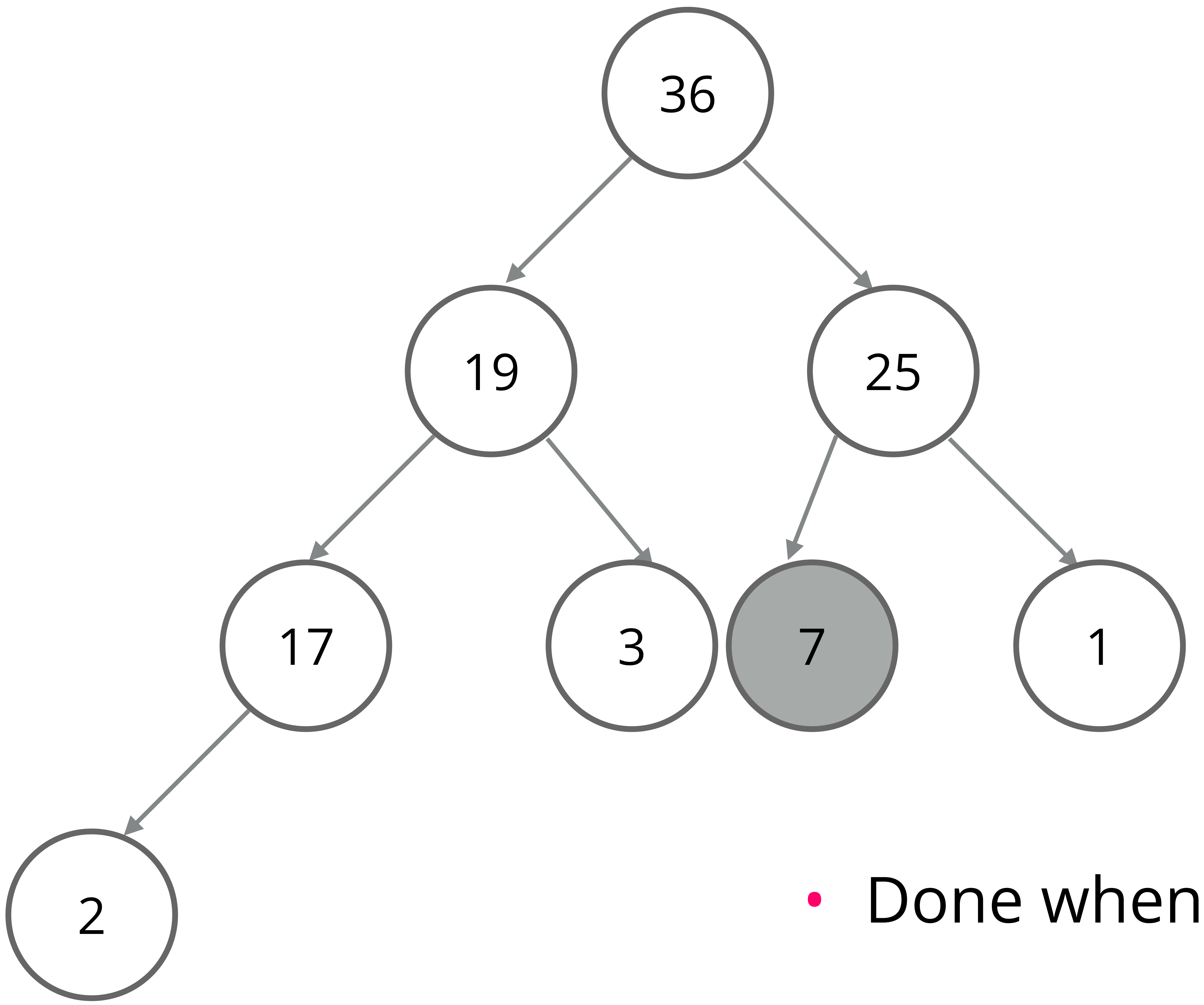


- Then, sink 7 (find the bigger child)



Worksheet answers

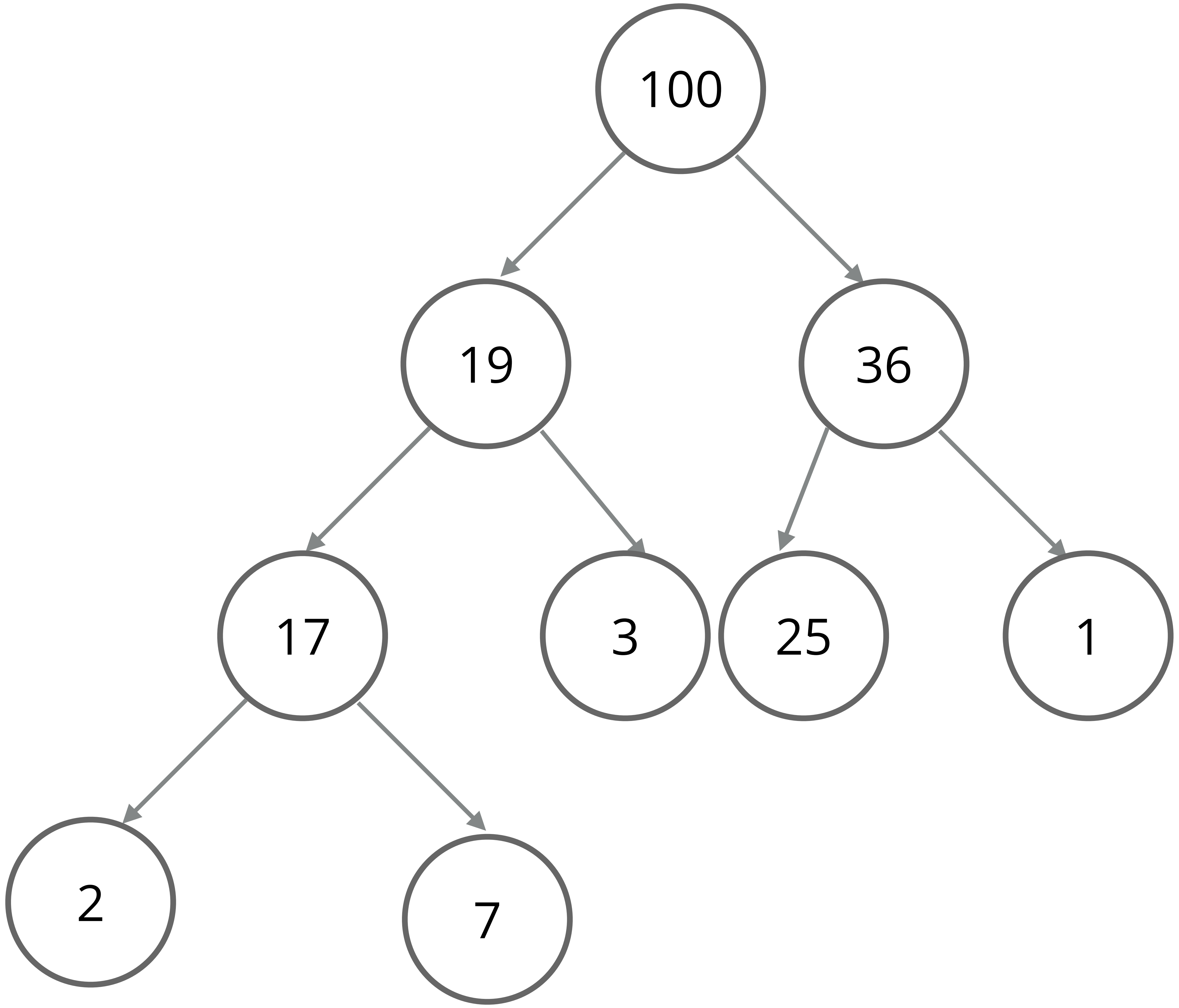
- Then, sink 7 (find the bigger child)



- Done when 7 has no more bigger children

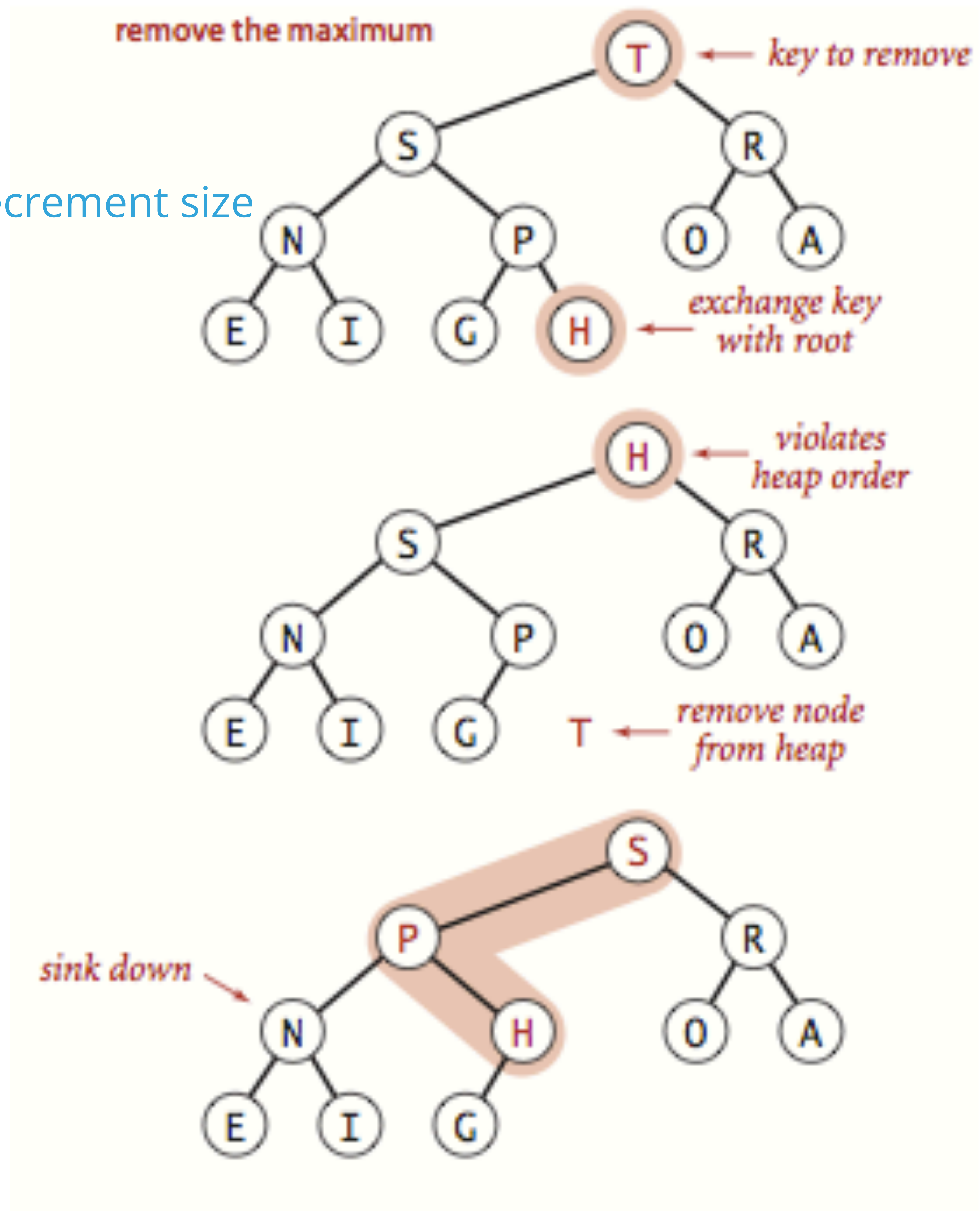
Worksheet time!

- Implement public E deleteMax().
- Assume precondition (n > 0) is true.
- Hint: you can do it in 4 lines of code.
- 1. find max
- 2. ??
- 3. ??
- 4. return max



Worksheet answers

```
public E deleteMax() {
    E max = a[1];       max is always the root
    a[1] = a[n--];      swap root with the last element, decrement size
    sink(1);            sink the last element to update tree
    return max;
}
```



Binary heap operation run times

- Insertion is $O(\log n)$ (because insert at the end, swim up to proper place).
- Delete max is $O(\log n)$ (because swap last node to root, and then sink down to proper place).
- Space efficiency is $O(n)$ (because of array representation).

http://algs4.cs.princeton.edu

2.4 BINARY HEAP DEMO

Priority Queues

Priority Queue

- An abstract data type of a queue where each element additionally has a *priority.*
- Two operations:
  - Dequeue, aka delete the maximum
  - Enqueue, aka insert
- How can we implement a priority queue efficiently?

Option 1: Unordered array

- The *lazy* approach where we defer doing work (deleting the maximum) until necessary.
- Insert is $O(1)$ and assumes we have the space in the array.
- Delete maximum is $O(n)$ (have to traverse the entire array to find the maximum element and exchange it with the last element).

Option 2: Ordered array

- The *eager* approach where we do the work (keeping the array sorted) up front to make later operations efficient.
- Insert is $O(n)$ (we have to find the index to insert and shift elements to perform insertion).
- Delete maximum is $O(1)$ (just take the last element which will be the maximum).

Option 3: Binary heap

- Will allow us to both insert and delete max in $O(\log n)$ running time.
- There is no way to implement a priority queue in such a way that insert *and* delete max can be achieved in $O(1)$ running time.
- Priority queues are synonymous to binary heaps.

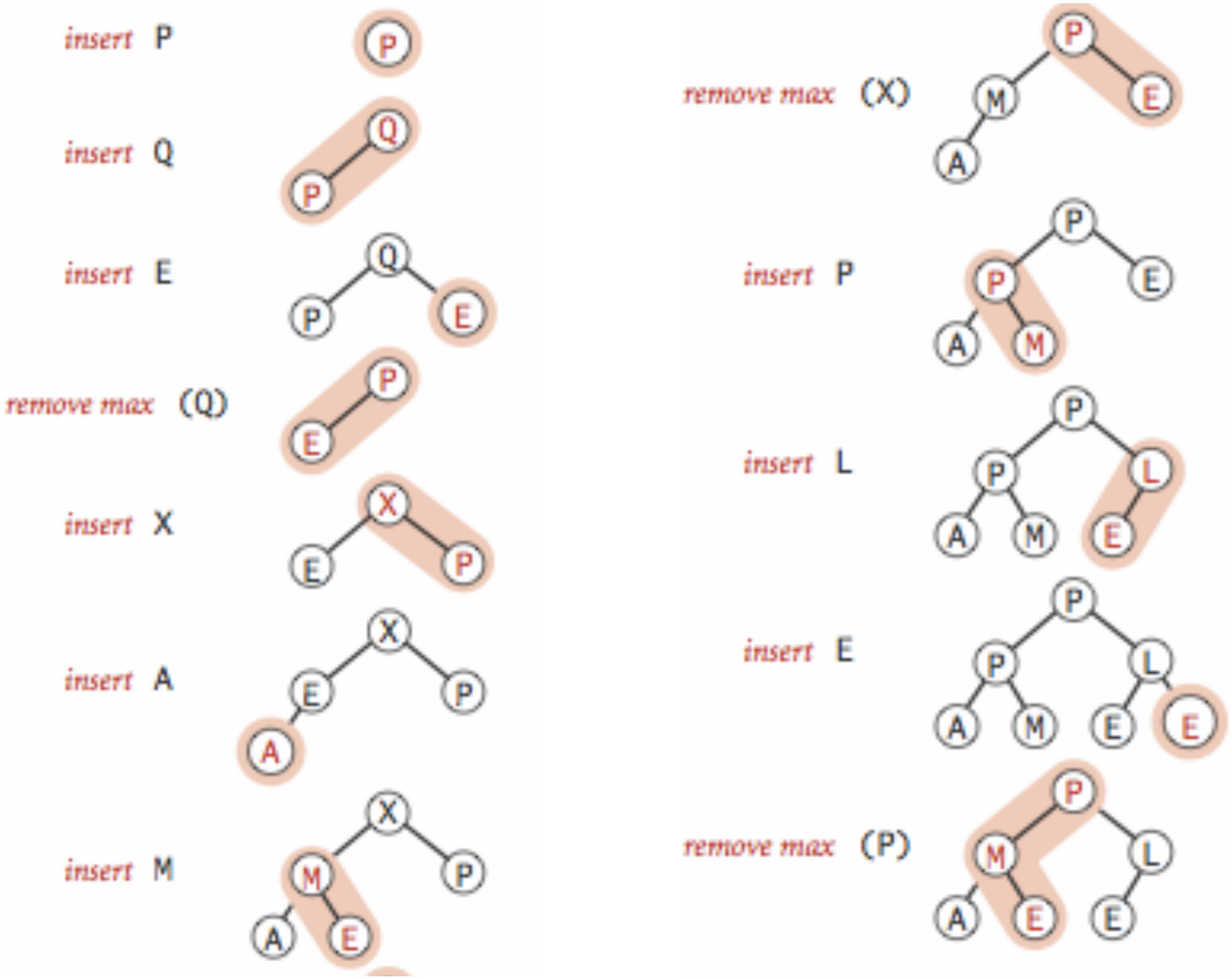
Worksheet time!

Given an empty binary heap that represents a priority queue, perform the following operations. Ideally draw the binary tree at each step, but compare with your neighbors what it looks like in the end, and what the 3 delete maxes return.

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Worksheet answers

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max



- Look into MaxPQ class https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/MaxPQ.java.html

Heapsort

Basic plan for heap sort

- Given an array to be sorted, use a priority queue to develop a sorting method that works in two steps:
- 1) Heap construction: build a binary heap with all n keys that need to be sorted.
- 2) Sortdown: repeatedly remove and return the maximum key.
- Basically, we sort an array by constructing a binary heap and continually removing the max (root).

$O(n \log n)$ Naïve heap construction

- Insert n elements, one by one, swim up to their appropriate position.
  - Remember that insert() in a binary heap takes O(log n) time because swim takes O(log n) time)
- We can do better!

```java
private void swim(int k) {
    while (k > 1 && a[k / 2].compareTo(a[k]) < 0) {
        E temp = a[k];
        a[k] = a[k / 2];
        a[k / 2] = temp;
        k = k / 2;
    }
}

public void insert(E x) {
    a[++n] = x;
    swim(n);
}
```

$O(n)$ Heap construction

- Recall `sink(k)`: small nodes who are parents are sunken down to their proper place (switched with their larger child)
- Key insight: After `sink(k)` completes, the subtree rooted at k is a heap. Basically, performing sink guarantees the subtree at node *k* is a valid binary heap because of the switches.

```
private void sink(int k) {
    while (2 * k <= n) {
        int j = 2 * k;
        if (j < n && a[j].compareTo(a[j + 1]) < 0) j++;
        if (a[k].compareTo(a[j]) >= 0) break;
        E temp = a[k];
        a[k] = a[j];
        a[j] = temp;
        k = j;
    }
}
```

$O(n)$ Heap construction algorithm

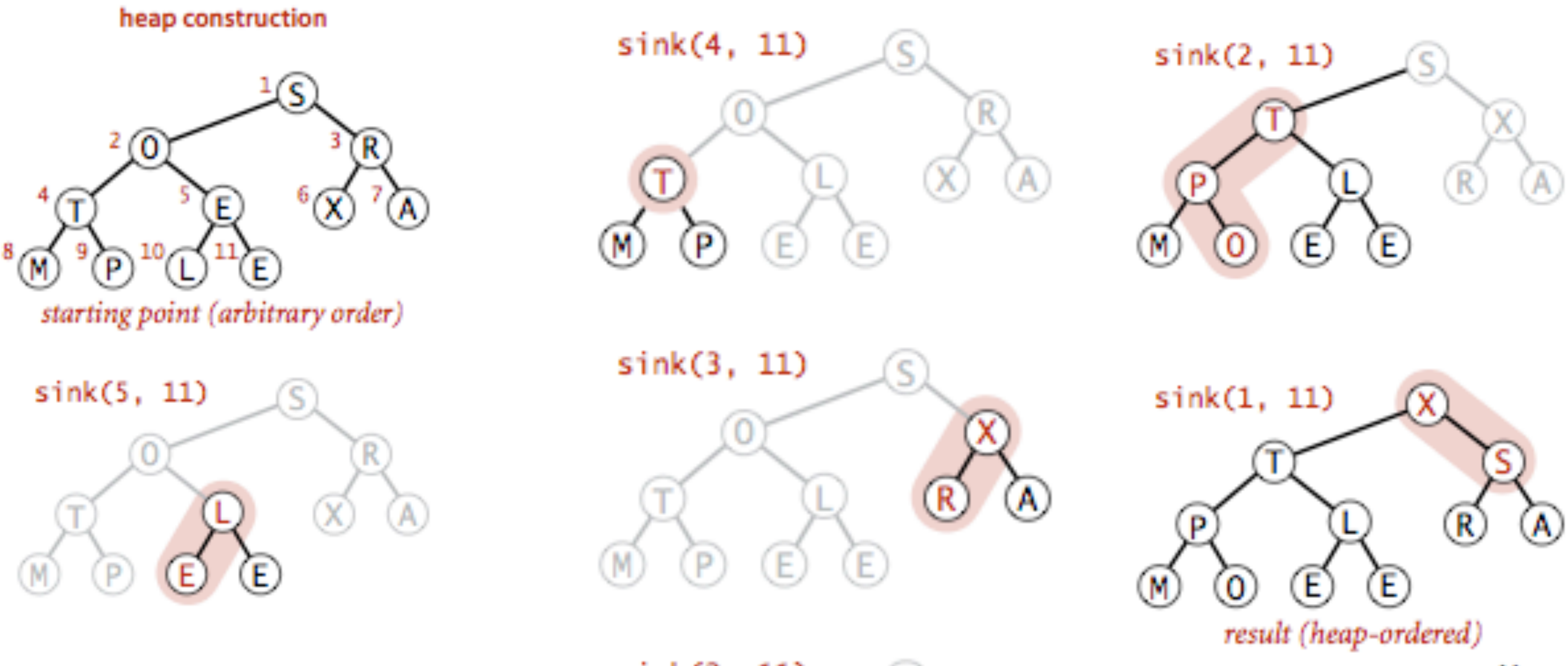
- 1. Insert all nodes as is, in indices 1 to n (e.g., starting point is the first element is the root, the second element is the left child, the third is the right child, etc.). This is a binary tree definitely not in heap order.
- 2. Sink each internal node, ignoring all the leaves (indices n/2+1,…,n). Remember the leaves will be placed in correct order since they are subtrees of the internal nodes.

```
public class HeapSort {
    public static <E extends Comparable<E>> void sort(E[] input) {
        int n = input.length;

        // create a 1-indexed array to make the math cleaner for this demo
        // (though you shouldn't do this in practice)
        E[] a = (E[]) new Comparable[n + 1];
        System.arraycopy(input, 0, a, 1, n);

        // Heap construction in O(n)
        for (int k = n / 2; k >= 1; k--) {
            sink(a, k, n);
        }
```

Example: SORTEDEXAMPLE

```
for (int k = n / 2; k >= 1; k--) {
```

n=11, so k=5 initially

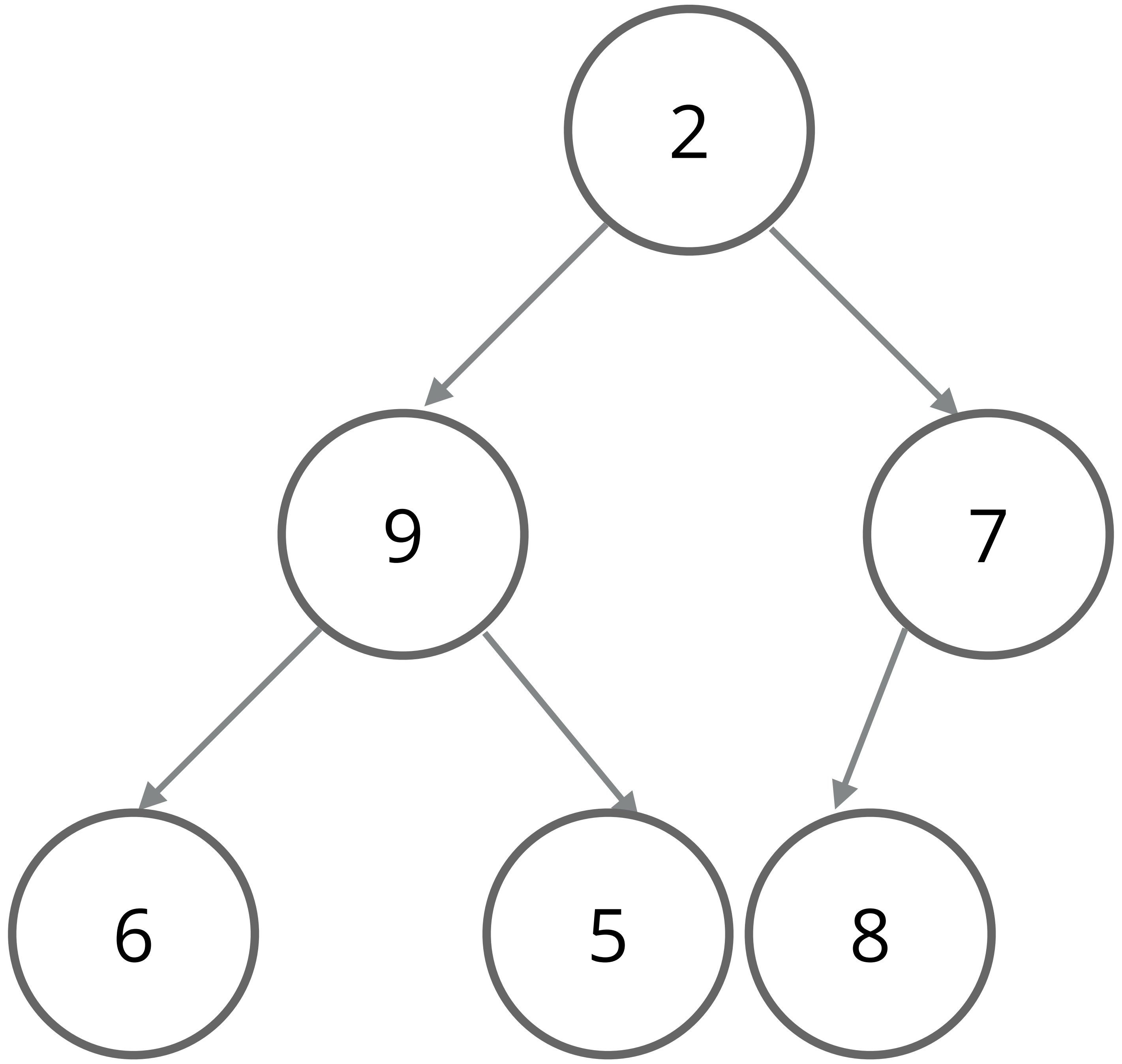


Worksheet time!

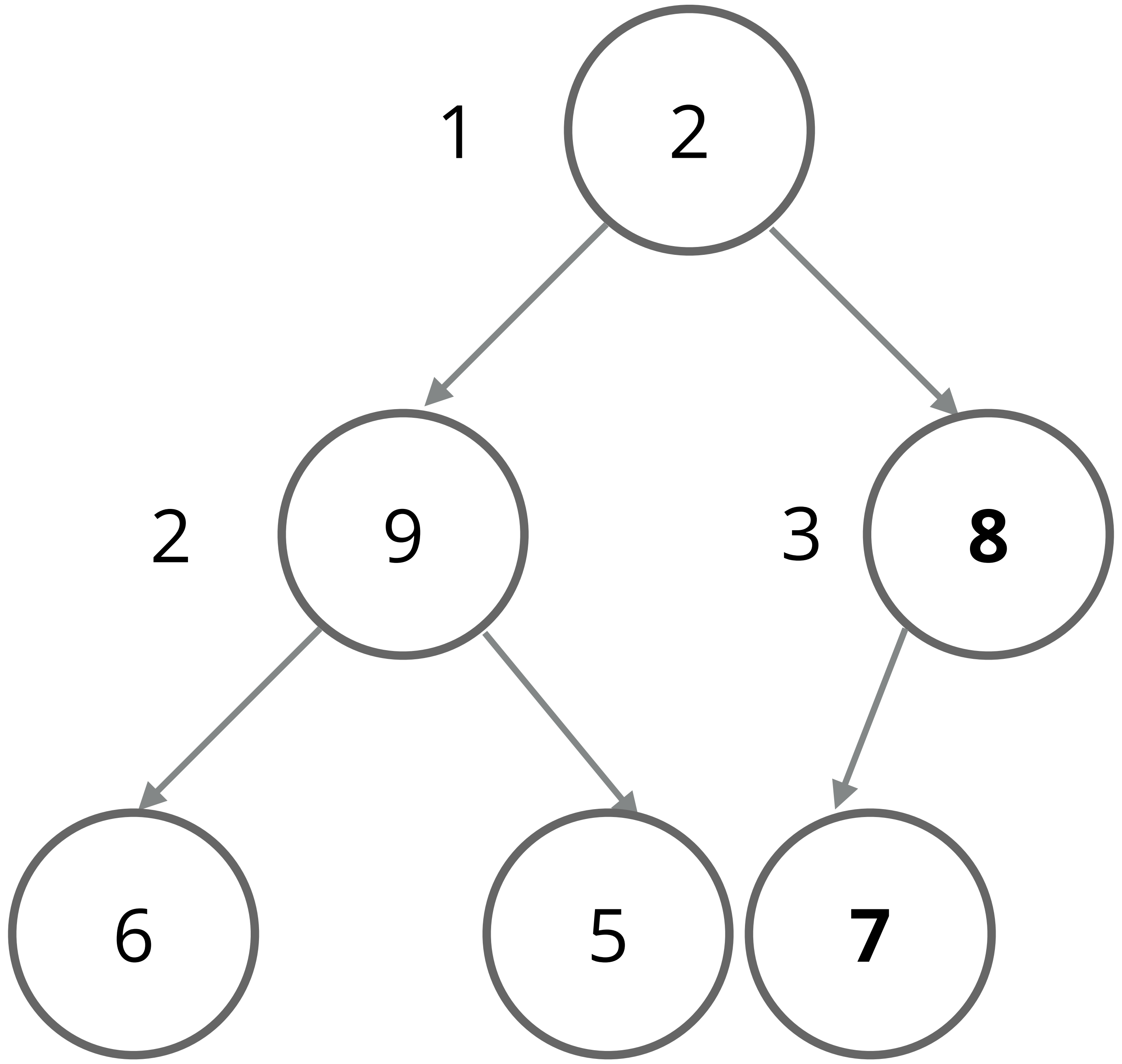
- Run the first step of heapsort, heap construction, on the array [2,9,7,6,5,8]. What is the resultant binary heap?

Worksheet answer

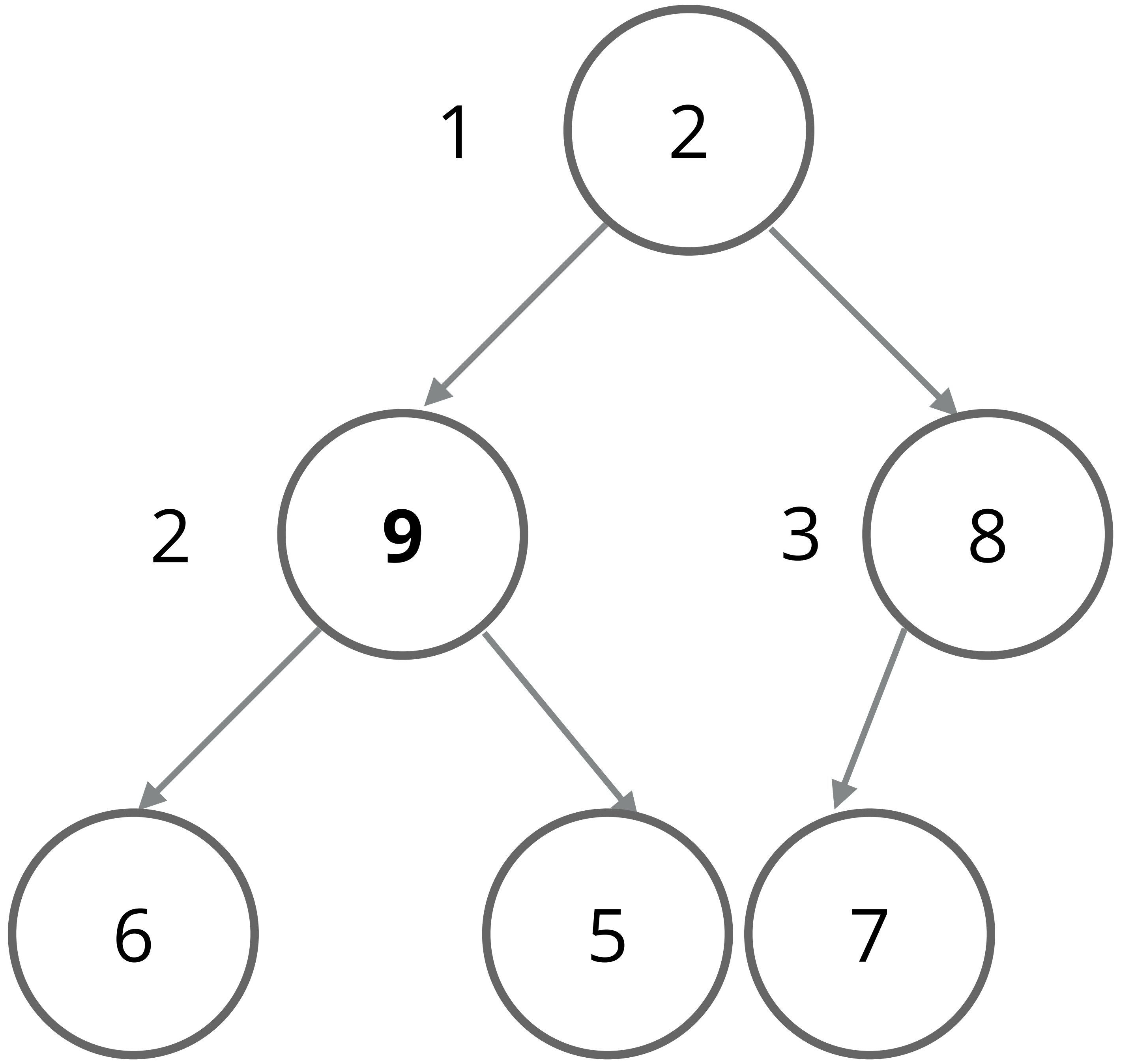
Step one: just in array order



2. sink(3, 6)



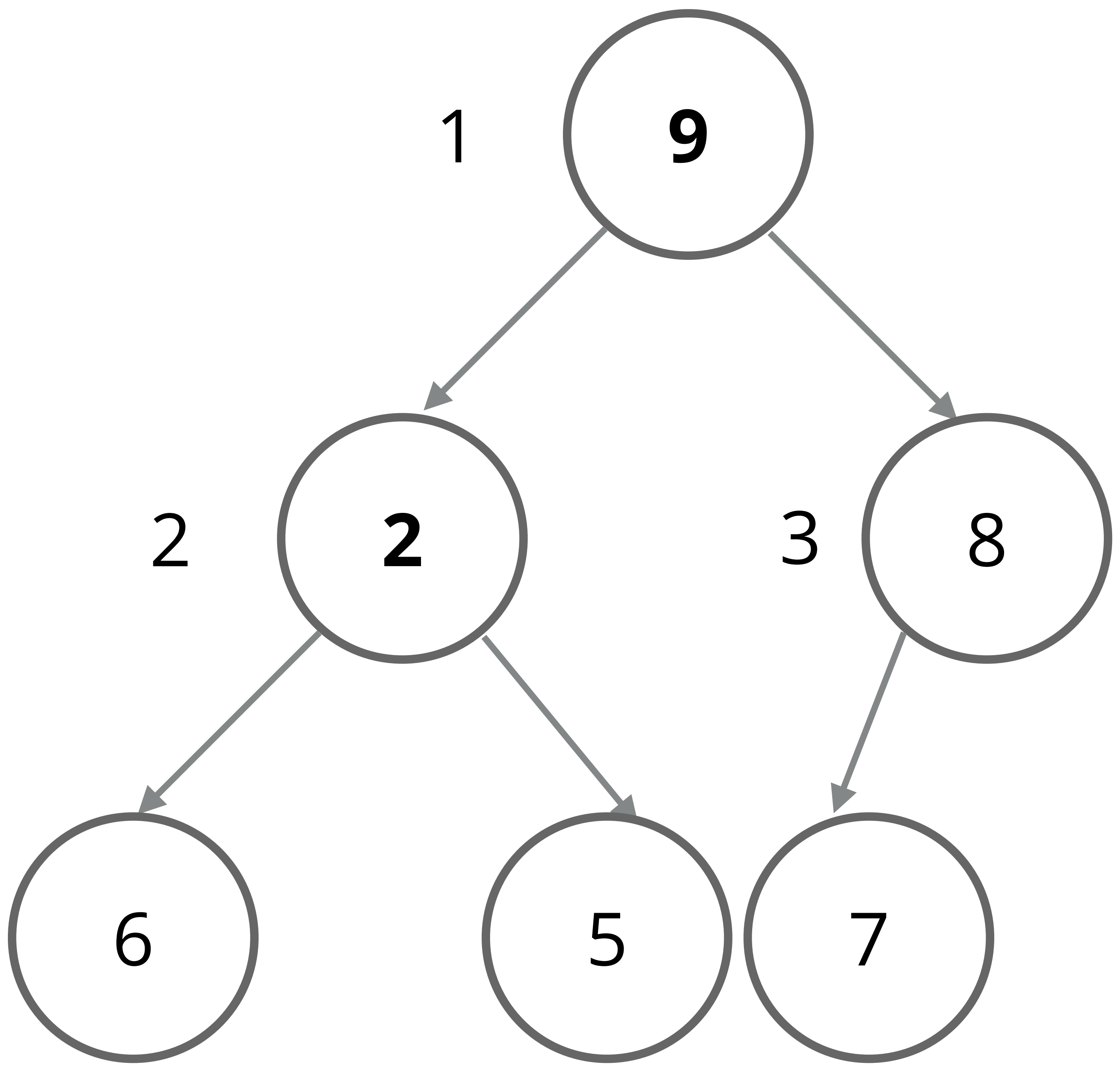
3. sink(2, 6)



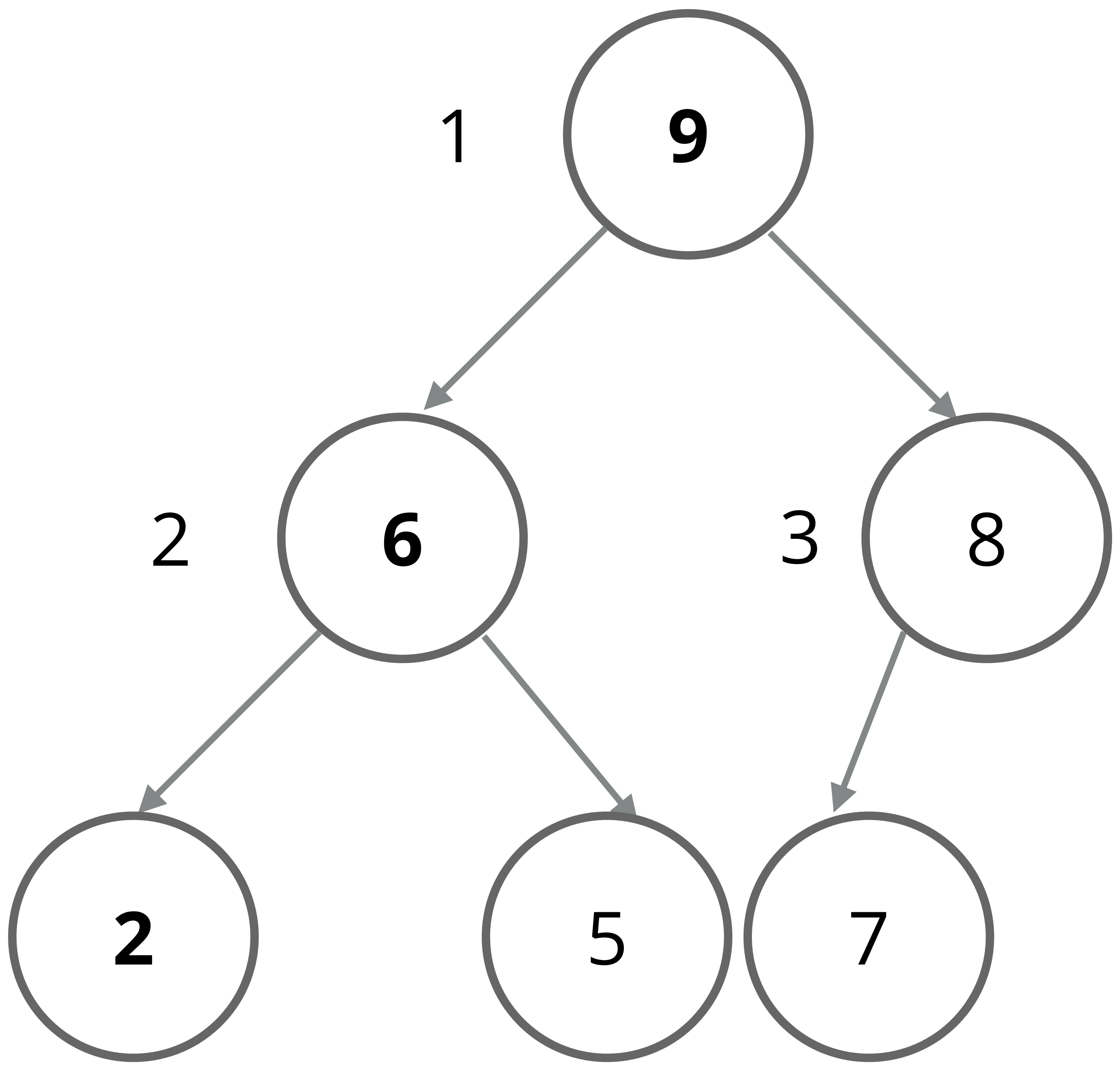
(no action needed)

Worksheet answer

4. sink(1,6)



part 1: swap 2 & 9 (9 > 8)



Final heap!

part 2: swap 6 & 2

Sortdown

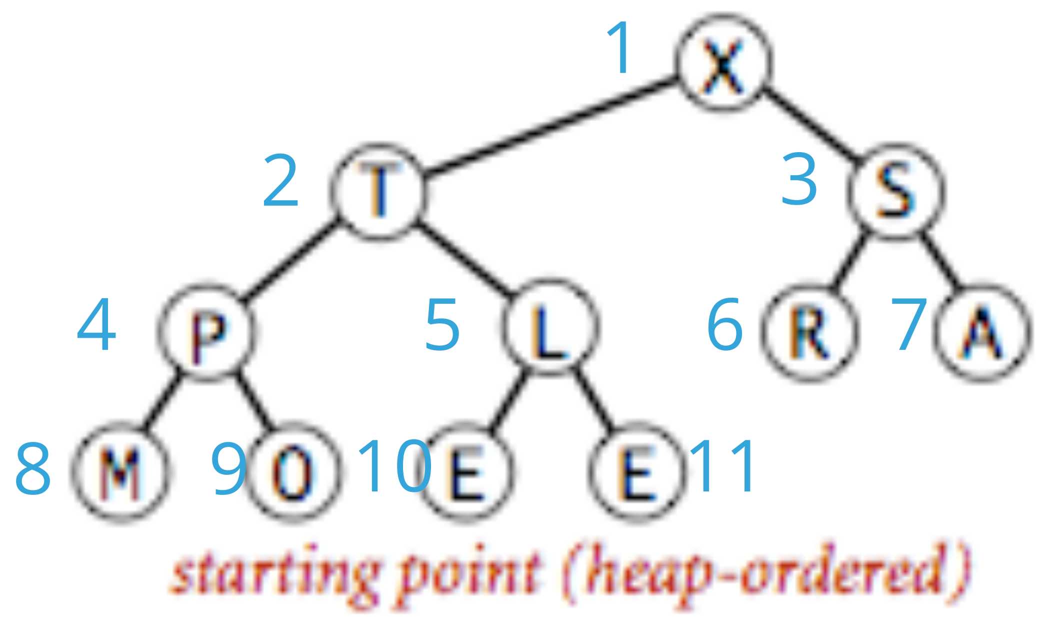
- Now that we have an ordered binary heap, all that remains is to pull out the roots (each subsequent max element).
- Recall: deleteMax() in binary heaps swaps the last element to be the new root and sinks that down.
- Key insight: After each iteration of sortDown, the array consists of a heap-ordered subarray of k elements, followed by a sub-array of n-k elements in final order.

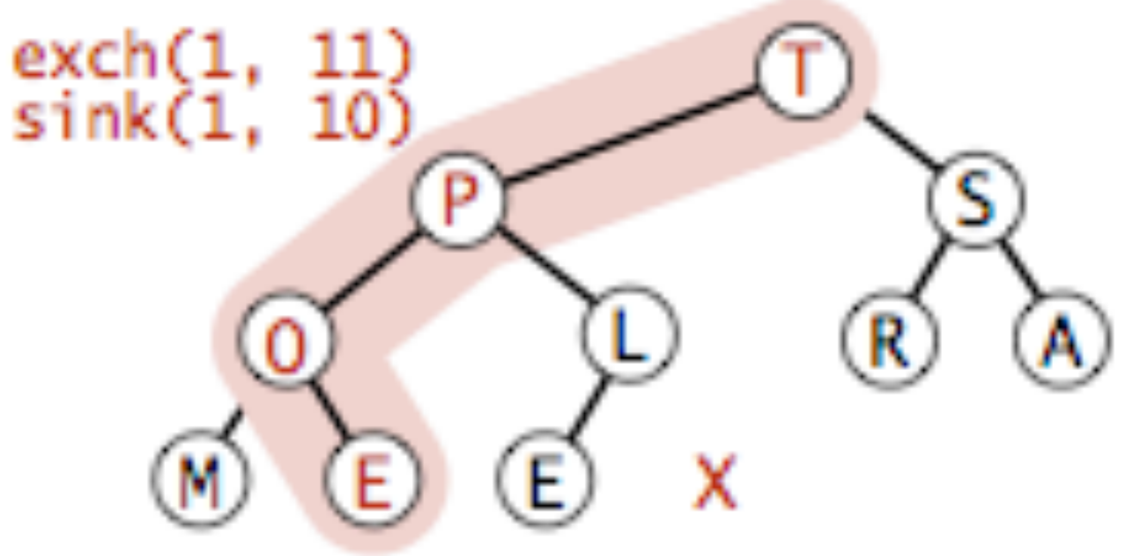
```
// Sorting in O(nlogn)
while (n > 1) {
    swap(a, 1, n--);
    sink(a, 1, n);
}
```

While the heap has > 1 element,
swap the root with the last element
sink the new root appropriately

```
while (n > 1) {
    swap(a, 1, n--);
    sink(a, 1, n);
}
```

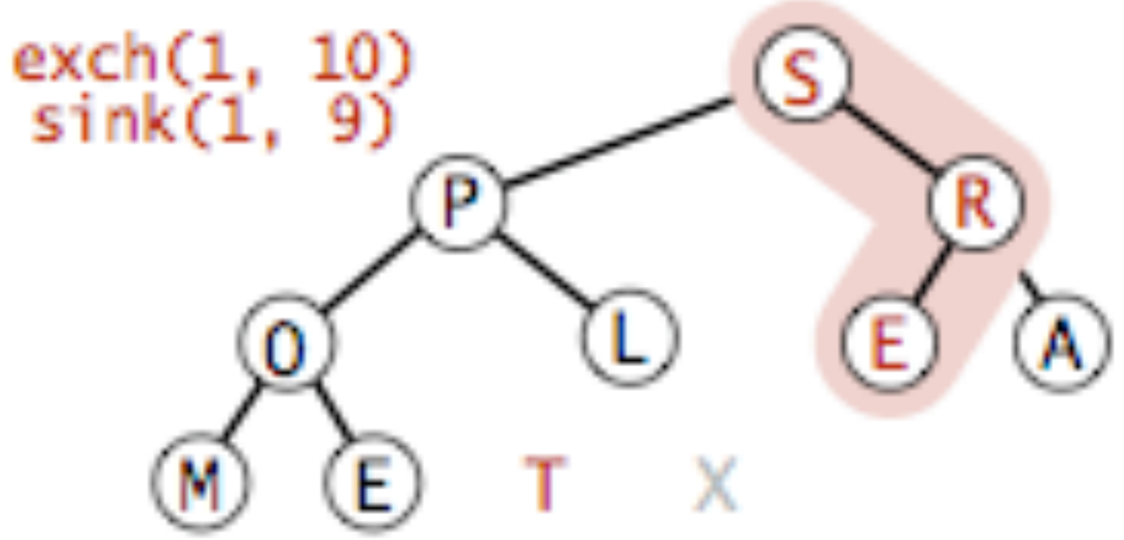
Sortdown example



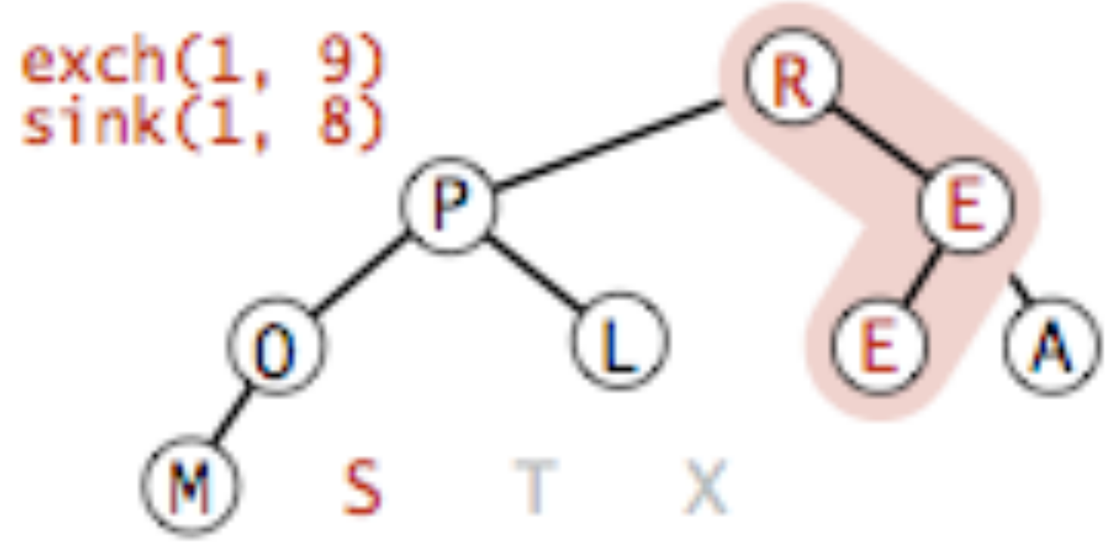


n = 11, so first we call swap (or “exch”) on (1, 11), then sink(1, 10)

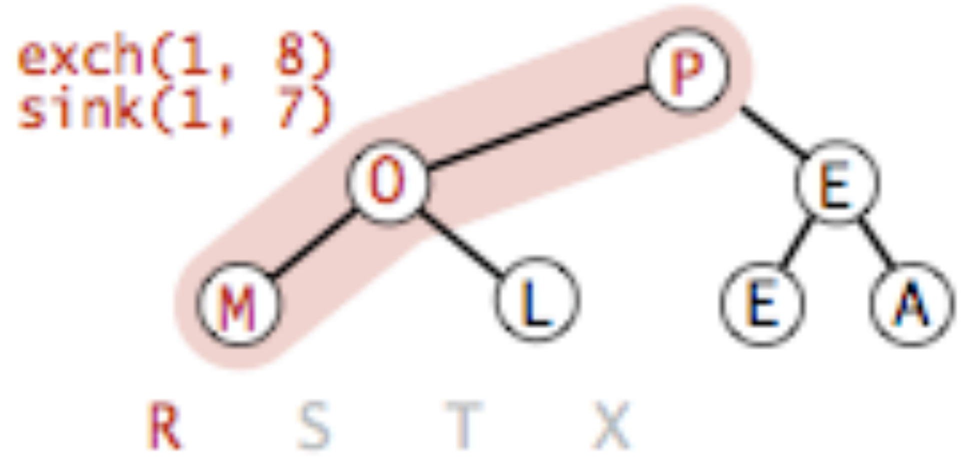
Swap X with E, sink down E -> T is new root
return X



swap T with E, sink down E -> S is new root
return T, X

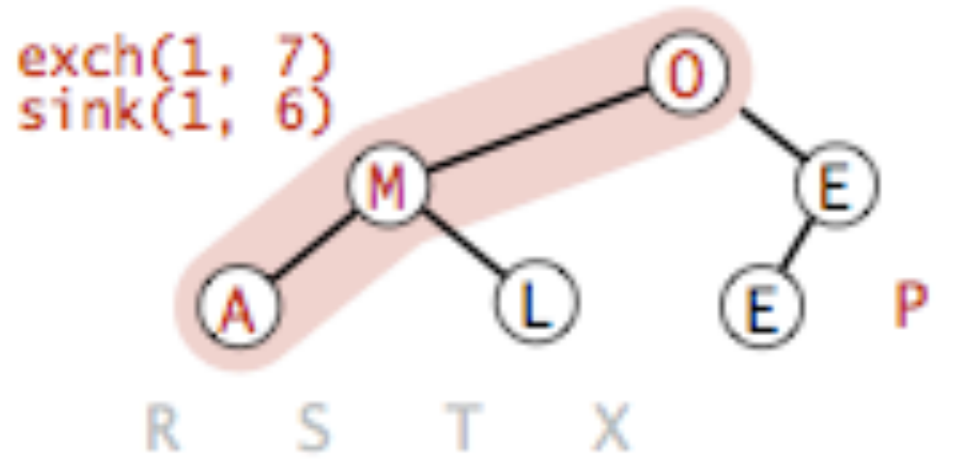


swap S with E, sink down E -> R is new root
return S, T, X



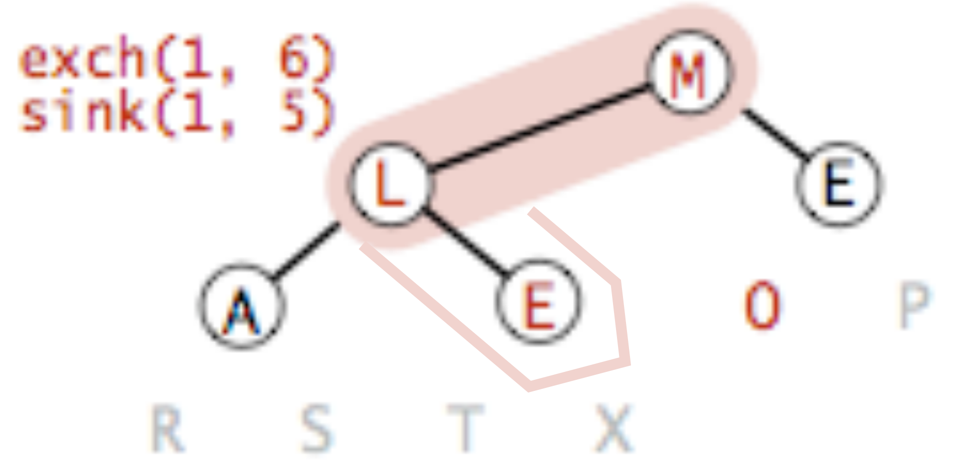
swap R with M, sink M -> P is new root

return R, S, T, X



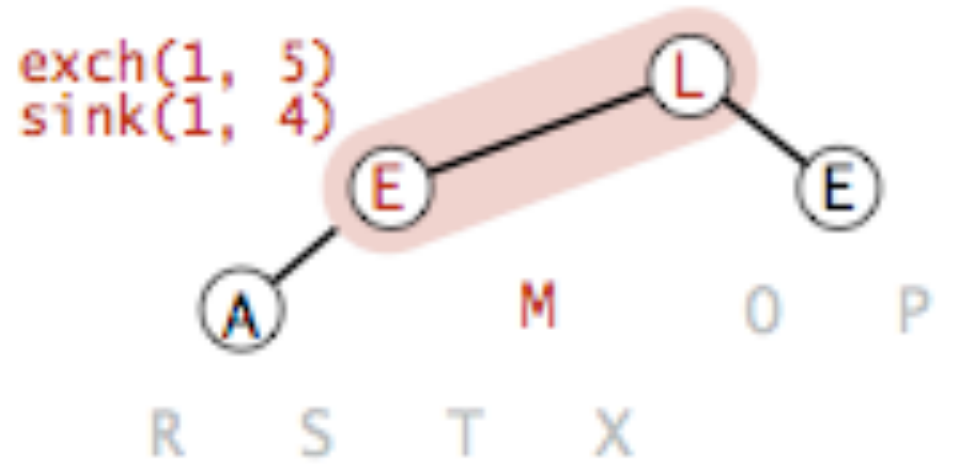
swap P with A, sink A -> O is new root

return P, R, S, T, X



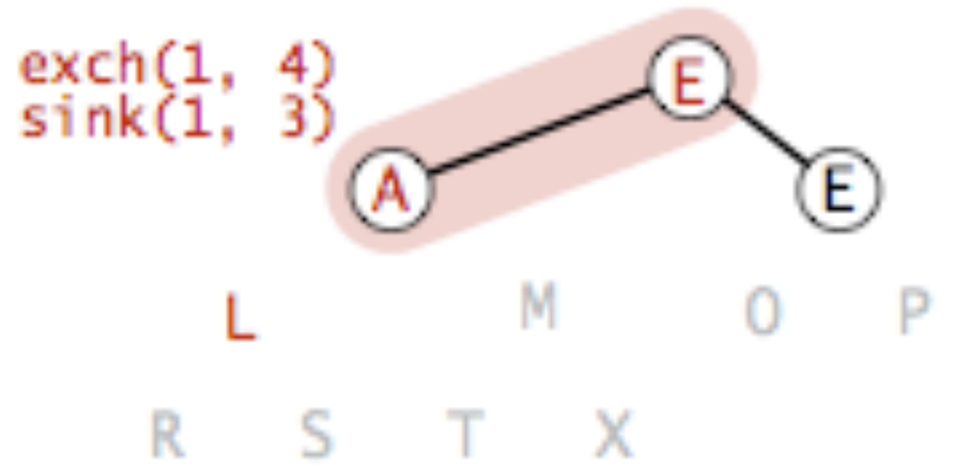
swap O with E, sink E -> M is new root

return O, P, R, S, T, X



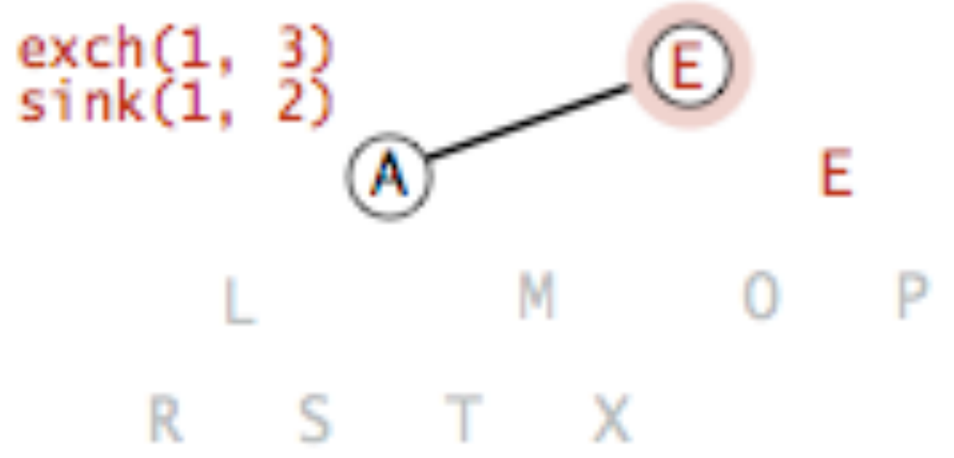
swap M with E, sink E -> L is new root

return M, O, P, R, S, T, X



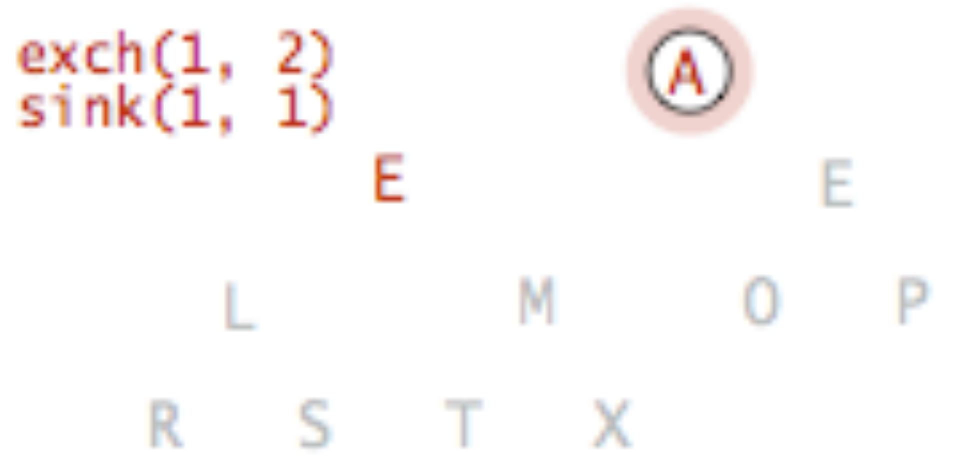
swap L with A, sink A -> E is new root

return L, M, O, P, R, S, T, X



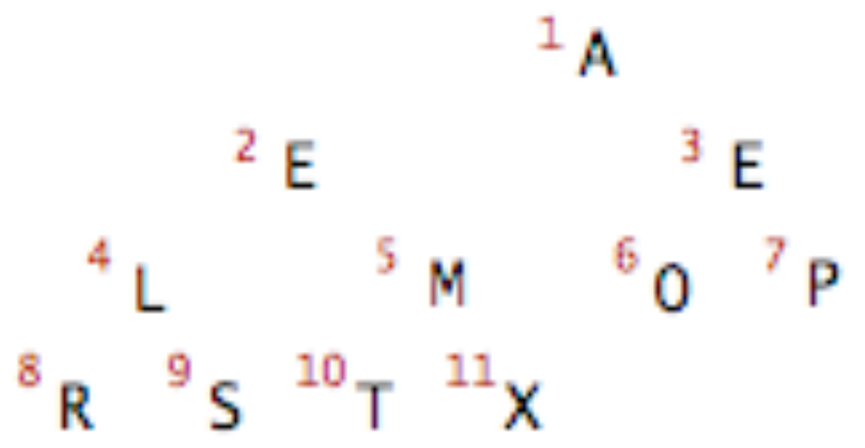
swap E with E, sink E (no action) -> E is new root

return E, L, M, O, P, R, S, T, X



swap E with A, sink A (A is just a single node, nothing to sink)

return E, E, L, M, O, P, R, S, T, X



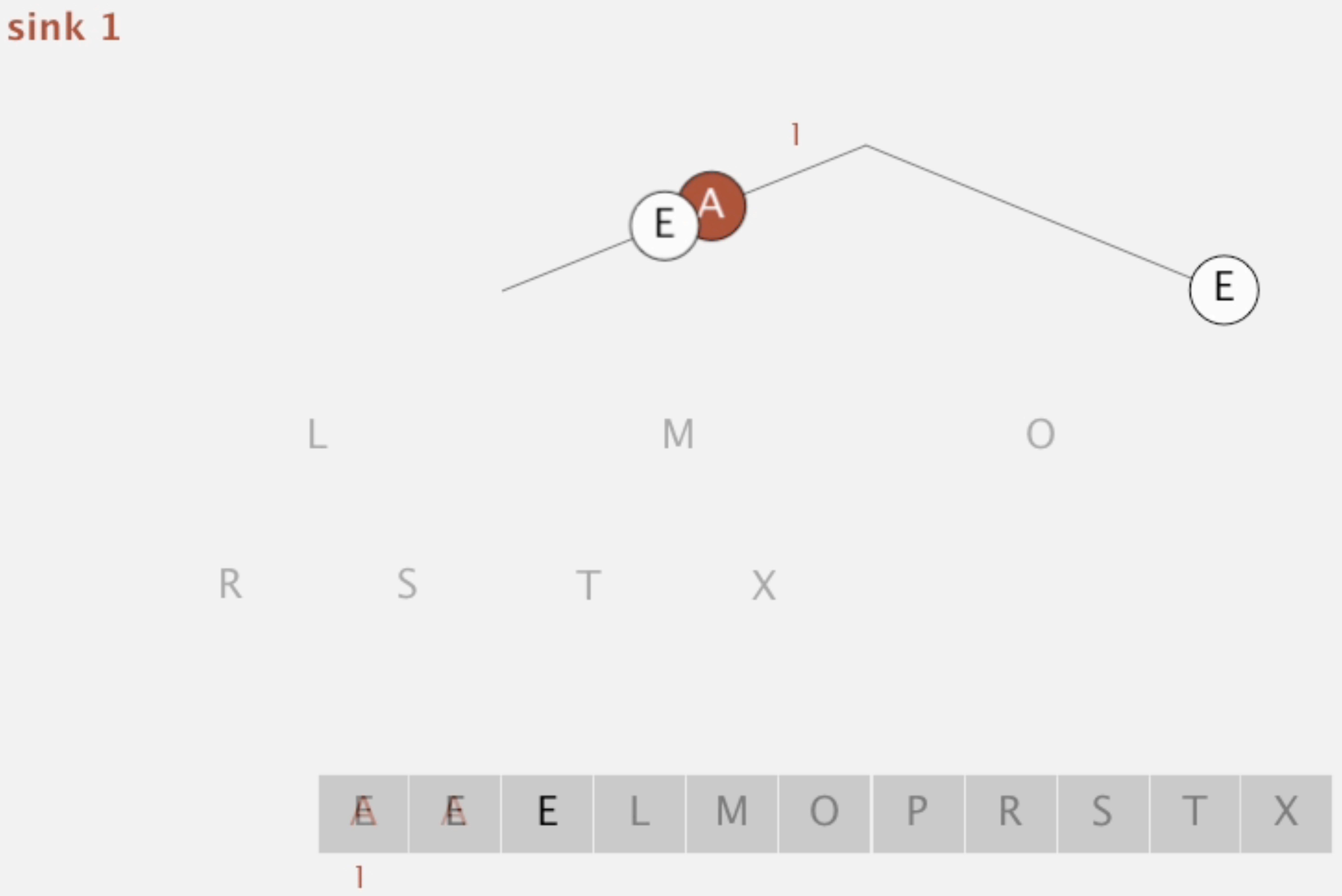
result (sorted)

because n = 1, we’re done

return A, E, E, L, M, O, P, R, S, T, X

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

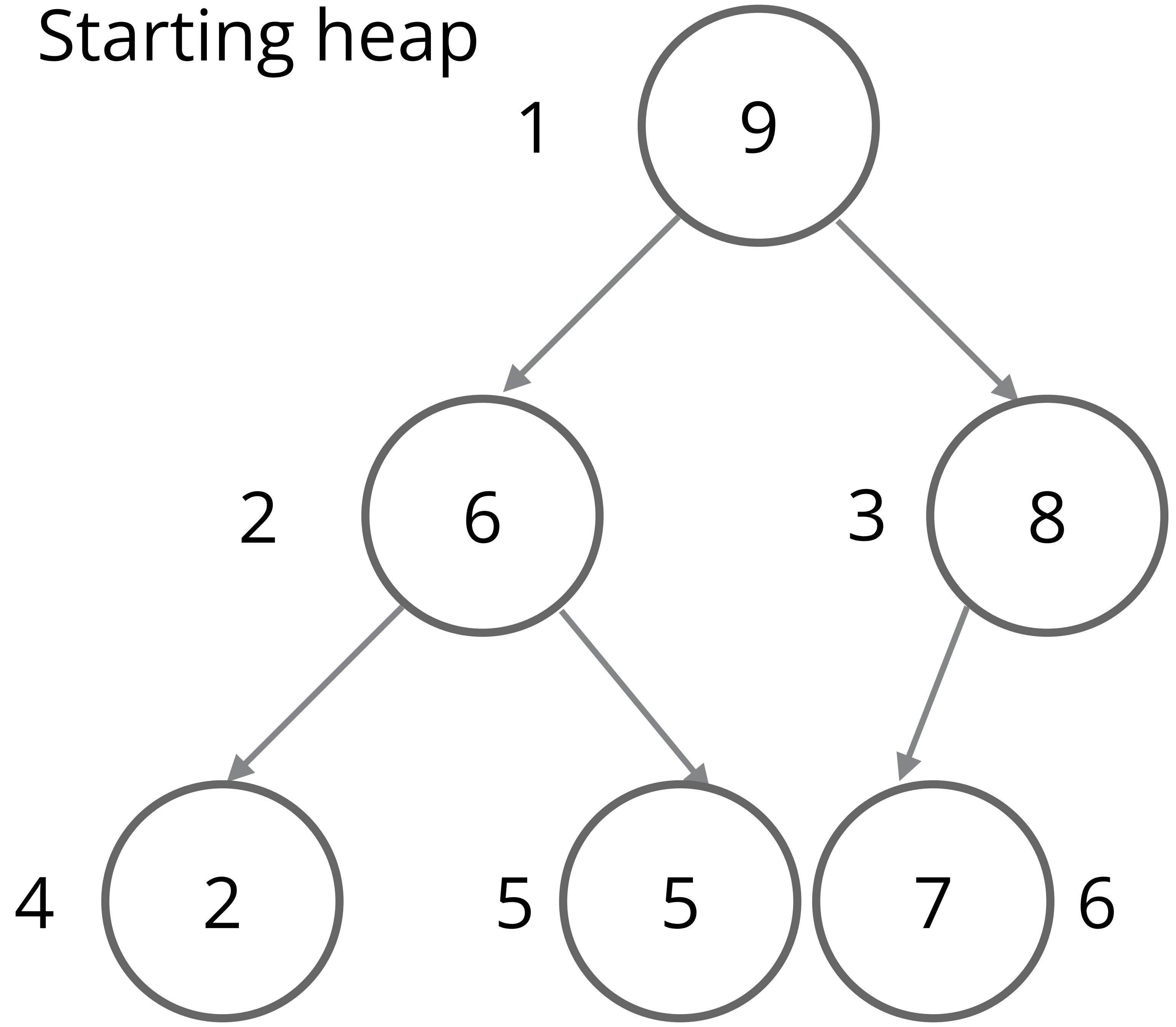


Worksheet time!

- Given the heap you constructed before, run the second step of heapsort, sortdown, to sort the array [2,9,7,6,5,8].

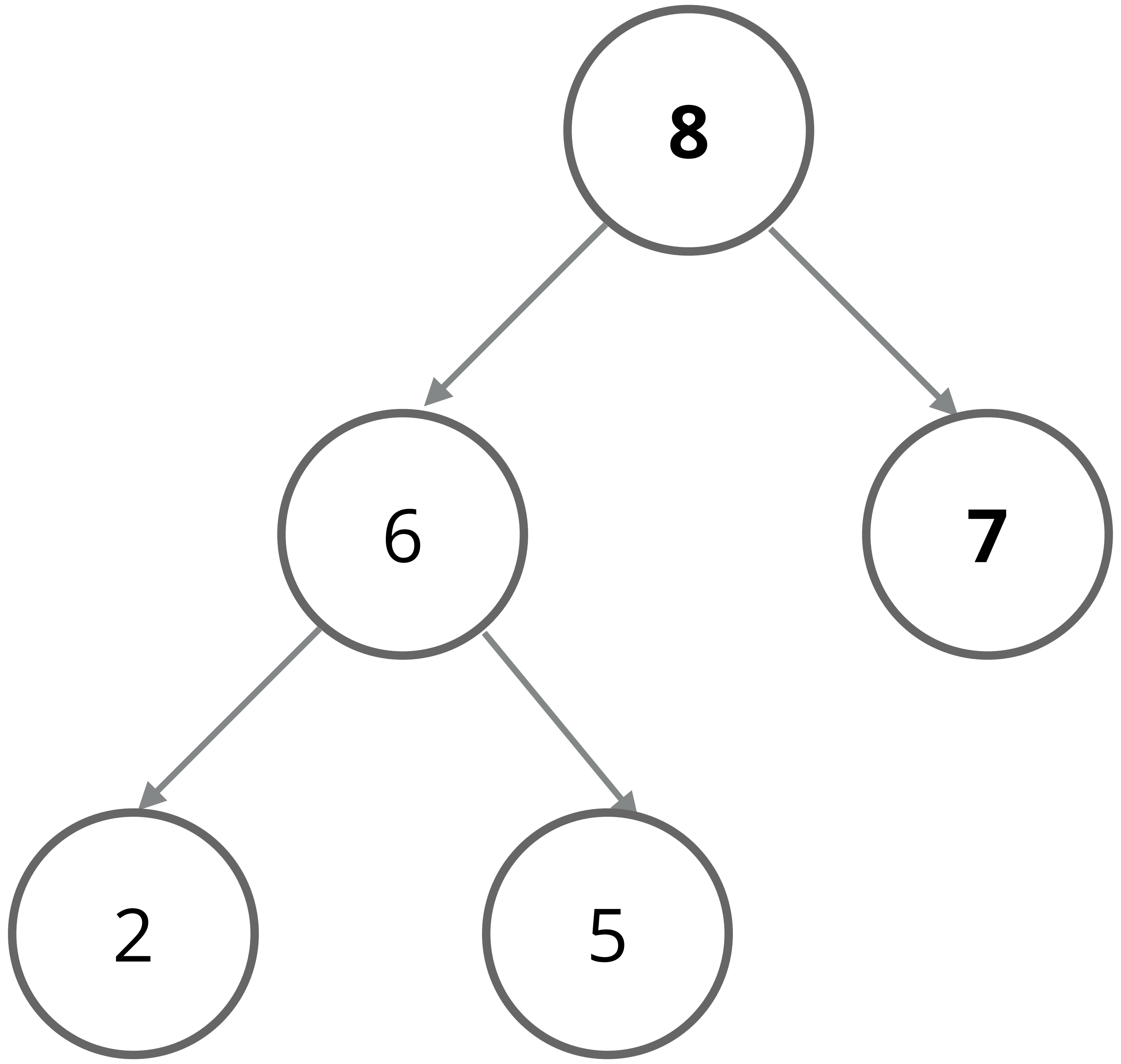
Worksheet answer

Starting heap



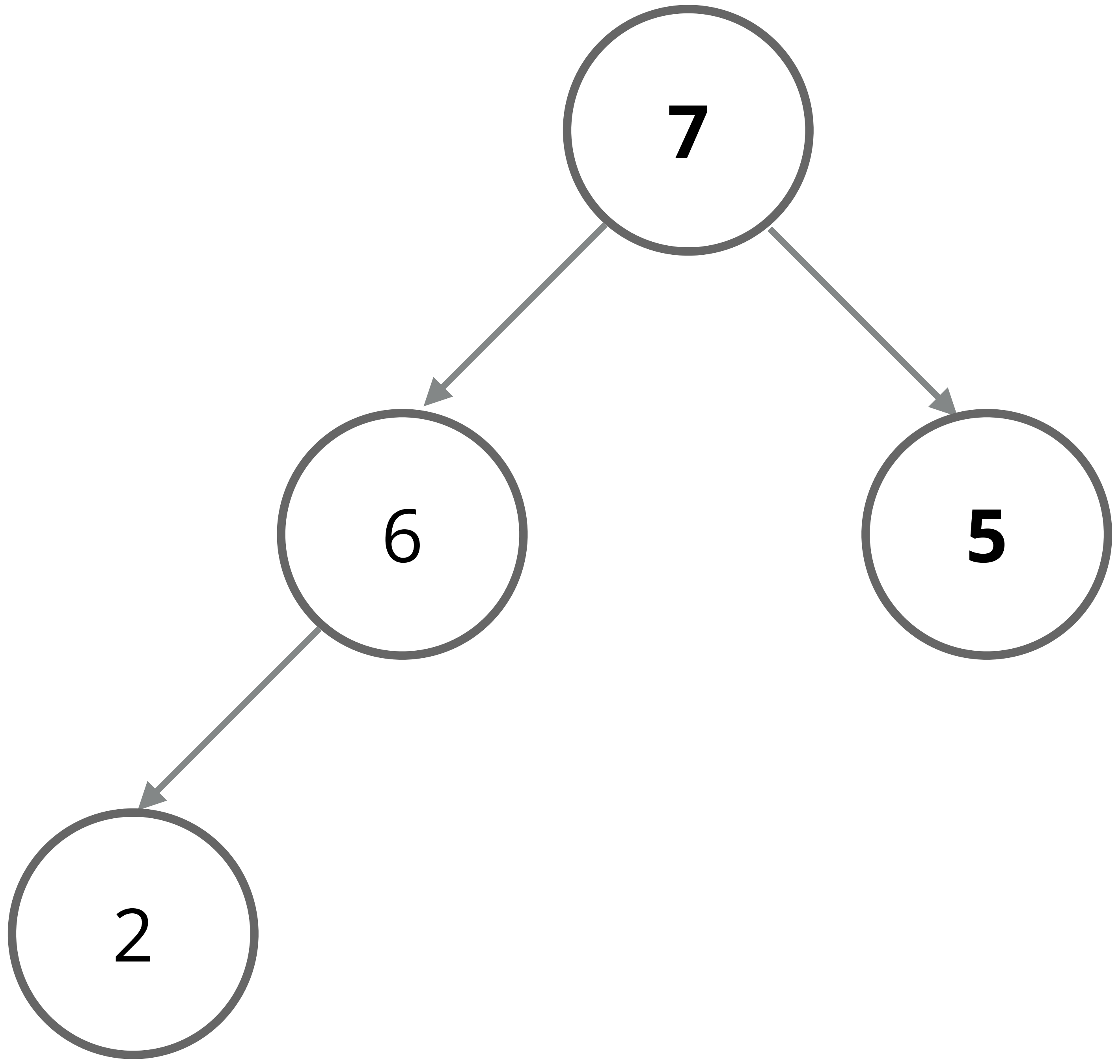
Return: 9

1. swap(1,6) sink(1,5) means swap 9 & 7 and sink 7



Return: 8, 9

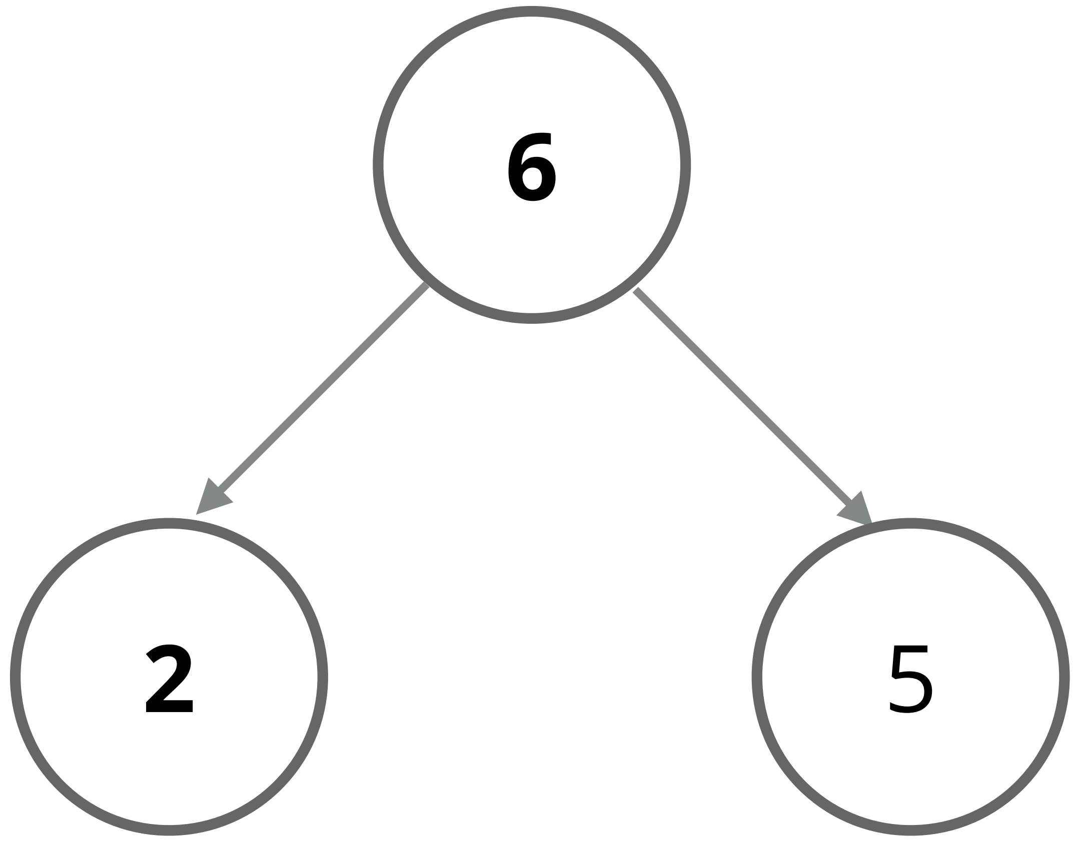
2. swap(1,5) sink(1,4) means swap 8 & 5 and sink 5



Worksheet answer

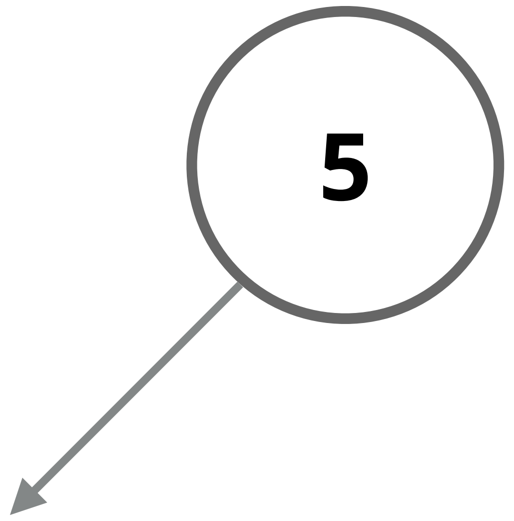
Return: 7, 8, 9

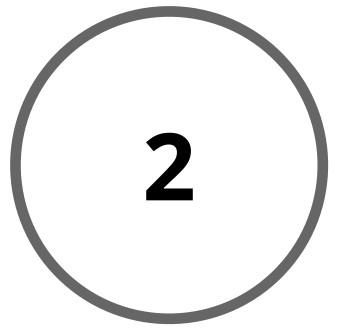
3. swap(1,4) sink(1,3) means
swap 7 & 2 and sink 2



Return: 6, 7, 8, 9

4. swap(1,3) sink(1,2) means
swap 6 & 5 and sink 5 (no sinking needed)





Return: 5, 6, 7, 8, 9

4. swap(1,2) sink(1,1) means
swap 5 & 2 and sink 2 (no sinking needed, single node)

2

5 6

7 8 9

5. done!

Return: 2, 5, 6, 7, 8, 9

Heapsort analysis

Heapsort analysis

- Heap construction (the fast version) makes $O(n)$ exchanges and $O(n)$ compares.
- Sortdown and therefore the entire heapsort $O(n \log n)$ exchanges and compares.
  - Each sink() is logn time, and we do n-1 sinks
- $O(n \log n)$ worst case. What about best case? Average case?
  - The same
- In-place (no need to copy anything).
- Not stable (we are swapping elements)

Heapsort analysis

- Review:
  - Mergesort: not in place, requires linear extra space.
  - Quicksort: quadratic time in worst case.
- Heapsort is optimal both for time and space in terms of Big-O, but:
  - Inner loop is longer than quicksort because of sink.
  - Poor use of cache because it accesses memory in non-sequential manner, jumping around the heap/array (more in CS105).
- In general, quicksort is preferred when it comes to speed, and mergesort is preferred when it comes to stability.

Sorting: we're done!

Which Sort	In place	Stable	Best	Average	Worst	Memory	Remarks
Selection	X		$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$	n exchanges
Insertion	X	X	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$	Fastest if almost sorted or small
Merge		X	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(n)$	Guaranteed performance; stable
Quick	X		$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$	$\Theta(\log n)$	$n \log n$ probabilistic guarantee; fastest in practice
Heap	X		$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$	$\Theta(1)$	Guaranteed performance; in place

Lecture 16 wrap-up

- HW6: On Disk sort due 11:59pm tonight
- The next lecture, dictionaries, will be the last thing on Checkpoint 2

Resources

- Reading from textbook: 2.5 (336-344)
- Heapsort visualization: https://algostructure.com/sorting/heapsort.php
- More visualization to compare the n and nlogn create heap approaches: https://visualgo.net/en/heap
- Practice problems behind this slide

Practice Problem 1

- Suppose that the sequence 16, 18, 9, 15, *, 18, *, *, 9, *, 20, *, 25, *, *, *, 17, 21, 5, *, *, *, 21, *, 5 (where a number means insert and an asterisk means delete the maximum) is applied to an initially empty priority queue. Give the sequence of numbers returned by the delete maximum operations.

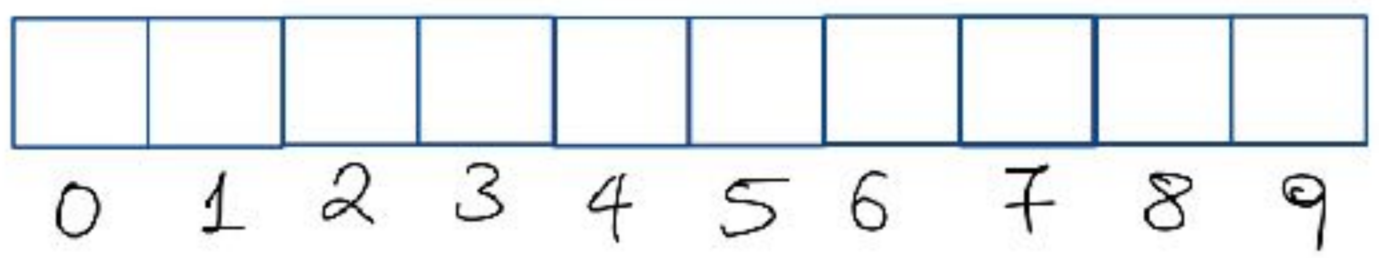
ANSWER 1

- Suppose that the sequence 16, 18, 9, 15, *, 18, *, *, 9, *, 20, *, 25, *, *, *, 17, 21, 5, *, *, *, 21, *, 5 (where a number means insert and an asterisk means delete the maximum) is applied to an initially empty priority queue. Give the sequence of numbers returned by the delete maximum operations.
- 18, 18, 16, 15, 20, 25, 9, 9, 21, 17, 5, 21

Code for priority queue option 1: Unordered array

```
public class UnorderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;       // elements
    private int n;          // number of elements

    // set inititial size of heap to hold size elements
    public UnorderedArrayMaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
        n = 0;
    }

    public boolean isEmpty()   { return n == 0; }
    public int size()          { return n;      }
    public void insert(Key x)  { pq[n++] = x;   }

    public Key delMax() {
        int max = 0;
        for (int i = 1; i < n; i++){
            if (pq[max].compareTo(pq[i]) < 0) {
                 max = i;
            }
        }
        Key temp = pq[max];
        pq[max] = pq[n-1];
        pq[n-1] = temp;

        return pq[--n];
    }
}
```

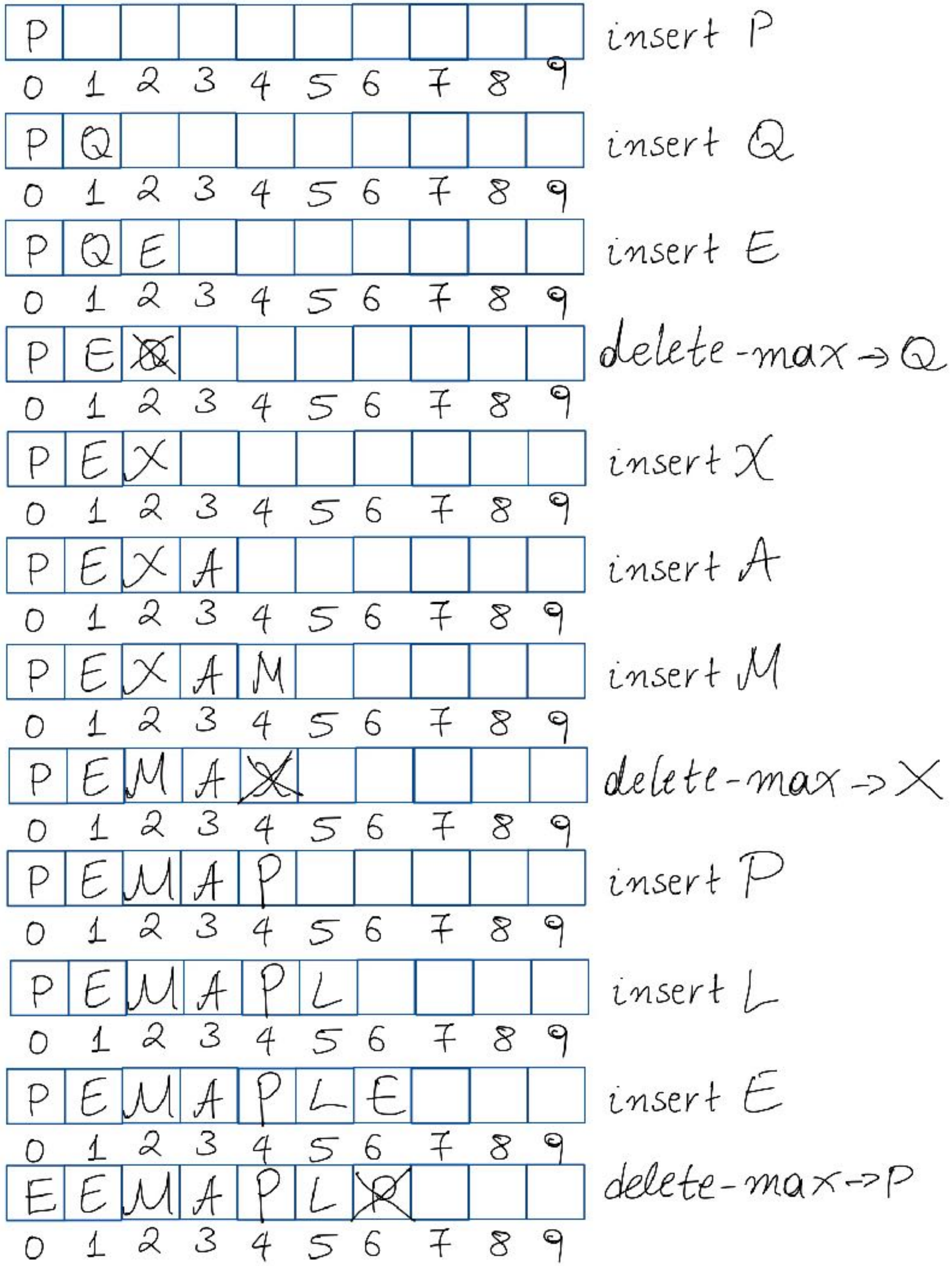
Practice problem 2



Given an empty array of capacity 10, perform the following operations in a priority queue based on an unordered array (lazy approach):

1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

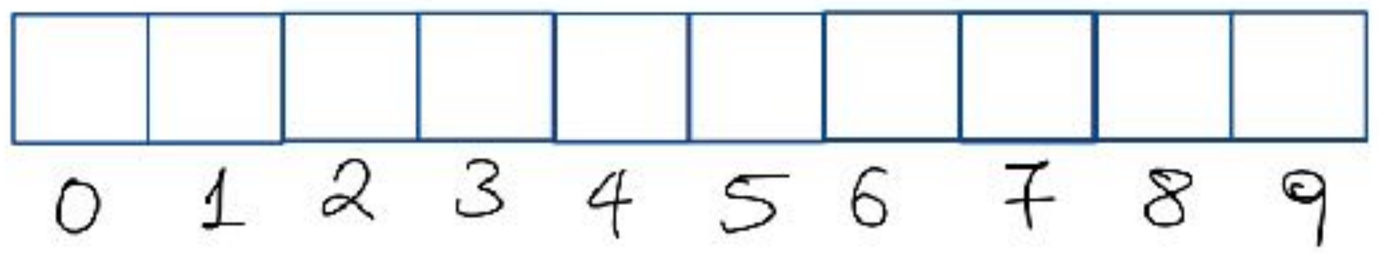
Answer 2


P
0 1 2 3 4 5 6 7 8 9
insert P
P Q
0 1 2 3 4 5 6 7 8 9
insert Q
P Q E
0 1 2 3 4 5 6 7 8 9
insert E
P E Q
0 1 2 3 4 5 6 7 8 9
delete-max → Q
P E X
0 1 2 3 4 5 6 7 8 9
insert X
P E X A
0 1 2 3 4 5 6 7 8 9
insert A
P E X A M
0 1 2 3 4 5 6 7 8 9
insert M
P E M A X
0 1 2 3 4 5 6 7 8 9
delete-max → X
P E M A P
0 1 2 3 4 5 6 7 8 9
insert P
P E M A P L
0 1 2 3 4 5 6 7 8 9
insert L
P E M A P L E
0 1 2 3 4 5 6 7 8 9
insert E
E E M A P L P
0 1 2 3 4 5 6 7 8 9
delete-max → P

Priority queue option 2: Ordered array

```
public class OrderedArrayMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq;          // elements
    private int n;             // number of elements

    // set inititial size of heap to hold size elements
    public OrderedArrayMaxPQ(int capacity) {
        pq = (Key[]) (new Comparable[capacity]);
        n = 0;
    }


    public boolean isEmpty() { return n == 0;  }
    public int size()        { return n;       }
    public Key delMax()      { return pq[--n]; }

    public void insert(Key key) {
        int i = n-1;
        while (i >= 0 && key.compareTo(pq[i]) < 0) {
            pq[i+1] = pq[i];
            i--;
        }
        pq[i+1] = key;
        n++;
    }
}
```

Practice Problem 3



1. Insert P
2. Insert Q
3. Insert E
4. Delete max
5. Insert X
6. Insert A
7. Insert M
8. Delete max
9. Insert P
10. Insert L
11. Insert E
12. Delete max

Given an empty array of capacity 10, perform the following operations in a priority queue based on an ordered array (eager approach):

Answer 3

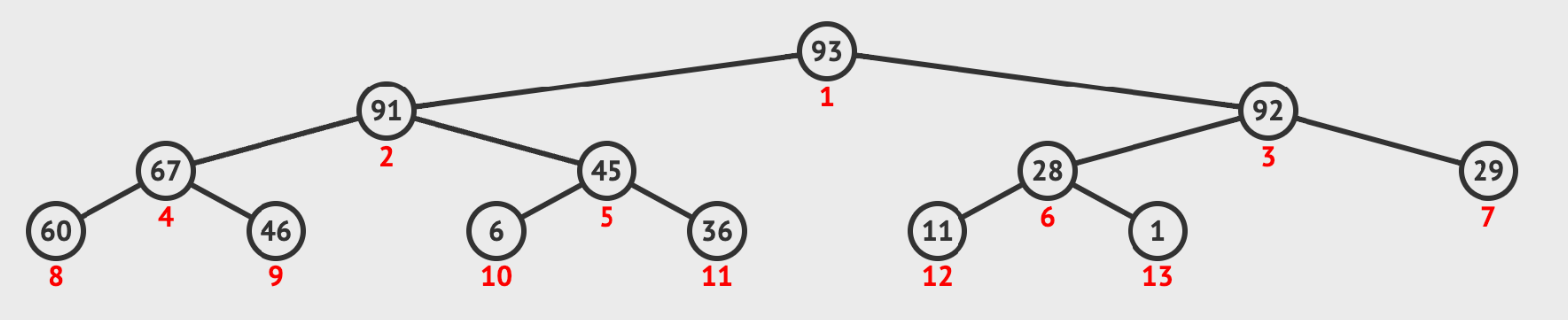
0	1	2	3	4	5	6	7	8	9	
P										insert P
P	Q									insert Q
E	P	Q								insert E
E	P	~~Q~~								delete-max → Q
E	P	X								insert X
A	E	P	X							insert A
A	E	M	P	X						insert M
A	E	M	P	~~X~~						delete-max → X
A	E	M	P	P						insert P
A	E	L	M	P	P					insert L
A	E	E	L	M	P	P				insert E
A	E	E	L	M	P	~~P~~				delete-max → P

Practice Problem 4: Heapsort

- Given the array [93,36,1,46,91,92,29,60,67,6,45,11,28], apply heap sort. Visualize what the heap will initially look like (apply the O(n) heap construction algorithm) and visualize it during sortdown as well.

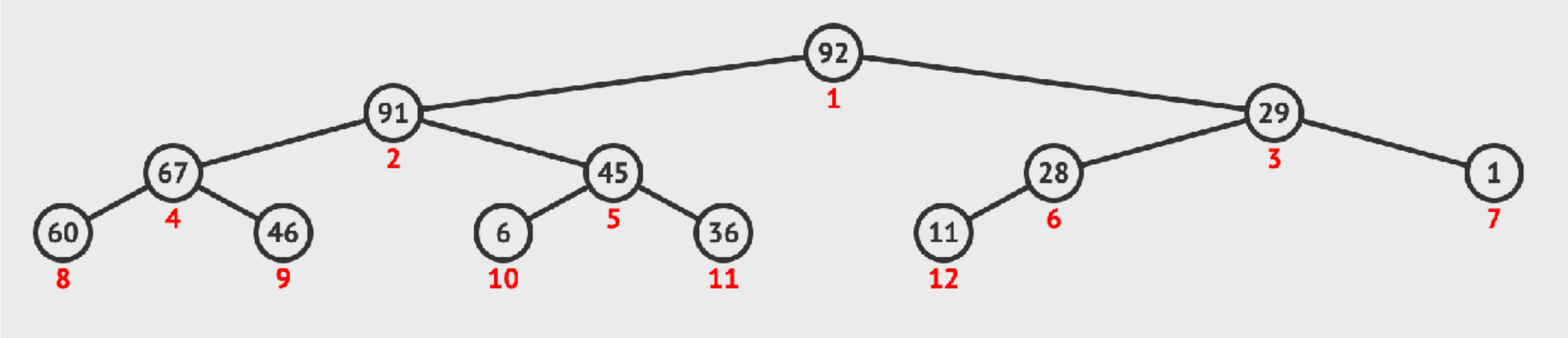
ANSWER 4

- Given the array [93,36,1,46,91,92,29,60,67,6,45,11,28], apply heap sort. Visualize what the heap will initially look like (apply the O(n) heap construction algorithm) and visualize it during sortdown as well.
- Heap construction step:

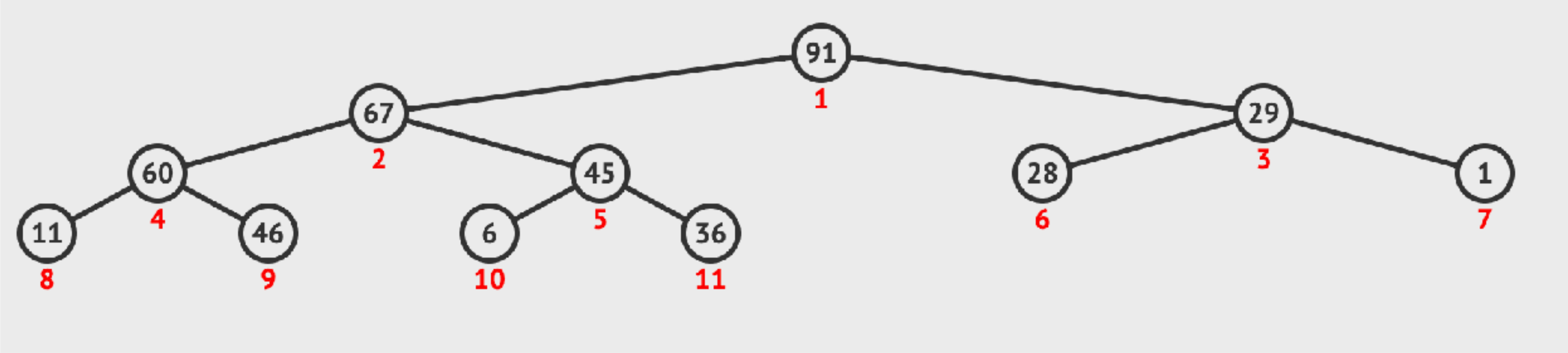


ANSWER 4: sortdown

- Extract max (93)

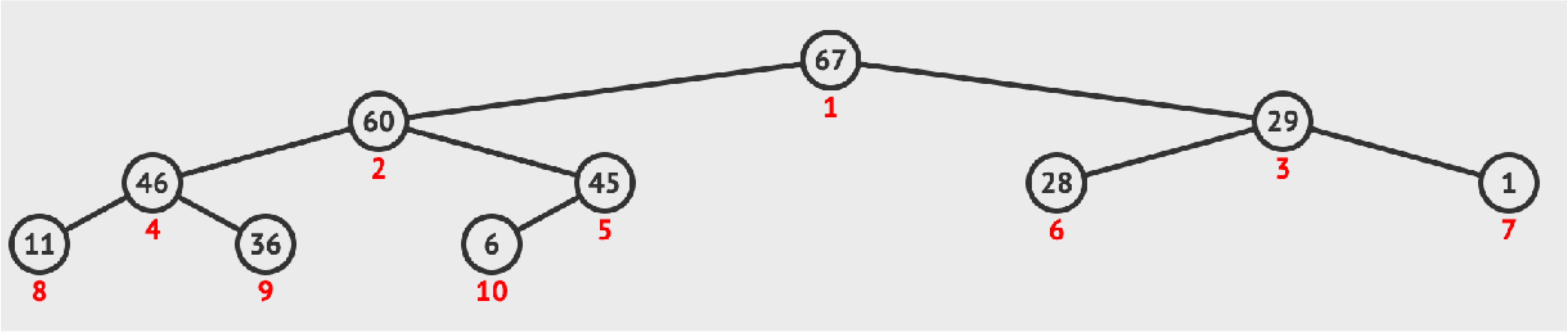


- Extract max (92)

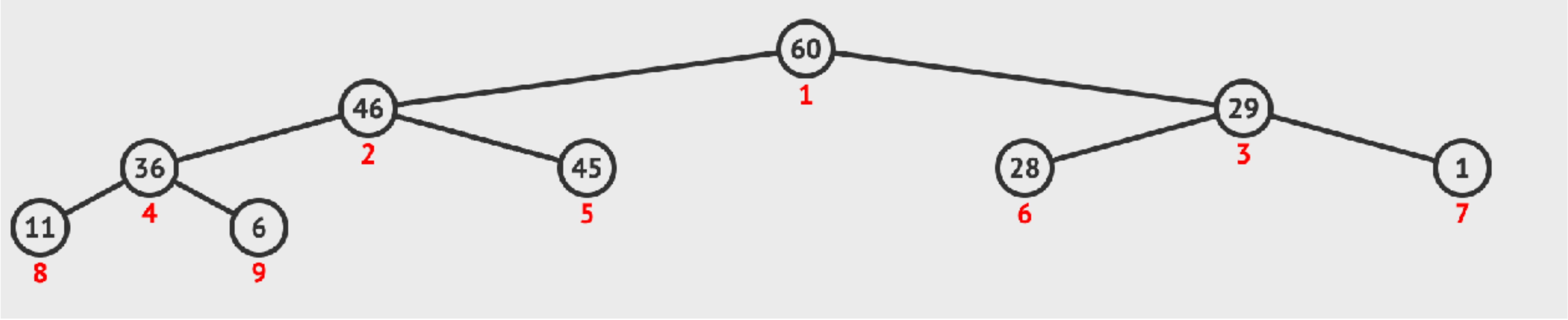


ANSWER 4

- Extract max (91)

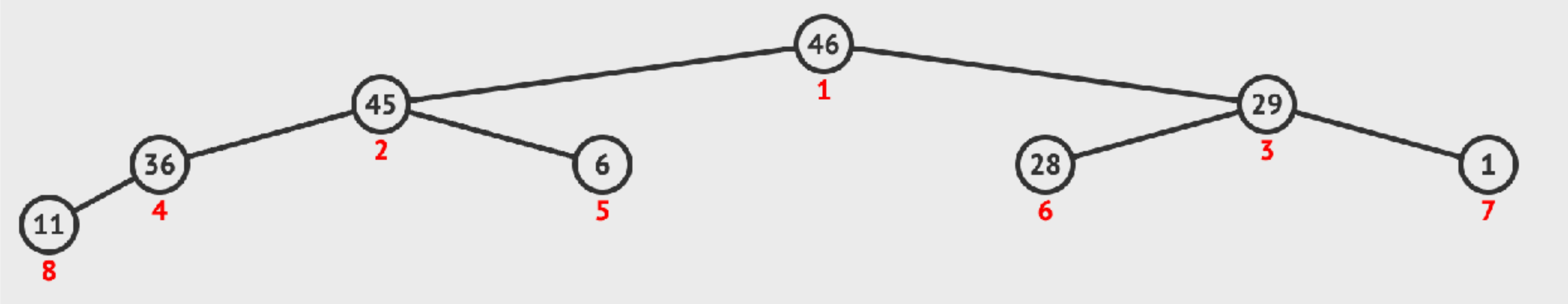


- Extract max (67)

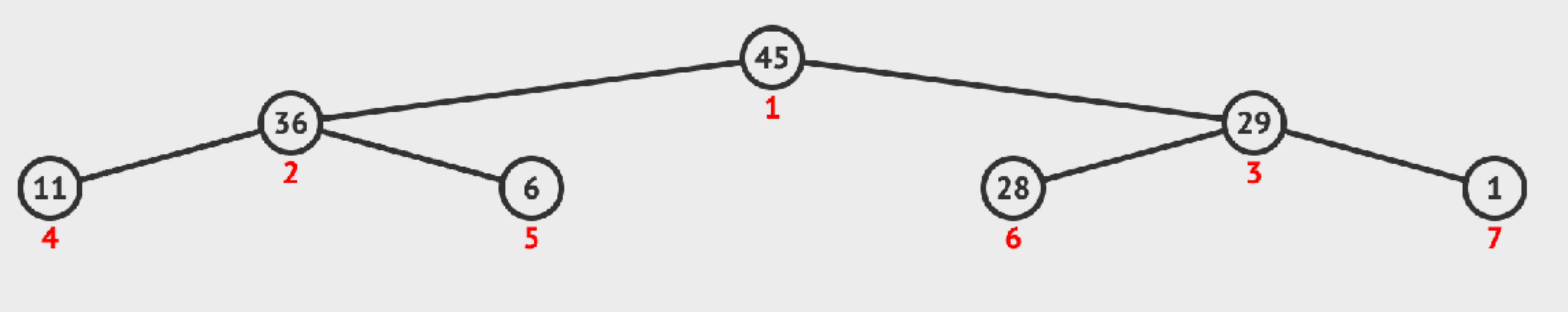


ANSWER 4

- Extract max (60)

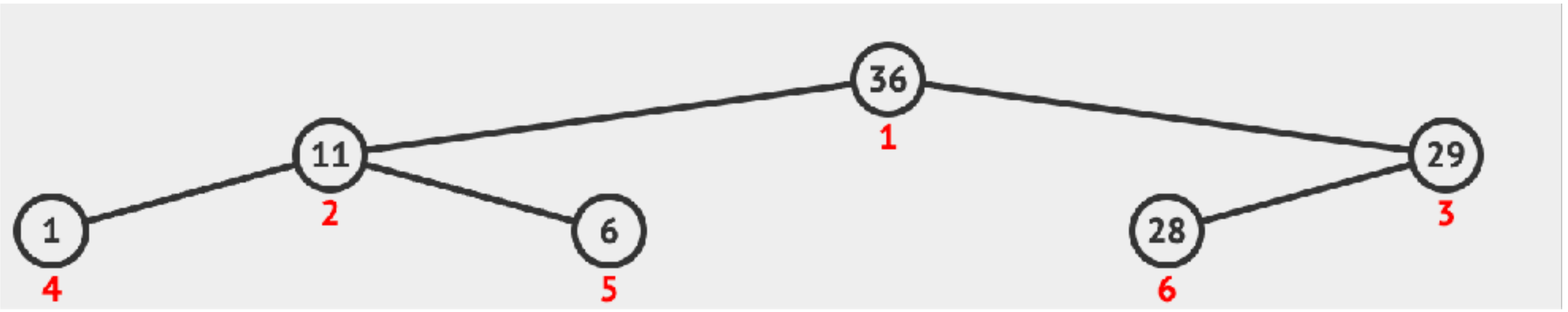


- Extract max (46)

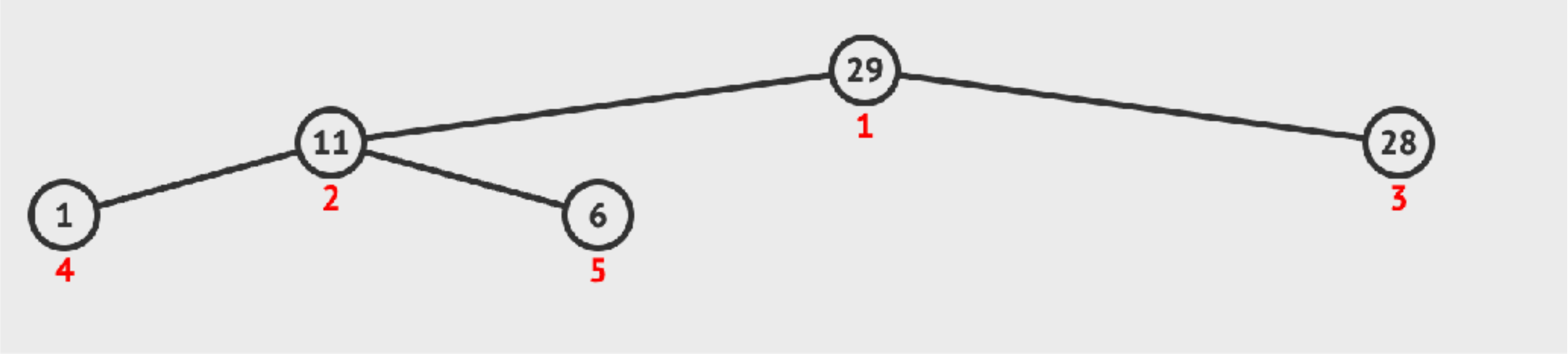


ANSWER 4

- Extract max (45)

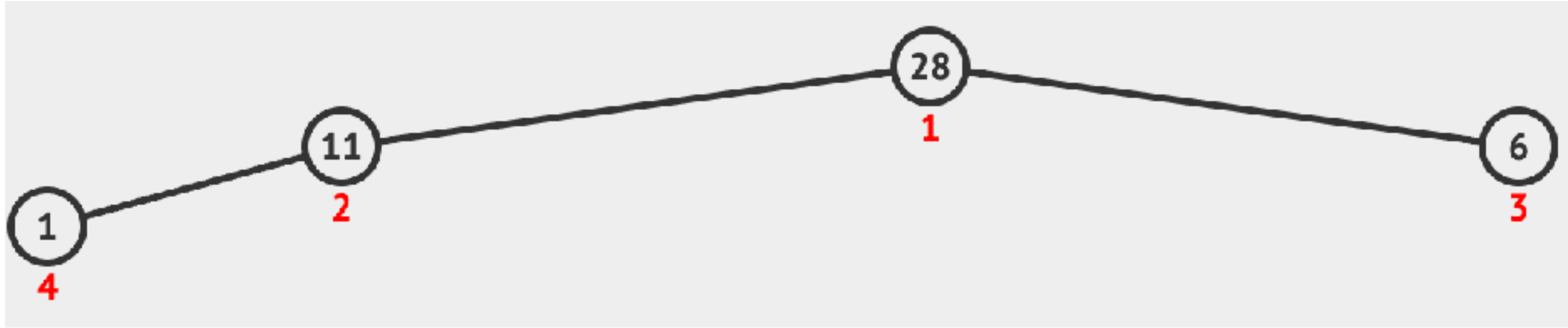


- Extract max (36)

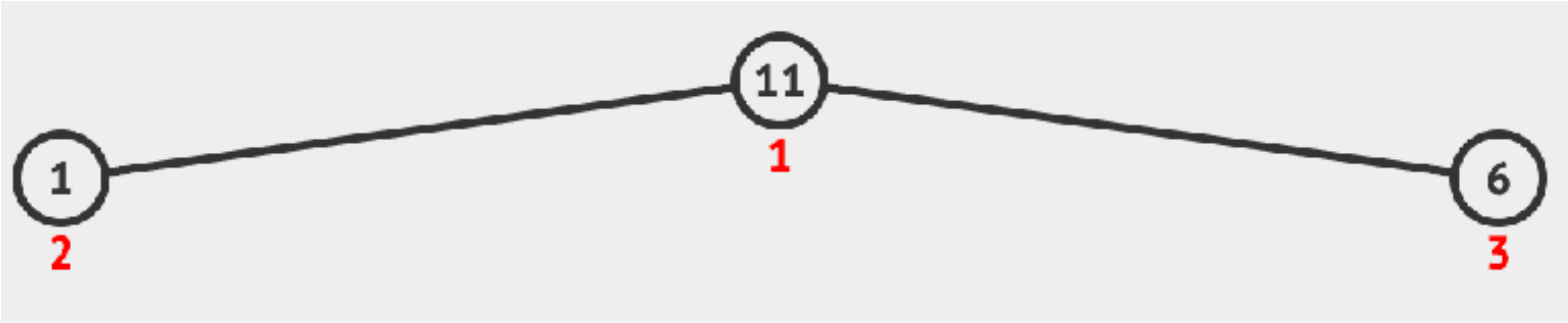


ANSWER 4

- Extract max (29)



- Extract max (28)



ANSWER 4

- Extract max (11)

- Extract max (6)

- Extract max (1)