Heap – 30 points; Suggested work & submission time less than 30min

Submission instructions:
Please submit a single C file (use any file name) on TEACH under FinalExamProblem3 before 11am. We will not accept hand-written solutions.

Implement a C function, `removeThresholdHeap()`, that removes from Heap all elements with **lower priority** than `threshold`, while preserving the heap property, as illustrated in Figure 1.

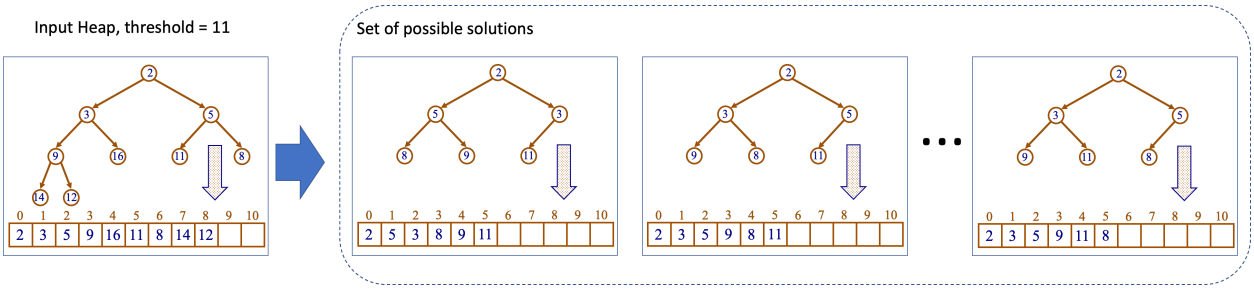

Fig. 1. For a given heap on the left and threshold = 11, the function produces one of possible solutions depicted on the right. Note that all of the depicted solutions are equivalent heaps, so your goal is to produce one of them.

The input arguments include a pointer to Heap implemented as a dynamic array, and the threshold. The function has a **strict memory constraint** that absolutely no new data structures can be formed and no new memory locations can be allocated, except for a couple of local integer variables. Also, time complexity of the function should not exceed $O(n \log n)$, where n is the number of elements in Heap.

In the following, you are given the definitions of `struct` data types for elements of Heap, and the function for percolating down a node of Heap. You should not modify this function and `struct` data types, and you are allowed to add your own functions as you find appropriate. If your solution calls other functions, you will have to implement them for this problem. There is no need to implement the main function for this code.

```
struct Element{
   int priority; /* the lower value the higher priority */
   char string[100];
};

#define TYPE struct Element

struct DynArr{
   TYPE *data;  /* pointer to the data array */
   int size;  /* number of elements in the array */
   int capacity;  /* capacity of the array */
};
```

```
/* Percolate down a node of Heap
   Input: heap -- pointer to the heap
          index -- index of a node for percolating down
*/
void percolateDownHeap(struct DynArr *heap, int index) {
    assert(heap && index < heap->size);
    int higher;
    TYPE tmp;
    int maxIdx = heap->size;
    int leftIdx = index * 2 + 1; /* left child index */
    int rghtIdx = index * 2 + 2; /* right child index */
    while (leftIdx < heap->size){
        if (rghtIdx < maxIdx)
            higher = heap->data[leftIdx].priority < heap->data[rghtIdx].priority ?
                     leftIdx : rghtIdx;
        else
            higher = leftIdx;
        if(heap->data[higher].priority < heap->data[index].priority){
            tmp = heap->data[higher];
            heap->data[higher] = heap->data[index];
            heap->data[index] = tmp;
            index = higher;
            leftIdx = index * 2 + 1;
            rghtIdx = index * 2 + 2;
        }
        else
            leftIdx = heap->size;
    }
}


/* Remove from Heap elements with lower priority than a threshold
    - Input: heap = pointer to a heap implemented as a dynamic array.
             threshold =  positive integer.
    - Pre-conditions: heap was initialized well and exists in memory, but may be empty.
      Also, heap correctly established the priority relationships among all of its elements.
    - Constraints: time complexity <= O(n log n); no new data structures allowed,
                   no new memory locations can be allocated, except for a couple of integers
*/
void removeThresholdHeap(struct DynArr *heap, int threshold){
   assert(heap && threshold > 0);

   /* FIX ME */

}
```