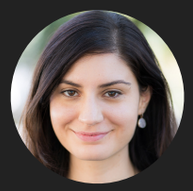
SORTING

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

17: Heapsort



Alexandra Papoutsaki
she/her/hers

In the previous two lectures, we saw a new way of organizing data: rather than thinking of them in a linear fashion, we considered hierarchical relationships that were modeled as trees: sets of nodes that have a parent-child relationship. We saw binary trees (trees with at most 2 children), traversed them in 4 ways (pre-, in-, post, and level-order), learned about heaps (array representations of complete heap-ordered binary trees, where every node is larger than or equal to both of its children), and talked about priority queues and why heaps and priority queues are considered synonymous. We will use these blocks to discuss a new sorting algorithm, heap sort which will complete the sorting unit of the course.

Lecture 17: Heapsort

- Heapsort

Ready for heapsort?

Basic plan for heap sort

- 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.

- Heapsort is a very simple algorithm: given an array of n elements that we need to sort, we can use any priority queue implementation (but, as we saw, it makes sense to use a binary heap) to enter each of the n elements one by one to the priority queue and when done, repeatedly remove and return the maximum.
- We will discuss an efficient way of constructing the heap momentarily but once our n insertions are complete, we know that the largest element is at the root. This triggers the sortdown process: We will delete the largest element and return it and proceed with the deletion of the second largest element and so on.
- By the end, we will have returned the n elements in non-decreasing order. If we had a min-heap, where the minimum rather than the maximum element is stored at the root, similarly, during the sortdown step we would return the elements in non-increasing order.

$O(n \log n)$ Heap construction

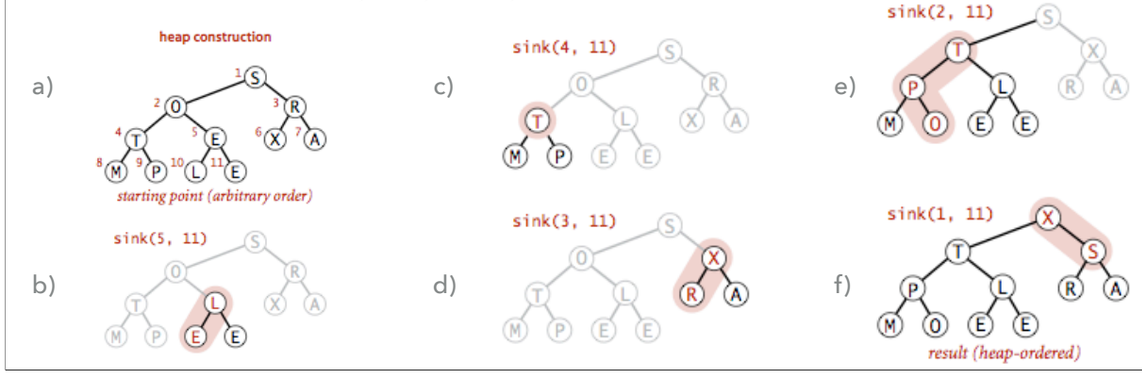
- Insert n elements, one by one, swim up to their appropriate position.
- We can do better!
- Key insight: After `sink(a,k,n)` completes, the subtree rooted at k is a heap.

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

- A simple way of constructing the heap would be to insert the n elements one by one and swim them up their appropriate position in O(nlogn) time (n insertions, each O(logn) time). We can do better than that. Instead, we will use a smarter implementation that starts from right to left and uses sink() to make subheaps as we go. This much better implementation (runs in O(n) instead of O(nlogn) time) is based on the following key insight: Every position in the array is the root of a small subheap; sink() works on such subheaps, as well. If the two children of a node are heaps, then calling sink() on that node makes the subtree rooted there also a heap.

$O(n)$ Heap construction

- Insert all nodes as is in indices 1 to n. We will turn this binary tree into a heap.
- Ignore all leaves (indices n/2+1,...,n). Sink each internal node
-
```
for(int k = n/2; k >= 1; k--)
      sink(a, k, n);
```



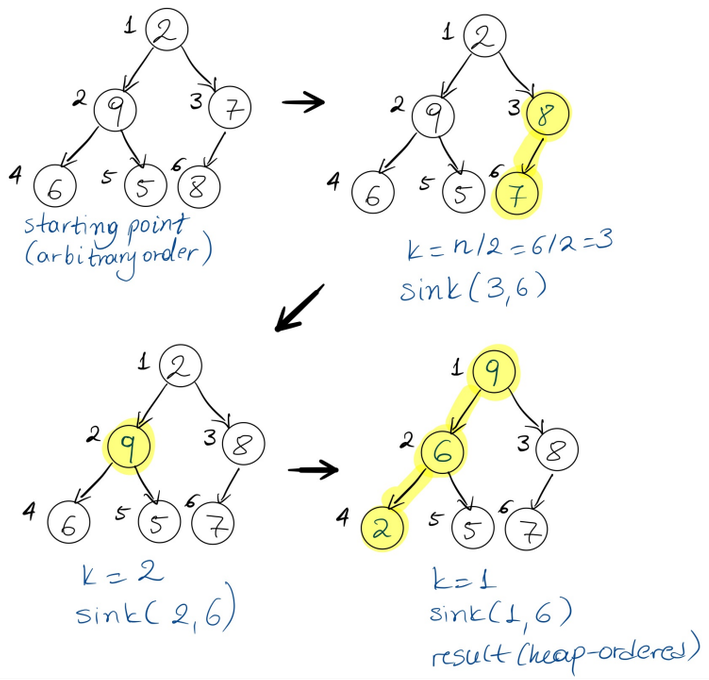
- Let's see this in action by sorting the array [S,O,R,T,E,X,A,M,P,L,E].
- a) We start by creating a complete tree where the first element is placed at index 1, the second at index 2 and so on. Our goal is to turn this complete tree into a heap.
- Although we won't prove it, all leaves appear in indices (n/2)+1,…n. We can ignore leaves, since they are already trivial heaps, and start with the right-most internal node which is located at position n/2.
- b) We will start by sinking the k=n/2 node and repeatedly reduce k by 1. Here, we have n=11 and k starts at 5. We sink the element at index 5, that is E, which is exchanged with L. We now have a subheap rooted at index 5.
- c) We decrease k by 1, having now k=4 and sink the element at index 4, that is T. Since T is larger than both of its children, we know that the subtree rooted at 4 is a subheap.
- d) We decrease k by 1, having now k=3 and sink the element at index 3, that is R. R is exchanged with the larger child, X, and the subtree rooted at index 3 is a subheap.
- e) We decrease k by 1, having now k=2 and sink the element at index 2, that is O which is exchanged first with T and then with P. The subtree rooted at index 2 is a subheap.
- f) We decrease k by 1, having now k=1 and sink the element at index 1, that is S which is exchanged with the larger child X. The entire tree now has been heap-ordered and the root contains the maximum element, X.

Practice Time - Worksheet #17

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

To practice, let’s run the first step of heapsort, heap construction, on the array [2,9,7,6,5,8]

Answer: Heap construction



This is how your heap construction should look like. We start with placing the elements in arbitrary level-order. Starting at the first internal node (index 3), we repeatedly sink every internal node all the way to the root, resulting to a heap-ordered complete tree - a binary heap.

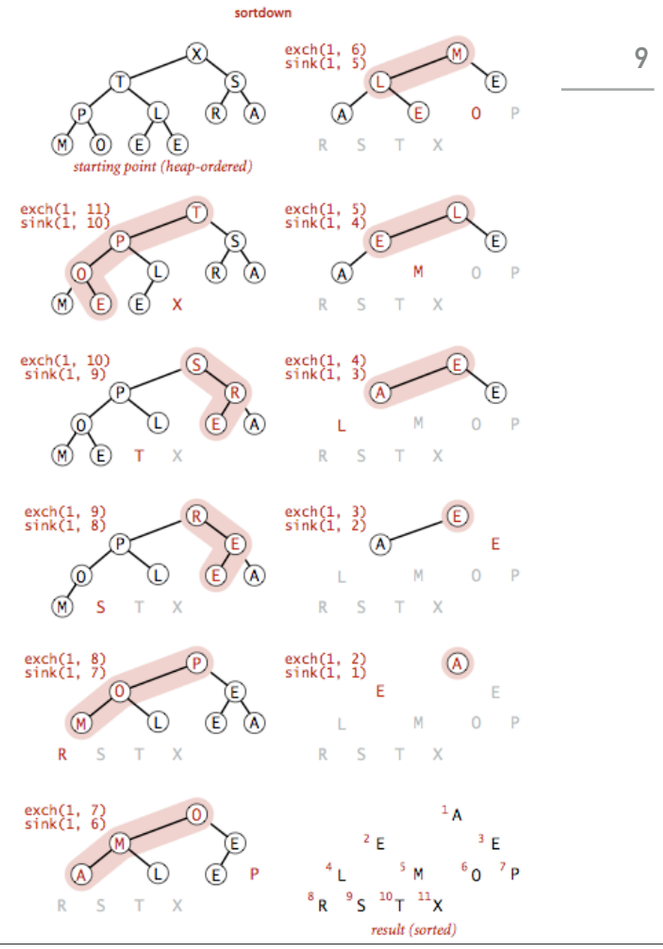
Sortdown

- Remove the maximum, one at a time, but leave in array instead of nulling out.
- ```
  while(n>1){
    exch(a, 1, n--);
    sink(a, 1, n);
  }
  ```
- Key insight: After each iteration the array consists of a heap-ordered subarray followed by a sub-array in final order.

- Once we have constructed the heap, the second step of the heap sort algorithm is the sortdown process, where we remove and return the maximum element (found at the root). If you remember, delete max in heaps exchanges the root with the last element (found in position n), sinks it till it finds its appropriate position, and finally nulls it out.
- We will skip this last step and instead we'll only apply the exchange and sink step and will use the n counter to keep track of how many elements we have left in our binary heap.
- The key insight here is that after each iteration of sortdown, we're left with a heap-ordered subarray of k elements and the remaining n-k elements are provided in the final sorted order.
  ```

Sortdown

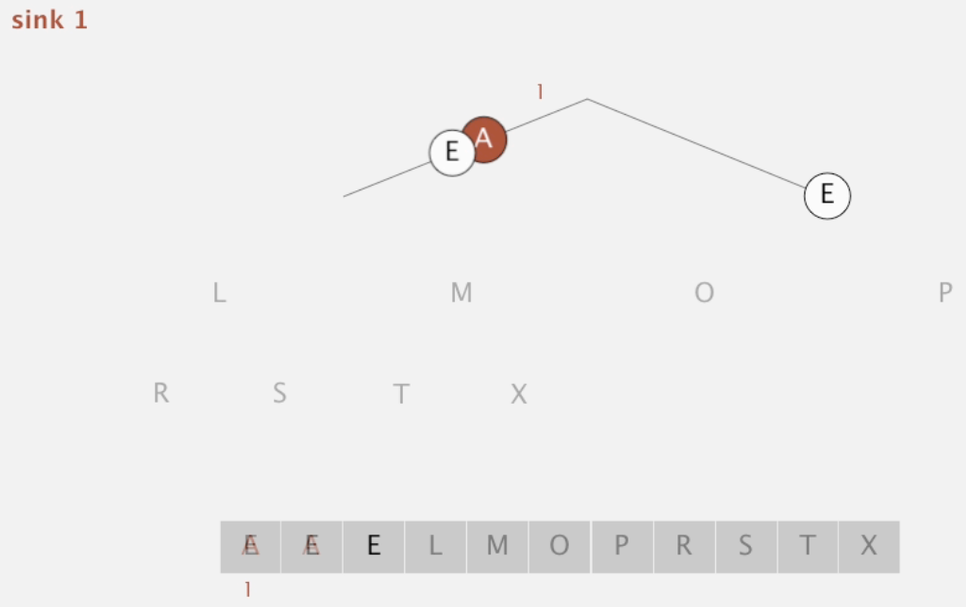
```
while(n>1){
   exch(a, 1, n--);
   sink(a, 1, n);
 }
```



- Let’s see sortdown through the same example. Earlier, we constructed a heap for the [S,O,R,T,E,X,A,M,P,L,E] array. Our heap will have the global maximum at its root.
- Sine we have 11 elements, we will start with exchanging the 1-index element (X) with the 11-index element (E) and then sink E. At the end of this step, we will be left with a heap of 10 elements and the 11-index element will be the largest element of the original array.
- We will continue by exchanging the new 1-index element (T) with the 10-index element (E) and then sink E. At the end of this step, we will be left with a heap of 9 elements and the remaining subarray (indices 10 and 11) will consist of the two largest elements of the original array.
- We will continue by exchanging the new 1-index element (S) with the 9-index element (E) and then sink E. At the end of this step, we will be left with a heap of 8 elements and the remaining subarray (indices 9, 10 and 11) will consist of the three largest elements of the original array.
- The same process is applied on each of the remaining elements until we are left with an empty heap on the left and an array of 11 sorted elements in non-decreasing order.

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.



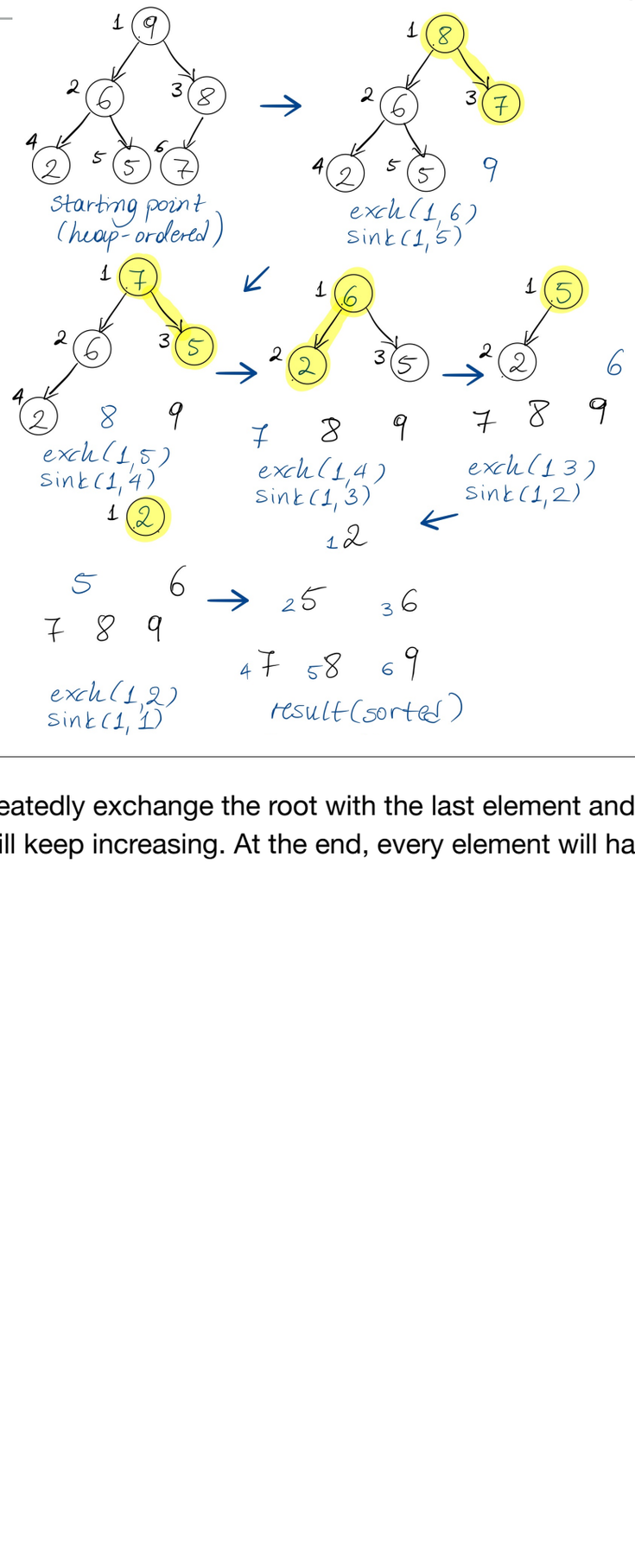
This video captures the entire example that we just saw, by first building the heap, starting at the first internal node and sinking it, and once it has constructed the heap by running sortdown and repeatedly deleting the maximum at the time element so that it eventually sorts the entire array.

Practice Time

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

To practice, let’s run heapsort on the array [2,9,7,6,5,8] by first constructing the heap and then running sortdown.

Answer: Sortdown



Your sortdown will start with the binary heap you just created and will repeatedly exchange the root with the last element and sink it. With every iteration, you will have an increasingly smaller heap and a subarray of finally sorted elements that will keep increasing. At the end, every element will have found its final position.

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.
- In-place sorting algorithm with $O(n \log n)$ worst-case!
- Remember:
  - 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 longer than quick sort.
  - Poor use of cache because it accesses memory in non-sequential manner, jumping around.
    - more in CS105!
  - Not stable.

- Let’s now analyze heapsort as we’ve done with every other sorting algorithm we’ve seen so far.
- The construction of the heap made O(n) exchanges and O(n) comparisons.
- Sortdown took O(nlogn) exchanges and O(nlogn) comparisons.
- Since heapsort consists of heap construction + sortdown the total running time is O(n)+O(nlogn) = O(nlogn) both for the comparisons and exchanges.
- Since it does not require auxiliary space, heap sort is an in-place algorithm which has the additional advantage of having an O(nlogn) guaranteed performance.
- We have already seen in our examples that heap sort is not stable.
- In practice, it is also slower than quick sort and than mergesort.
- Therefore, quick sort is preferred when it comes to speed and merge sort is preferred when it comes to stability.

Sorting: Everything you need to remember about it!

Which Sort	In place	Stable	Best	Average	Worst	Remarks
Selection	X		$O(n^2)$	$O(n^2)$	$O(n^2)$	n exchanges
Insertion	X	X	$O(n)$	$O(n^2)$	$O(n^2)$	Use for small arrays or partially ordered
Merge		X	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; stable
Quick	X		$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$n \log n$ probabilistic guarantee; fastest!
Heap	X		$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Guaranteed performance; in place

Putting everything we have seen together so far about comparison-based sorting algorithms, we end up with this table that you can use when reviewing sorting algorithms. Remember, as with everything we've seen in CS62, you need to understand the requirements of your application to pick the appropriate algorithm.

Lecture 17: Heapsort

- Heapsort

This concludes the lecture on heapsort and the sorting unit.

Readings:

- Recommended Textbook:
  - Chapter 2.4 (Pages 308-327), 2.5 (336-344)
- Website:
  - Priority Queues: https://algs4.cs.princeton.edu/24pq/
- Visualization:
  - Create (compare the n and nlogn approaches) and heapsort: https://visualgo.net/en/heap

Worksheet

- Lecture 17 worksheet

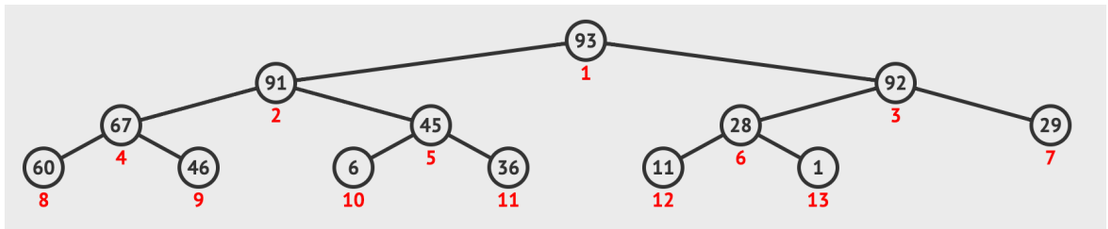
In the meantime, you can read more about today's lecture in the textbook and its website and practice running heap sort on this tool. As always, practice makes perfect so make sure you practice with priority queues and heap sort and/or give the practice problems of the textbook a try.

Practice Problem 1

- 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) algorithm) and visualize it at the end of each deletion.

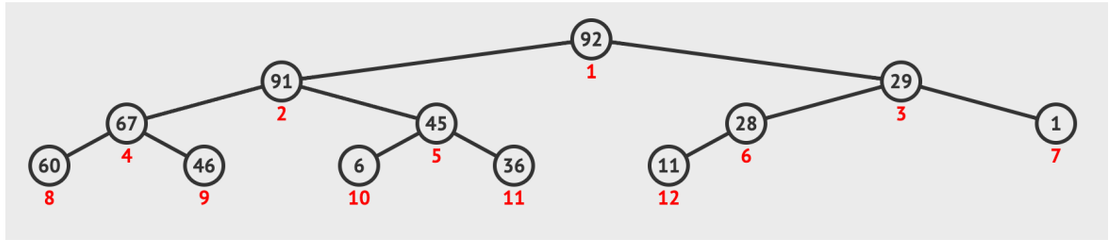
ANSWER 1

- 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 all the steps of the sortdown.
- Heap construction

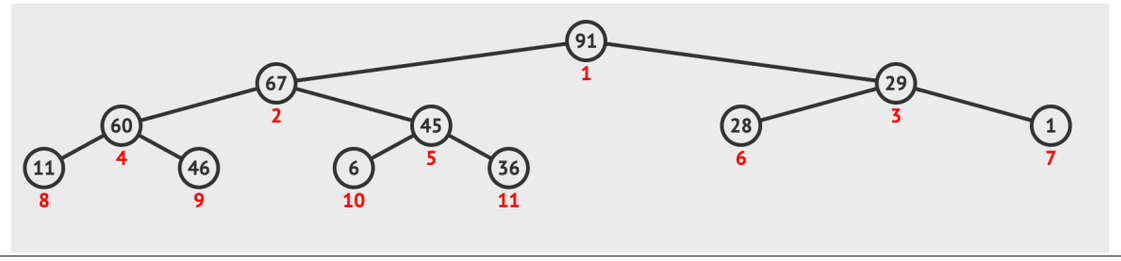


ANSWER 1

Extract max (93)

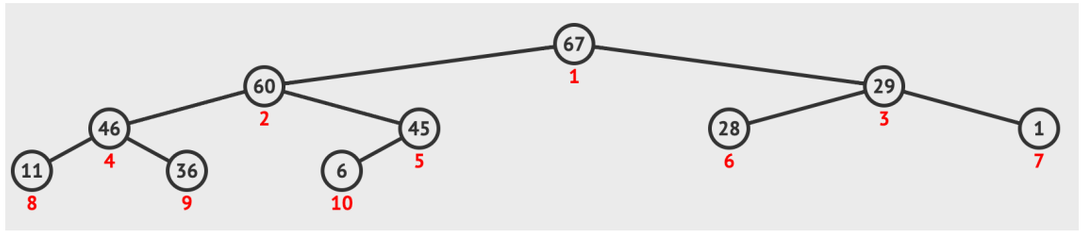


Extract max (92)

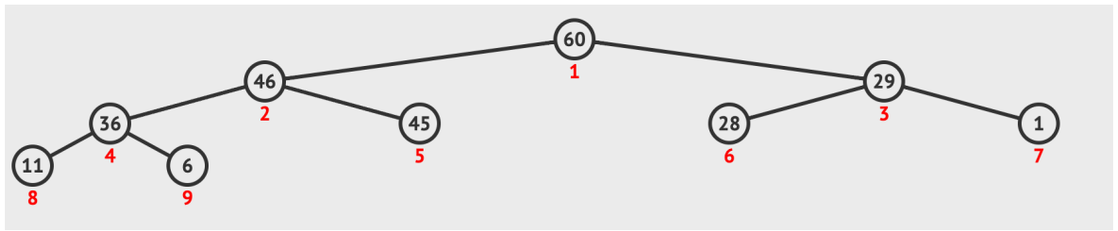


ANSWER 1

- Extract max (91)

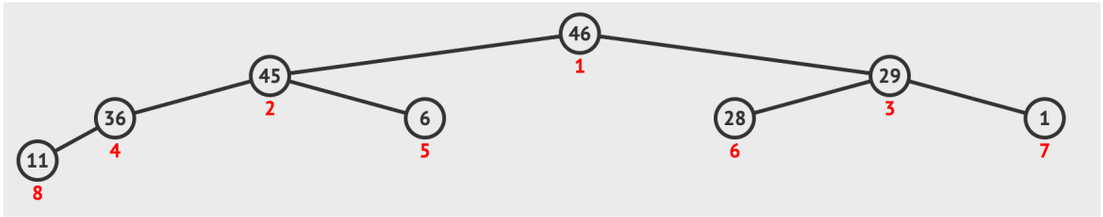


- Extract max (67)

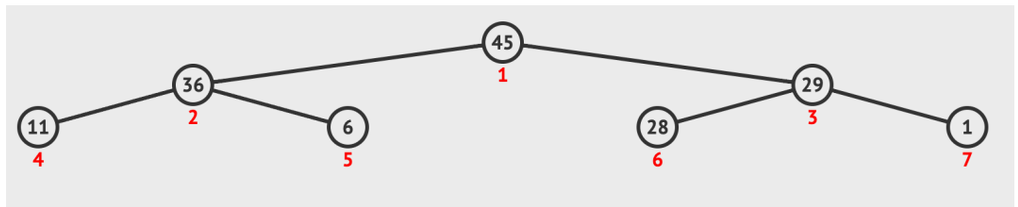


ANSWER 1

- Extract max (60)

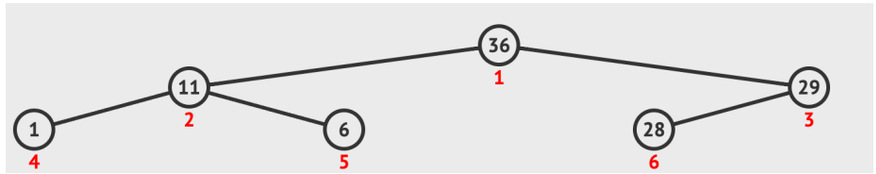


- Extract max (46)

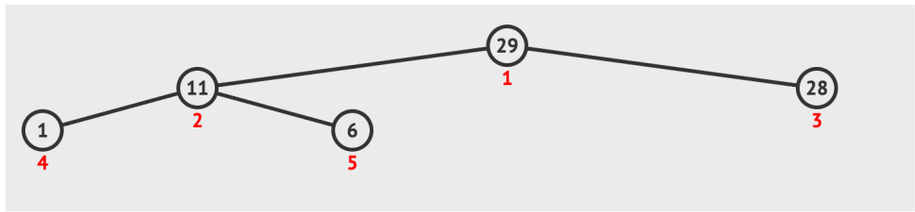


ANSWER 1

- Extract max (45)

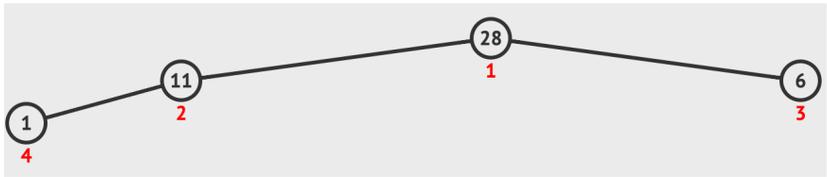


- Extract max (36)

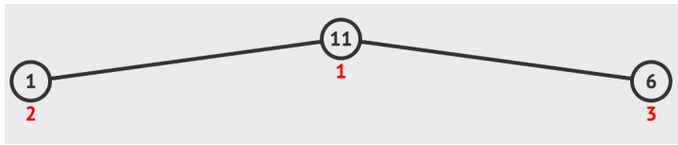


ANSWER 1

- Extract max (29)



- Extract max (28)



ANSWER 1

- Extract max (11)

- Extract max (6)

- Extract max (1)