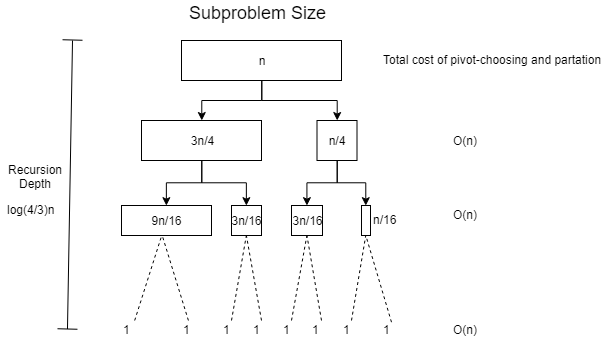
**Question 4**

a) The cost of quicksort can be decomposed into three 4 parts as the following recurrence relation shows

The cost of choosing a pivot is content time O (1). The partition based on the pivot required one comparison between each element and the pivot. Thus, its time complexity is O(n). The recurrence relation can be simplified to

The time complexity of quicksort is related to the depth of recursion.

Given that the oracle algorithm can choose a pivot that is guaranteed to lie in between top 25% and top 75%, the worst time complexity occurs when pivot is either at the 25% or the 75% of the elements. In this case, the problem is divided into two subproblem of size and of the original problem. Thus, the recursion depth is .



The cost of pivot-choosing and partition at each depth is O(n).

Hence, the total cost of quicksort using the oracle algorithm in the worst case is

If we convert log (4/3) to log2, we get

Thus, Quicksort does not perform worse than using the oracle as a pivot selection strategy.

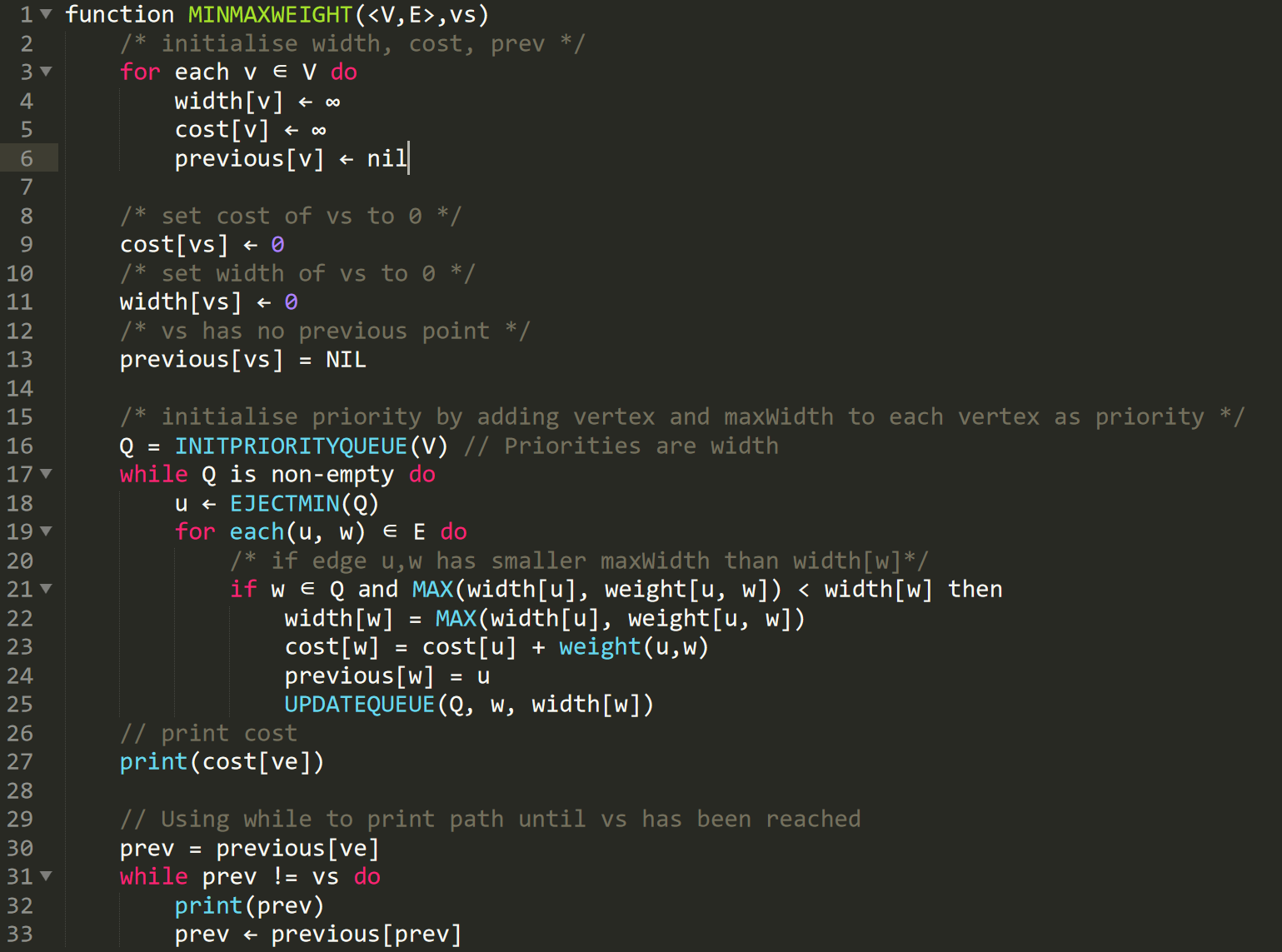
b) The probability that a randomly selected element of the array falls in between the top

25% and top 75% is 0.5.

c)

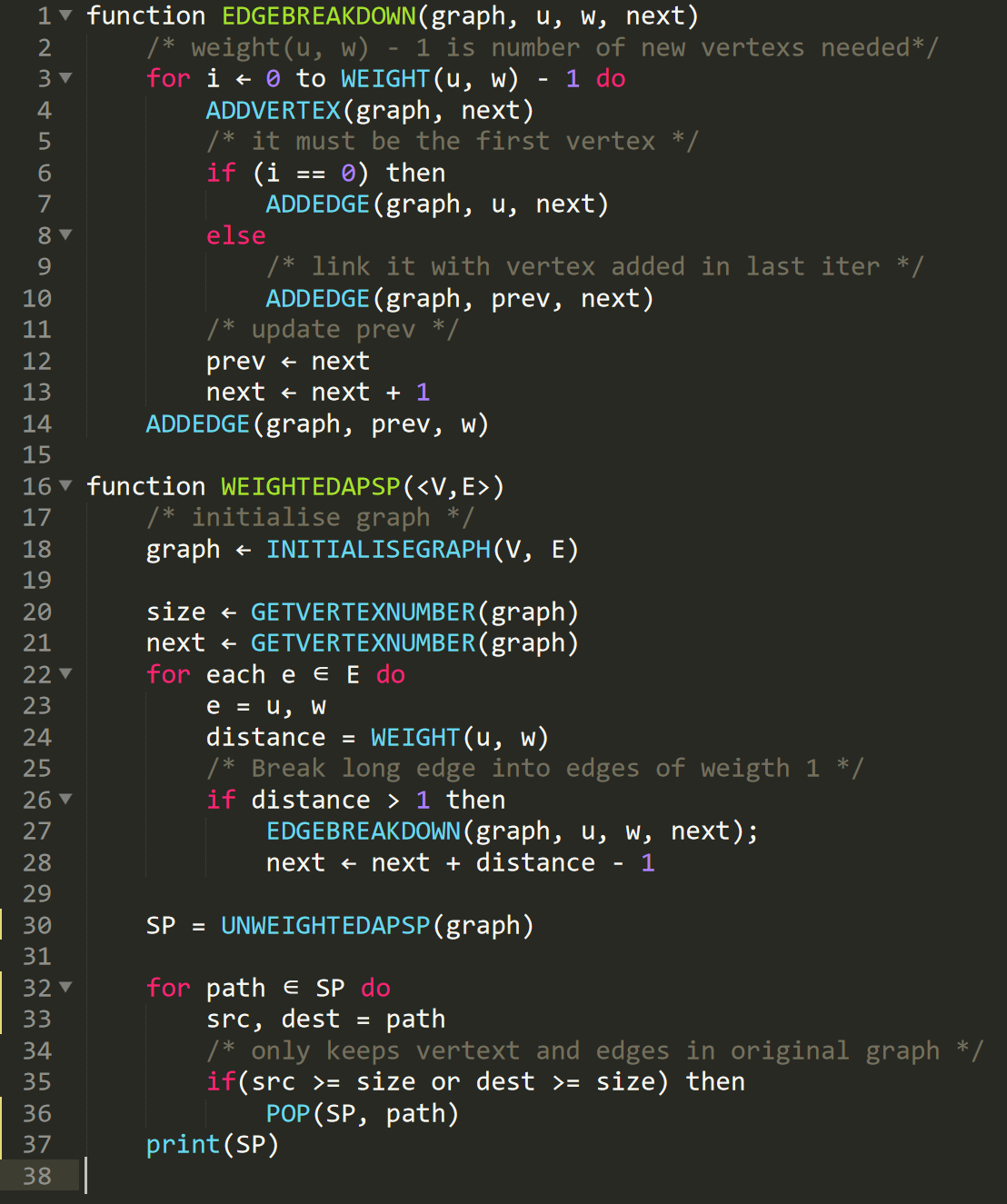
**Question 5**

The following screenshot shows the pseudocode to find MinMaxWeightPath. It adapted the idea of greedy algorithm from Dijkstra’s algorithm. Comparing to Dijkstra’s algorithm to find shortest path, the criteria of selecting the next point to visit is max weight along the path from vs. Its worst-case time complexity with heap as priority queue and Adjacency List is

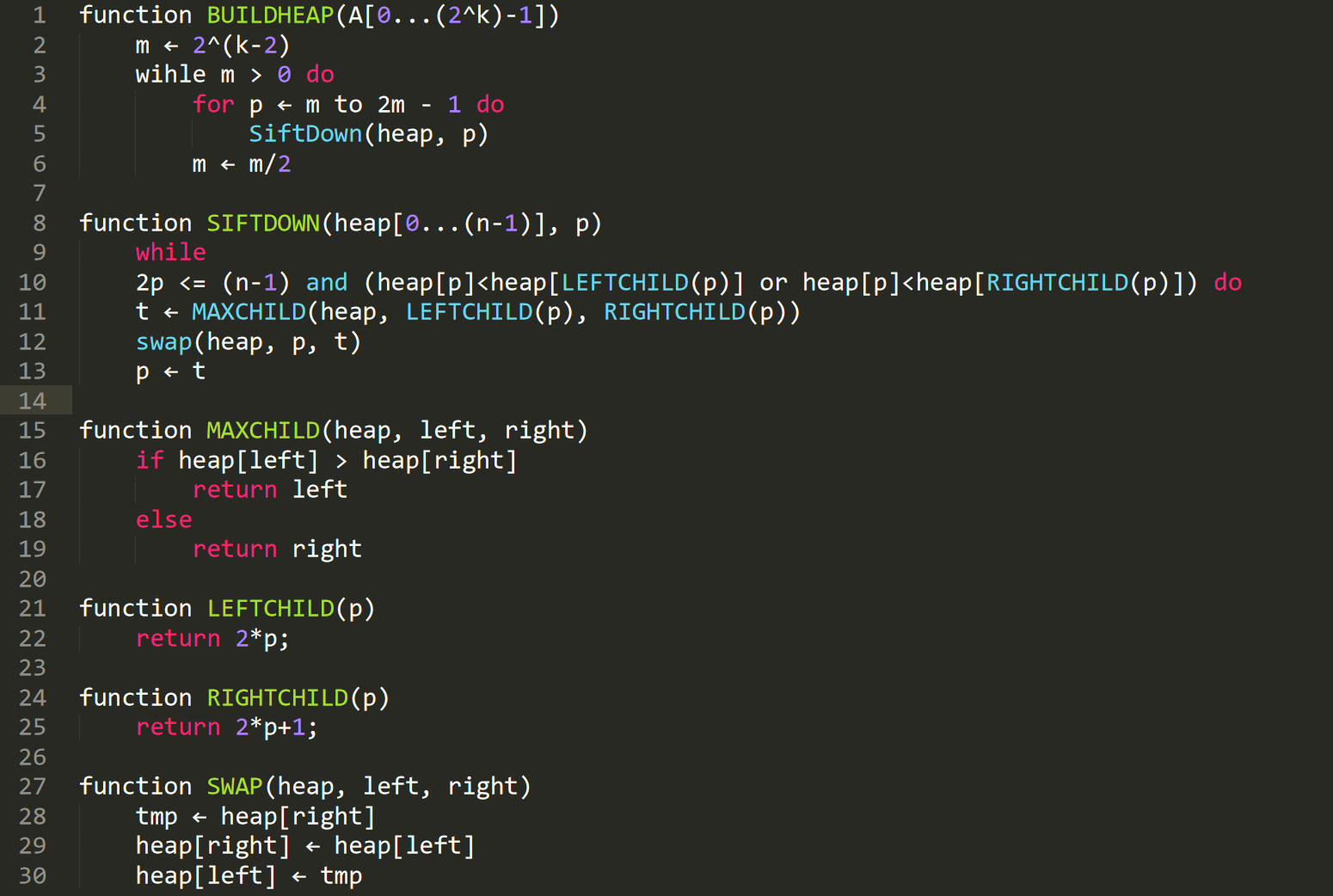
****

**Question 6**

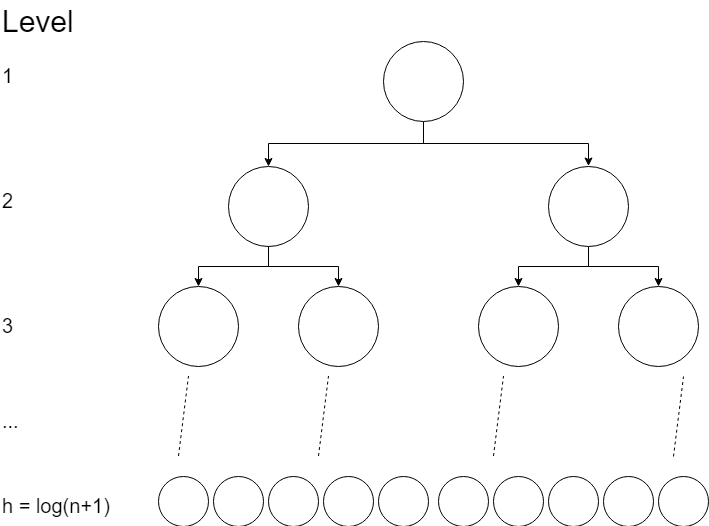
The following screenshot shows the pseudocode to compute all pair shortest path using UnweightedAPSP algorithm. It converts a weighted graph to an unweighted graph by adding additional vertex and edges. Edge in the weighted graph with length k is divided into k edges of length 1 in the unweight graph. Thus, the output of UnweightedAPSP algorithm includes shortest path connecting original vertex and additional vertices. These paths are removed later to print correct paths.

****

**Question 7**

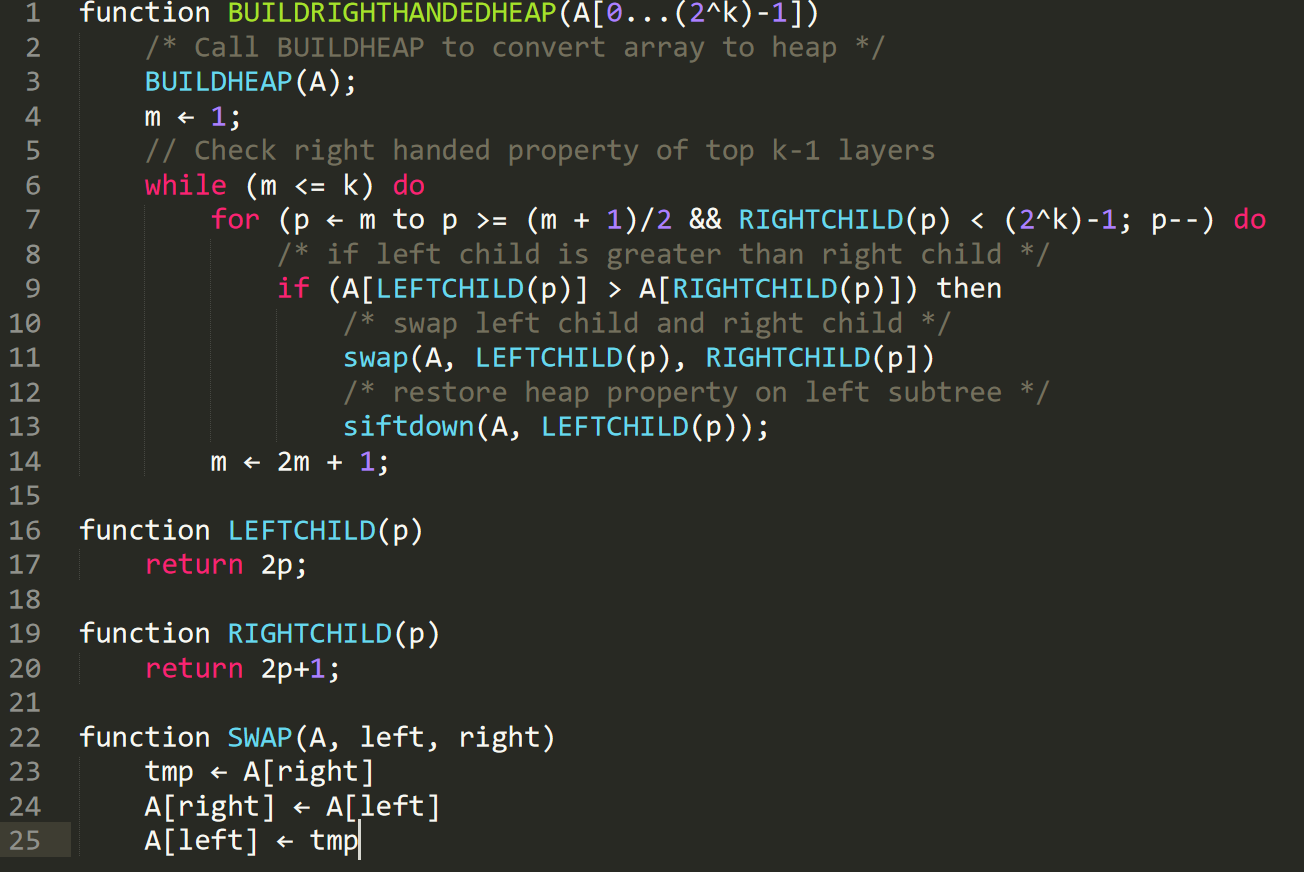


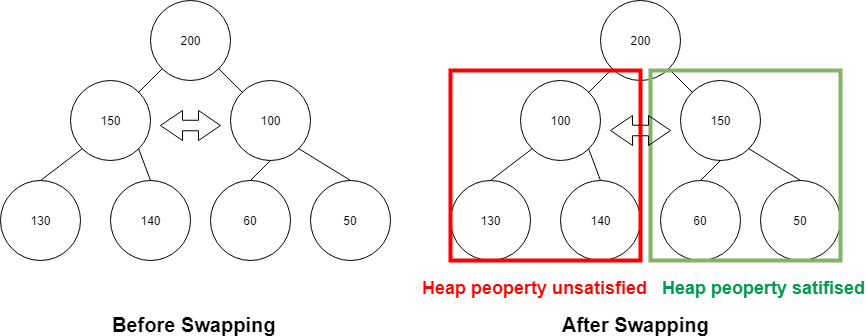
Given the heap is a full binary tree, the worst-case time complexity of building a heap using siftdown is O(n). The buildheap function builds heap bottom-up by traversing each level upward from the level above leaves.

****

In the worst case, nodes at level i need to be swapped for h-i times to store heap property. Thus, the upper-bound complexity is given by

The following screenshot show the pseudocode to build a right handed heap. It first call buildheap to convert A into heap then check if right handed property is satisfied.



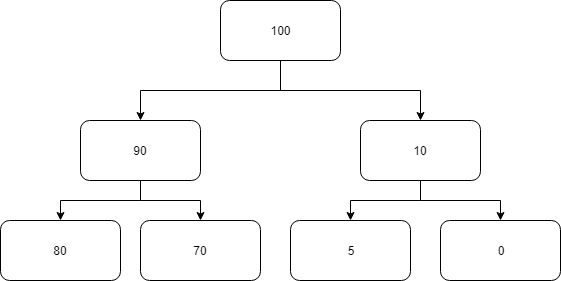
****

As the above illustration shows, after swapping the left child and right child of a node to achieve right-handed property, its left child no longer satisfies heap property. In the worst case, for a node at level i, (h – i) swaps are required to restore heap property. Note that, only half of nodes (left nodes) need to be fixed.

Hence, the total cost of restoring heap property after swapping is given by

**Question 8**

Heap property states that each parent is not smaller than its children. It doesn’t state that each child must be smaller than parent’s “brothers”. In other word, heap property doesn’t guarantee that elements on k layer are greater than all elements on k+1 layer.



As the above heap demonstrates, top-3 elements (100, 90, 80) are not in the first 2 layers.