Priority Queues- 2

Heap Sort

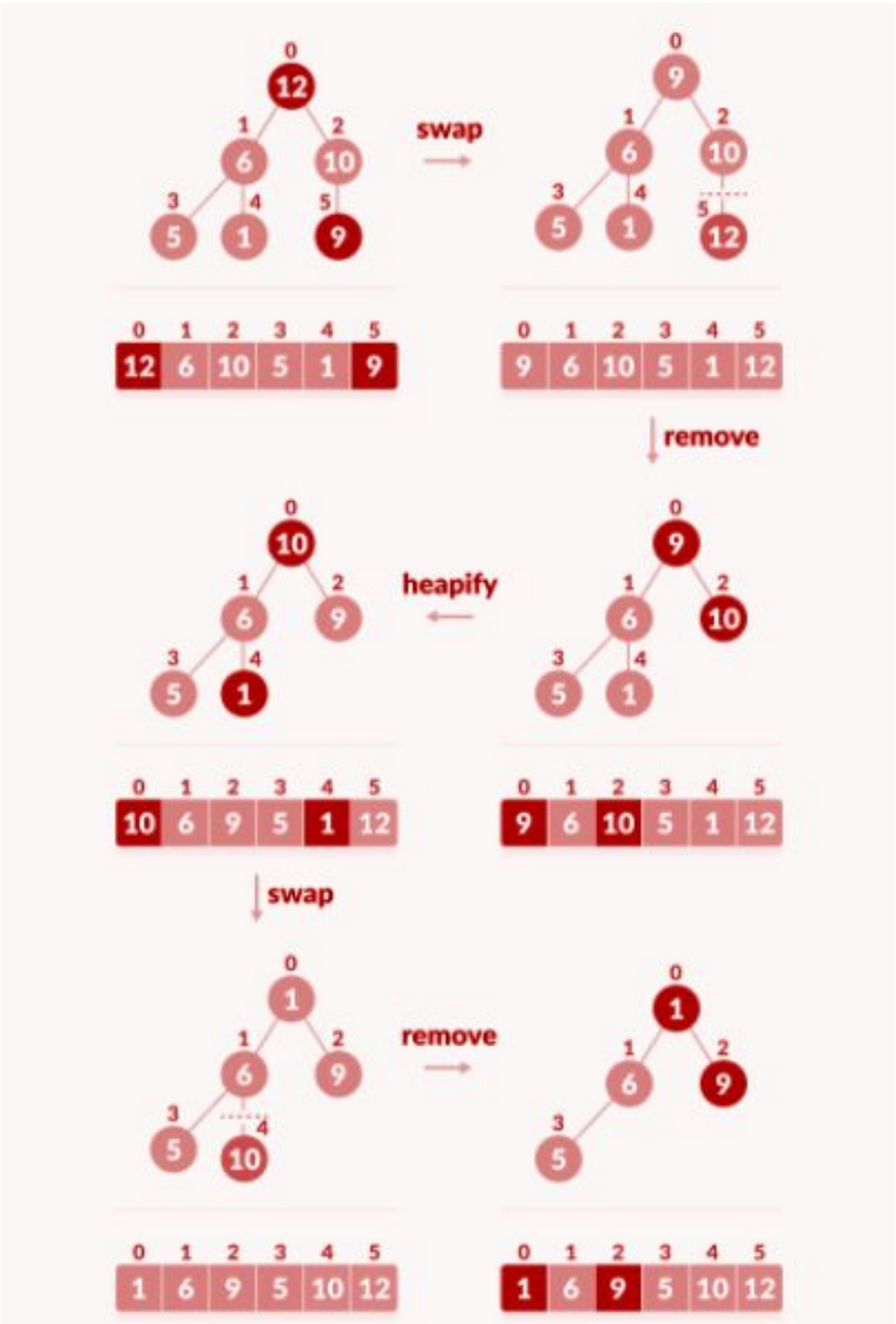
- Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a great worst-case runtime of **O(nlog(n))** regardless of the input data.
- As the name suggests, Heap Sort relies heavily on the heap data structure - a common implementation of a Priority Queue.
- Without a doubt, Heap Sort is one of the simplest sorting algorithms to implement, and coupled with the fact that it's a fairly efficient algorithm compared to other simple implementations, it's a common one to encounter.

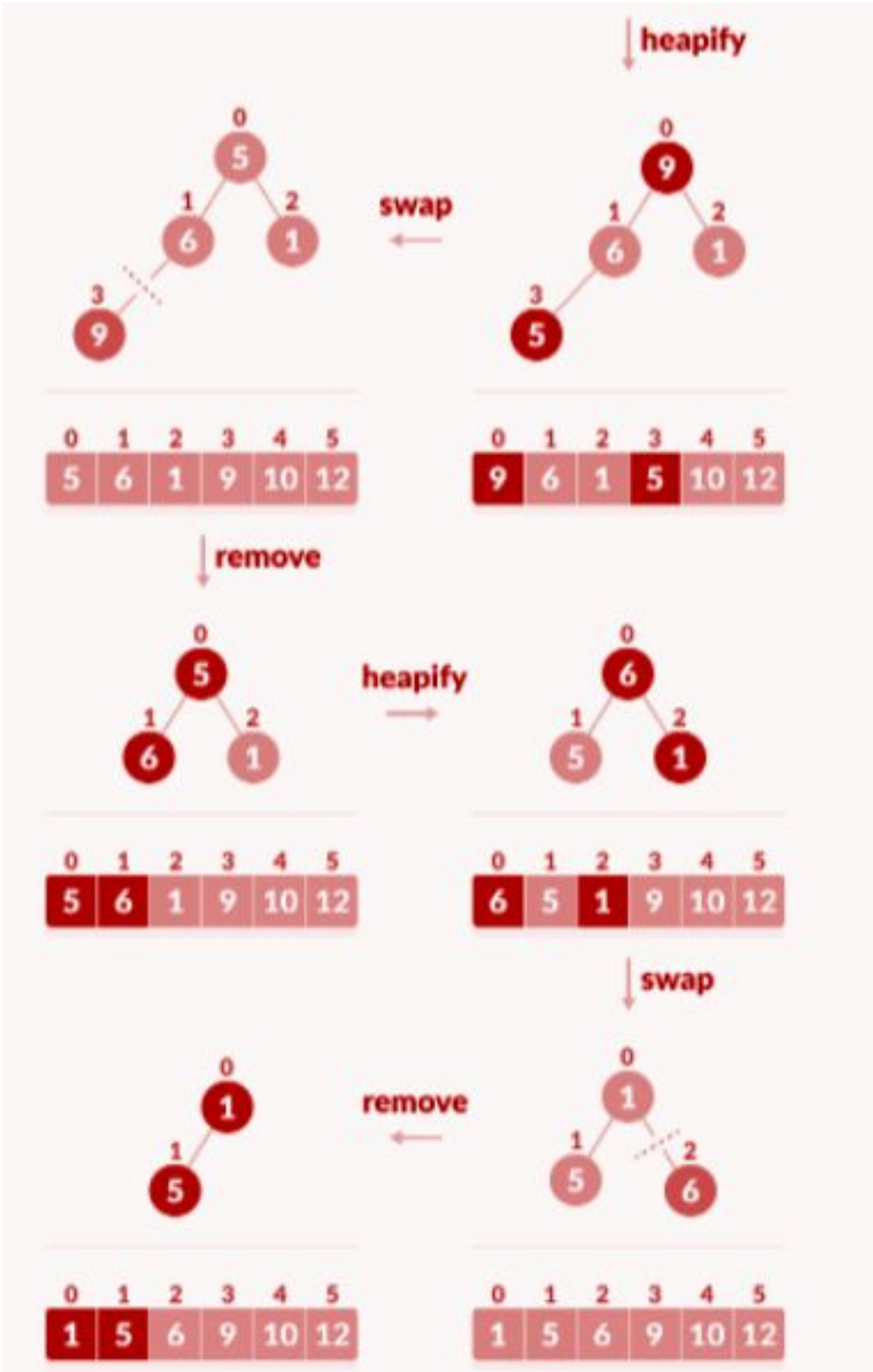
Algorithm

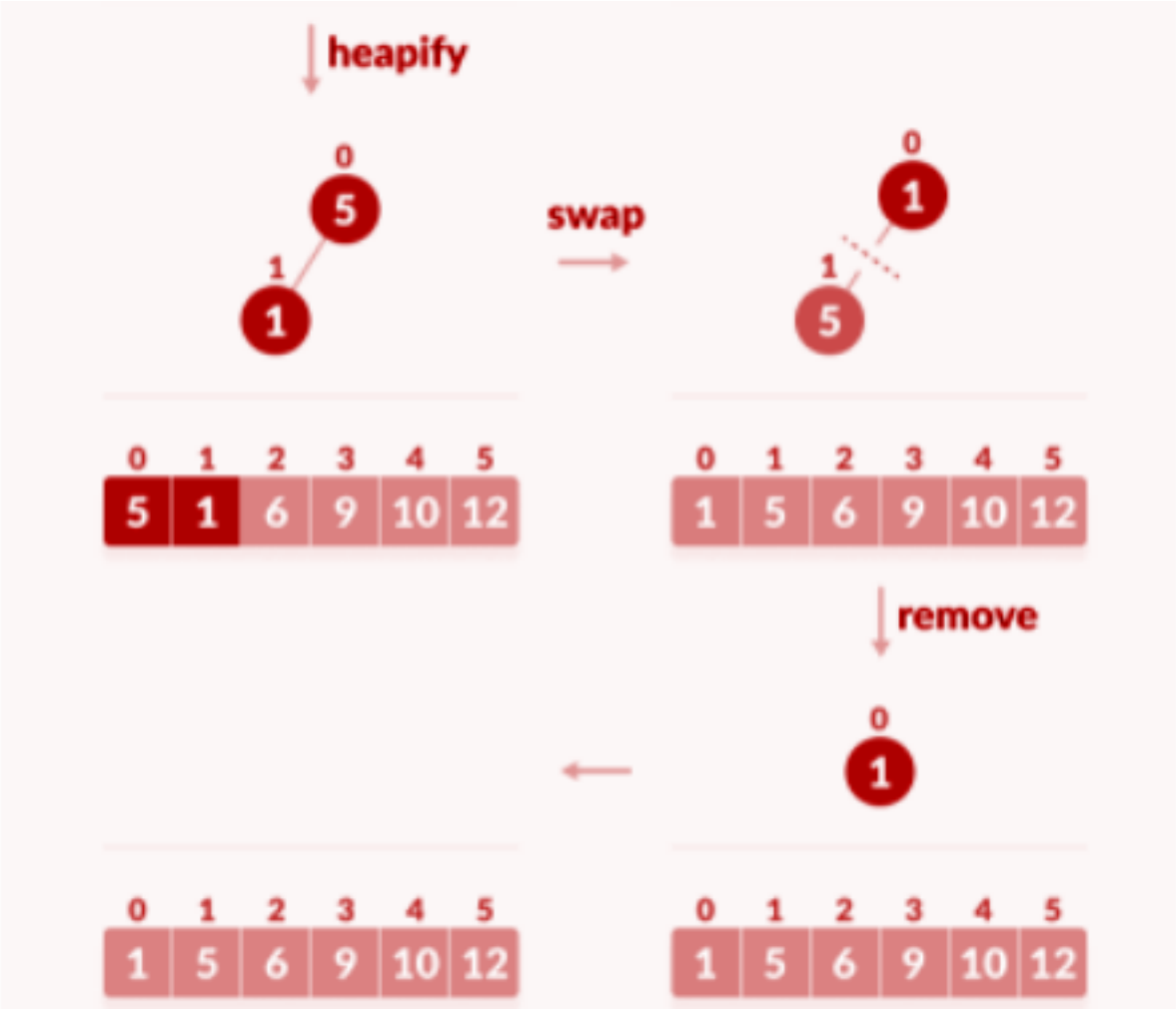
- The heap-sort algorithm inserts all elements (from an unsorted array) into a maxheap.
- Note that heap sort can be done **in-place** with the array to be sorted.
- Since the tree satisfies the Max-Heap property, then the largest item is stored at the root node.
- **Swap**: Remove the root element and put at the end of the array (nth position)
- Put the last item of the tree (heap) at the vacant place.
- **Remove**: Reduce the size of the heap by 1.
- **Heapify:** Heapify the root element again so that we have the highest element at root.
- The process is repeated until all the items in the list are sorted.

Consider the given illustrated example:

-> Applying heapsort to the unsorted array **[12, 6, 10, 5, 1, 9]**


swap
remove
heapify
swap
remove


heapify
swap
remove
heapify
swap
remove



Go through the given **Python Code** for better understanding:

```
def heapSort(arr):
    n = len(arr)
    # Build a maxheap. last parent will be at ((n//2)-1)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract the max elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)
```

In-built Min-Heap in Python

A heap is created by using python's inbuilt library named **heapq**. This library has the relevant functions to carry out various operations on a **min-heap** data structure. Below is a list of these functions.

- **heapify** - This function converts a regular list to a heap. In the resulting heap, the smallest element gets pushed to index position 0. But the rest of the data elements are not necessarily sorted.
- **heappush** – This function adds an element to the heap without altering the current heap.
- **heappop** - This function returns the smallest data element from the heap.
- **heapreplace** – This function replaces the smallest data element with a new value supplied in the function.

Creating a Min-Heap

A heap is created by simply using a list of elements with the **heapify** function. In the below example we supply a list of elements and the heapify function rearranges the elements bringing the smallest element to the first position.

```
import heapq
H = [21,1,45,78,3,5]
# Use heapify to rearrange the elements
heapq.heapify(H)
print(H)
```

When the above code is executed, it produces the following result −

```
[1, 3, 5, 78, 21, 45]
```

Inserting into heap

Inserting a data element to a heap always adds the element at the last index. But you can apply the heapify function again to bring the newly added element to the

first index only if it is the smallest in value. In the below example we insert the number 8.

```
import heapq
H = [21,1,45,78,3,5]
# Convert to a heap
heapq.heapify(H)
print(H)
# Add element
heapq.heappush(H,8)
print(H)
```

When the above code is executed, it produces the following result −

```
[1, 3, 5, 78, 21, 45]
[1, 3, 5, 78, 21, 45, 8]
```

Removing from heap

You can remove the element at the first index by using this function. In the below example the function will always remove the element at the index position 1.

```
import heapq
H = [21,1,45,78,3,5]
# Create the heap
heapq.heapify(H)
print(H)
# Remove element from the heap
heapq.heappop(H)

print(H)
```

When the above code is executed, it produces the following result −

```
[1, 3, 5, 78, 21, 45]
[3, 21, 5, 78, 45]
```

Replacing in a Heap

The **heapreplace** function always removes the smallest element of the heap and inserts the new incoming element at some place not fixed by any order.

```
import heapq
H = [21,1,45,78,3,5]
# Create the heap
heapq.heapify(H)
print(H)
# Replace an element
heapq.heapreplace(H,6)
print(H)
```

```
[1, 3, 5, 78, 21, 45]
[3, 6, 5, 78, 21, 45]
```

In-built Max-Heap in Python

To implement a max-heap, the **heapq** library has the following functions:

- **_heapify_max** - This function converts a regular list to a max-heap.
- **_heappop_max** - This function returns the largest data element from the heap.
- **_heapreplace_max** – This function replaces the maximum data element with a new value supplied in the function.
- **_siftdown_max-** Pushes a new element, but compares with all its parents, and pushes all the parents down until it finds a place where the new item fits.

K-Smallest Elements in a List

This is a good example of problem-solving via a heap data structure. The basic idea here is to create a min-heap of all n elements and then extract the minimum element K times (We know that the root element in a min-heap is the smallest element).

Approach

- Build a min-heap of size **n** of all elements.
- Extract the minimum elements **K** times, i.e. delete the root and perform heapify operation **K** times.
- Store all these K smallest elements.

Note: The code written using these insights can be found in the solution tab of the problem itself.