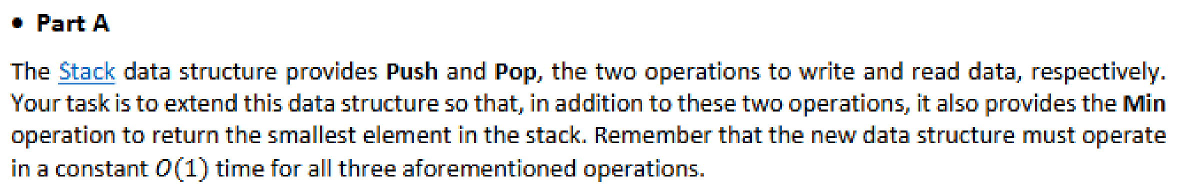
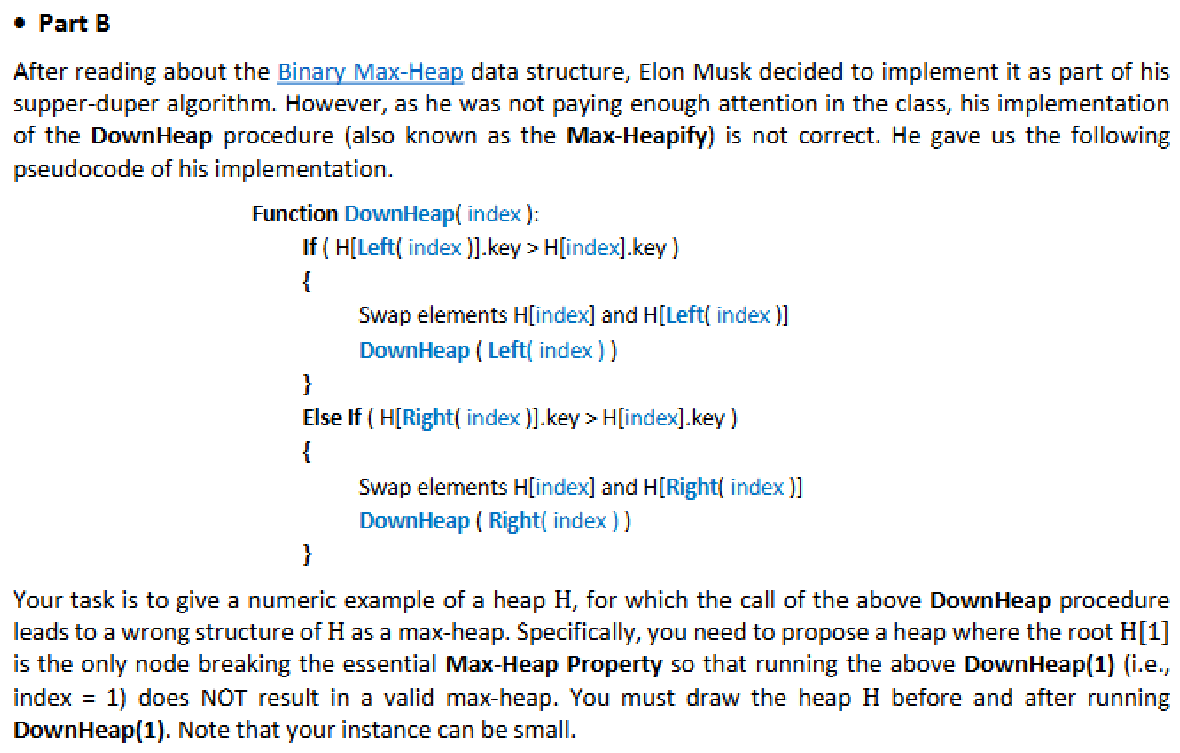


- **Part A**

The Stack data structure provides **Push** and **Pop**, the two operations to write and read data, respectively. Your task is to extend this data structure so that, in addition to these two operations, it also provides the **Min** operation to return the smallest element in the stack. Remember that the new data structure must operate in a constant $O(1)$ time for all three aforementioned operations.

A： The main stack stores all elements, and the auxiliary stack stores the current minimum value of each step. When pushing an element, if the new element is ≤ the top of the auxiliary stack, it is also pushed into the auxiliary stack; when popping an element, if the element popped is equal to the top of the auxiliary stack, the auxiliary stack is popped out simultaneously. In this way, when taking the minimum value, you only need to return to the top of the auxiliary stack, and the operations of pushing, popping, and taking the minimum value can all be completed in O(1) time.



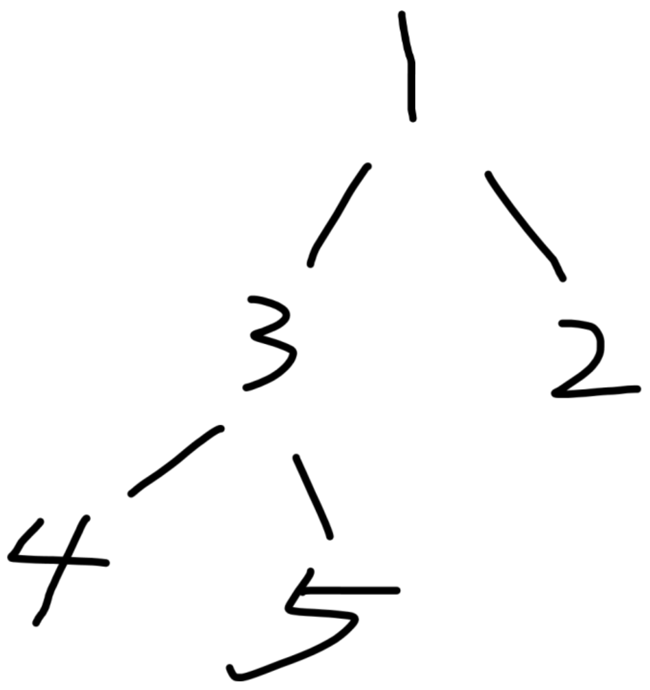
- **Part B**

After reading about the Binary Max-Heap data structure, Elon Musk decided to implement it as part of his supper-duper algorithm. However, as he was not paying enough attention in the class, his implementation of the **DownHeap** procedure (also known as the **Max-Heapify**) is not correct. He gave us the following pseudocode of his implementation.

```
Function DownHeap( index ):
    If ( H[Left( index )].key > H[index].key )
    {
        Swap elements H[index] and H[Left( index )]
        DownHeap ( Left( index ) )
    }
    Else If ( H[Right( index )].key > H[index].key )
    {
        Swap elements H[index] and H[Right( index )]
        DownHeap ( Right( index ) )
    }
```

Your task is to give a numeric example of a heap H, for which the call of the above **DownHeap** procedure leads to a wrong structure of H as a max-heap. Specifically, you need to propose a heap where the root $H[1]$ is the only node breaking the essential **Max-Heap Property** so that running the above **DownHeap(1)** (i.e., index = 1) does NOT result in a valid max-heap. You must draw the heap H before and after running **DownHeap(1)**. Note that your instance can be small.

A： Let heap where the root (index 1) violates the max-heap property (before downheap):

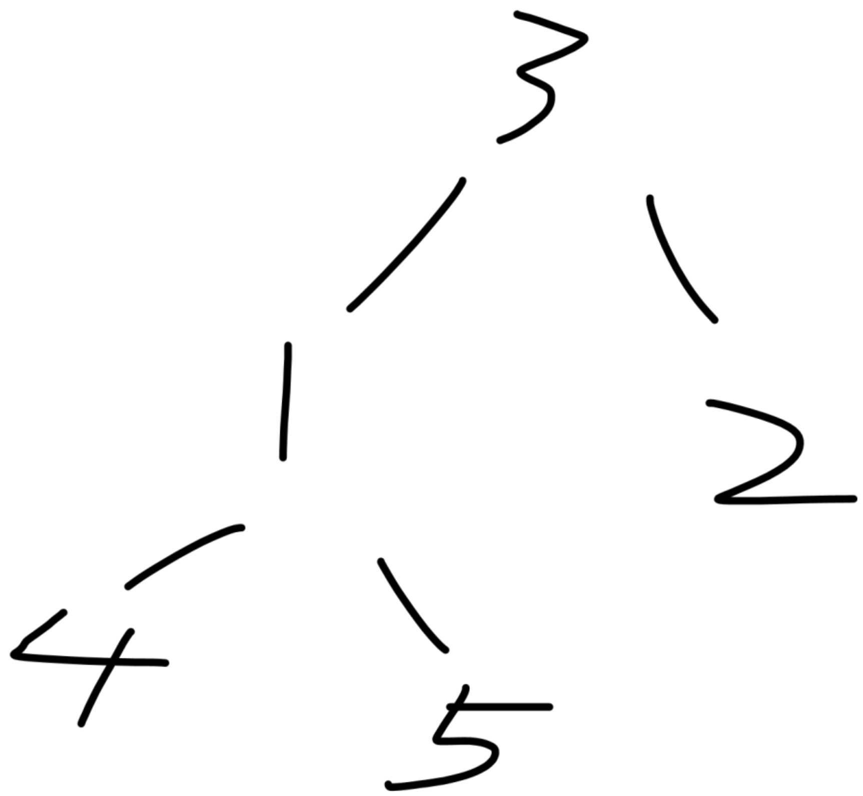


The array will be : [1,3,2,4,5]

Compare left child (3) with root (1). Since 3>1, swap them.

Recursively call downheap on the left child (now index 2).

The new structure becomes:



But the Elon's code does not check this subtree again.

If with the downheap(1):

The subtree rooted at index 2 still violates the max-heap property. The correct DownHeap should recursively compare and swap with the largest child, but Elon's code terminates prematurely.

- **Part C**

Assume now that Elon Musk has eventually fixed the problem above and developed a correct implementation of the Max-Heap that provides all the standard operations such as:

- **DeleteMax, Insert(key), IncreaseKey(index, key)** in a logarithmic $O(\log n)$ time, and
- **Max** in a constant $O(1)$ time,

where n is the total number of stored elements. Elon Musk wants to add a new capability to his max-heap. Specifically, he would like to implement a **DeleteElement(index)** operation that removes element **H**[index] from the max-heap H consisting of n elements in a logarithmic $O(\log n)$ time. Note that the **H**[index] does

NOT need to be the maximum-key element in H. Also, naturally, H must maintain the Max-Heap Property after deletion of **H**[index].

Your task is to help Elon Musk by providing a pseudocode of such a **DeleteElement(index)** operation accompanied with a brief analysis of its running time as a function of n elements.

A: Firstly, we can have a out of bounds check method, make sure the target index is in the valid range. For example : 1 < index < H_size.

Then exchange the target element with the last element of the heap. Delete the last element. For example: H_size – 1 => H_size ; index / 2 => parent .

Lastly, If the new element is larger or smaller than the parent node, recursively exchange it with the parent node until the heap order is restored.

For example : if index > 1 > parent ; shiftup (H,index)

Else: maxheap(H,index)