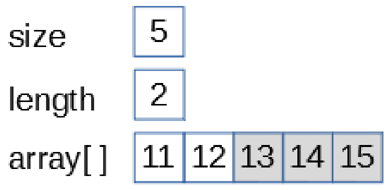
WORKSHEET WEEK 05: STACKS, QUEUES, PRIORITY QUEUES

5.1 Problems

Problem 1: (Stack Implementation as Array): Stack is implemented as an array. In our case the array has size $n = 5$. Stack contains integer numbers; initially the array has the following content.



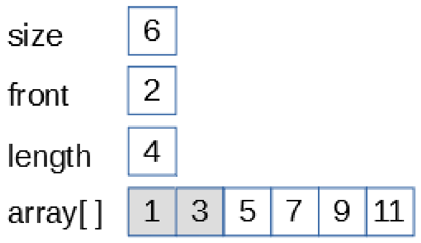
Stack has the physical representation with `length` $= 2$ (the number of elements in the stack), `size` $= 5$ (maximal number of elements contained in the stack). We have the following fragment:

```
pop();
push(21);
push(22);
pop();
push(23);
push(24);
pop();
push(25);
```

Draw the state of the array after every command. (Every `push(elt)` command assigns a new element into the element `array[length]`, then increments `length` by 1. The command `pop()` does not modify the array, but decreases `length` by 1.

If the command cannot be executed (`pop()` on an empty stack; `push(elt)` on a full stack), then the stack structure does not change at all (`array` or `length` are not modified). To help imagine the state of this stack, you can shade those cells that do not belong to the array.

Problem 2 (Queue Implementation as a Circular Array): A queue is implemented as an array with `size` elements; it has two extra variables `front` (pointer to the first element) and `length` (the current number of elements in the queue). Current state is shown in the figure:



Enumeration of array elements starts with 0. The array is filled in a circular fashion. The command `enqueue(elt)` inserts a new element at

$$(\texttt{front} + \texttt{length}) \bmod \texttt{size},$$

where "mod" means the remainder when dividing by `size`. It also increments the `length` element.

The command `dequeue()` does not change anything in the array, but increments `front` by 1 and decreases $length$ by 1. Thus the queue becomes shorter by 1.

```
dequeue();
enqueue(21);
dequeue();
enqueue(22);
enqueue(23);
enqueue(24);
dequeue();
```

Show the state of the array after every command – `array, length, front` variables after every line. (Shade the unused cells.)

Problem 3: Denote a, b, c to be the last 3 digits of your Student ID, and compute the following numbers:

- $F = ((a + b + c) \mod 3) + 2$
- $\texttt{x1} = (a + b + c) \mod 10$
- $\texttt{x2} = ((a + b) \cdot 2) \mod 10$
- $\texttt{x3} = ((b + c) \cdot 3) \mod 10$
- $\texttt{x4} = ((c + a) \cdot 7) \mod 10$

The queue Q is implemented as an array of size $N = 6$; its elements have indices from $\{0, 1, 2, 3, 4, 5\}$.

Initially the queue parameters are these:

- $\texttt{Q.front} = \texttt{F}$,
- $\texttt{Q.length} = 4$,
- $\texttt{Q.size} = 6$,

And the content of the array is the following:

i	0	1	2	3	4	5
array[i]	1	3	5	7	9	11

Somebody runs the following code on this queue:

```
Q.enqueue(x1)
Q.enqueue(x2)
Q.dequeue()
Q.dequeue()
// show the state of Q
Q.enqueue(x3)
Q.enqueue(x4)
Q.dequeue()
// show the state of Q
```

After Line 4 (and at the very end) show the current state of the queue `Q`. The state should display the content of the array and also the values of `Q.front` and `Q.length`.

You can use shading, if it helps to visualize the array cells that are not currently used by your queue.

Note: Painting something gray is not required (since front/length indicate the state of your queue anyway). But painting cells gray may be helpful, if you want to visualize where your queue has the useful values (and what is some old garbage – you can shade it over).

Problem 4:

(A) Assume that heap is implemented as a 0-based array (the root element is `H[0]`), and the heap supports DELETEMIN(H) operation that removes the minimum element (and returns the heap into consistent state).

Find, if the heap property holds in the following array:

$$H[0] = 6, 17, 25, 20, 15, 26, 30, 22, 33, 31, 20.$$

If it is not satisfied, find, which two keys you could swap in this array so that the heap property is satisfied again. Write the correct sequence of array H.

Note: A *consistent state* in a minimum heap means that the key in parent does not exceed keys in left and right child.

(B) Assume that heap is implemented as a 0-based array (the root element is `H[0]`), and the heap supports DELETEMAX(H) operation that removes the maximum element.

If the heap does not satisfy invariant (in a consistent max-heap, every parent should always be at least as big as both children), then show how to swap two nodes to make it correct.

$$96, 67, 94, 10, 67, 68, 69, 9, 10, 11, 50, 67.$$

Problem 5 (Insert into a min-heap): Show what is the final state of a heap after you insert number 6 into the following minimum-heap (represented as a zero-based array):

$$9, 18, 28, 23, 20, 29, 33, 25, 36, 34, 23.$$

Problem 6 (Delete maximum from a Max-Heap): Show what is the final state of a heap after you remove the maximum from the following heap (represented as a zero-based array):

$$96, 67, 94, 10, 67, 68, 69, 9, 10, 11, 50, 67.$$

Problem 7 (Removing from Maximum Heap): Here is an array for a Max-Heap:

16	14	10	8	7	9	3	2	4	1

The image shows array used to store Maximum Heap (a data structure allowing inserts and removal of the maximum element). The array starts with the 0-th element (and any parent node in such tree should always be at least as big as any of its children).

(A) Draw the initial heap based on this array. Heap should be drawn as a complete binary tree.

(B) Run the command $\text{DELETEMAX}(H)$ on this initial heap. Draw the resulting binary tree (after the heap invariant is restored – any parent node is at least as big as its children). Draw the binary tree image you get.

(C) On the tree that you got in the previous step (B) run the command $\text{INSERT}(H, x)$, where $x = a + b + c$ is the sum of the last three digits of your student ID. Draw the binary tree image you get.

(D) Show the array for the binary tree you got in the previous step (C) (i.e. right after the $\text{DELETEMAX}(H)$ and $\text{INSERT}(H, x)$ commands have been executed).

Problem 8: A *multiset* (or a *bag*) is any collection of items similar to a *set*, which can contain multiple copies of the same item. For example, $X = \{2, 2, 2, 3, 3, 5\}$ is a multiset. The 2-quantile (also known as the *median*) for a multiset $X = \{x_1, x_2, \ldots, x_k\}$ is the number m satisfying the following probability inequalities (where x_i is a randomly picked element from the multiset X – each element is selected with the same probability):

$$P(x_i < m) \leq \frac{1}{2}, \quad \text{and} \quad P(x_i \leq m) \geq \frac{1}{2}.$$

Suggest an efficient data structure to store "median-maintained multisets" X containing integer numbers that support the following operations:

Insert: Add a new element x to the multiset X.

ExtractMedian: Remove and return the median from the multiset X, if it belongs to X.

Median: Return the median from the multiset without removing it.

Size: Return the number of elements in the multiset.

Note: In Python you can compute this flavor of median like this:

```
import pandas as pd
    a = pd.Series([1,2,3,4])
    a.quantile(0.5, 'lower')  # should return 2
    # statistics.median([1,2,3,4]) would be 2.5
```

Problem 9: Run the Huffman algorithm to build prefix codes for a five-letter alphabet $\{A, B, C, D, E, F\}$, where the respective frequencies are $\{45, 13, 12, 16, 9, 5\}$.