Priority Queues- 1

Introduction

- Priority Queues are abstract data structures where each data/value in the queue has a certain priority.
- A priority queue is a special type of queue in which each element is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.
- **For example**, the element with the highest value is considered as the highest priority element. However, in some cases, we may assume the element with the lowest value to be the highest priority element. In other cases, we can set priorities according to our needs.

Difference between Priority Queue and Normal Queue

In a queue, the **First-In-First-Out(FIFO)** rule is implemented whereas, in a priority queue, the values are removed based on priority. The element with the highest priority is removed first.

Main Priority Queues Operations

- **Insert (key, data)**: Inserts data with a key to the priority queue. Elements are ordered based on key.
- **DeleteMin/DeleteMax**: Remove and return the element with the smallest/largest key.
- **GetMinimum/GetMaximum**: Return the element with the smallest/largest key without deleting it.

Auxiliary Priority Queues Operations

- **kth - Smallest/kth – Largest**: Returns the kth -Smallest/kth –Largest key in the priority queue.
- **Size**: Returns the number of elements in the priority queue.
- **Heap Sort**: Sorts the elements in the priority queue based on priority (key).

Priority Queue Applications

Priority queues have many applications - a few of them are listed below:

- **Data compression:** Huffman Coding algorithm
- **Shortest path algorithms**: Dijkstra's algorithm
- **Minimum spanning tree algorithms**: Prim's algorithm
- **Event-driven simulation**: Customers in a line
- **Selection problem:** Finding the kth- smallest element

Priority Queue Implementations

Before discussing the actual implementation, let us enumerate the possible options.

Unordered Array Implementation

- Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are performed by searching the key and then deleting.
- Insertions complexity: **O(1)**.
- DeleteMin complexity: **O(n)**

Unordered List Implementation

- It is very similar to array implementation, but instead of using arrays, linked lists are used.
- Insertions complexity: **O(1)**.
- DeleteMin complexity: **O(n)**.

Ordered Array Implementation

- Elements are inserted into the array in sorted order based on the key field. Deletions are performed at only one end.
- Insertions complexity: **O(n)**; DeleteMin complexity: **O(1)**.

Ordered List Implementation

- Elements are inserted into the list in sorted order based on the key field. Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.
- Insertions complexity: **O(n)**; DeleteMin complexity: **O(1)**.

Binary Search Trees Implementation

- Both insertions and deletions take **O(log(n))** on average if insertions are random (refer to Trees chapter).

Balanced Binary Search Trees Implementation

- Both insertions and deletion take **O(log(n))** in the worst case (refer to Trees chapter).

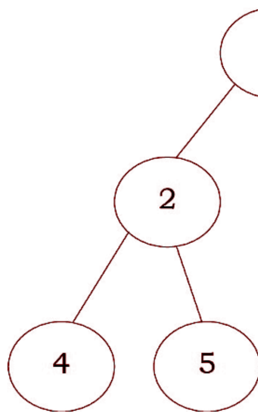
Binary Heap Implementation

In subsequent sections, we will discuss this in full detail.

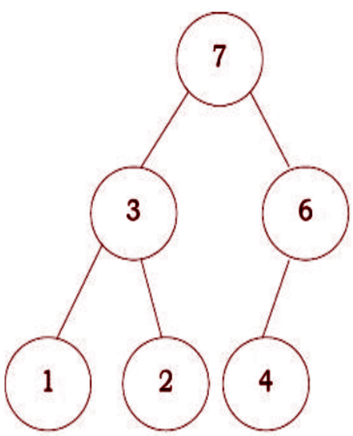
Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	$logn$ (average)	$logn$ (average)	$logn$ (average)
Balanced Binary Search Trees	$logn$	$logn$	$logn$
Binary Heaps	$logn$	$logn$	1

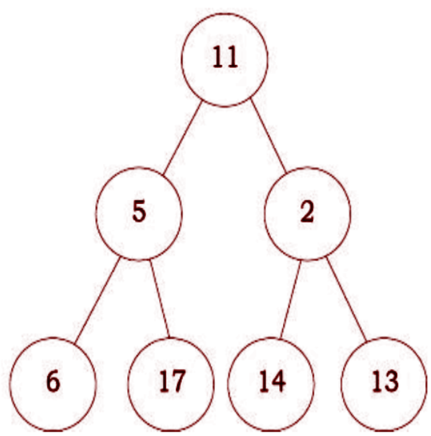
Heaps

- A heap is a tree with some special properties.
- The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children. This is called the **heap property**.
- A heap also has the additional property that all leaf nodes should be at **h** or **h – 1** level (where h is the height of the tree) for some h > 0 (complete binary trees).
- That means the heap should form a complete binary tree (as shown below).



In the examples below, the left tree is a heap (each element is greater than its children) and the right tree is not a heap (since 11 is greater than 2).

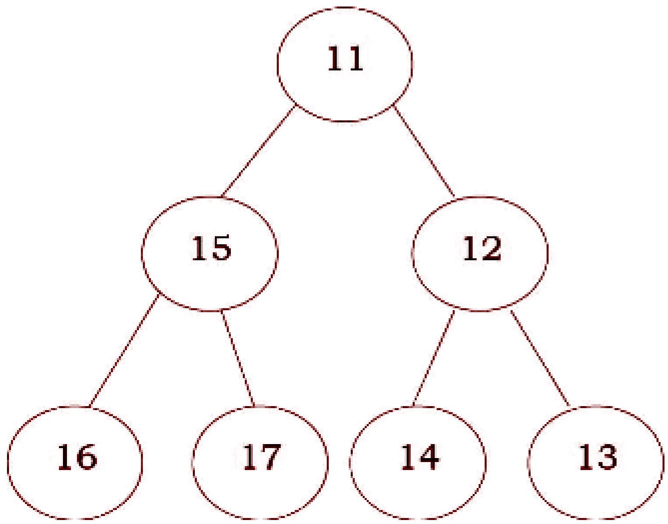




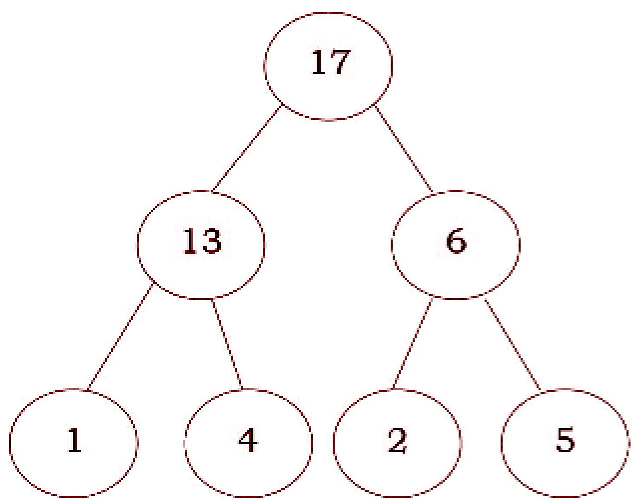
Types of Heaps

Based on the property of a heap we can classify heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children.



- **Max heap:** The value of a node must be greater than or equal to the values of its children.



Binary Heaps

- In a binary heap, each node may have up to two children.

- In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for the remaining discussion.

Representing Heaps: Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the discussion below let us assume that elements are stored in arrays, which starts at index 0. The previous max heap can be represented as:

17	13	6	1	4	2	5
0	1	2	3	4	5	6

Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

Heapify

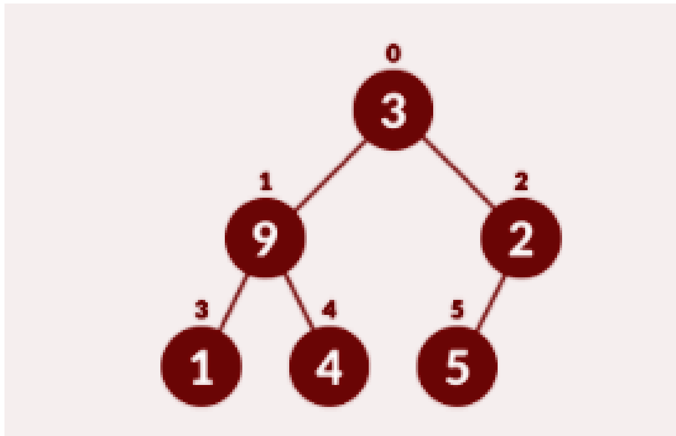
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

- Let the input array be

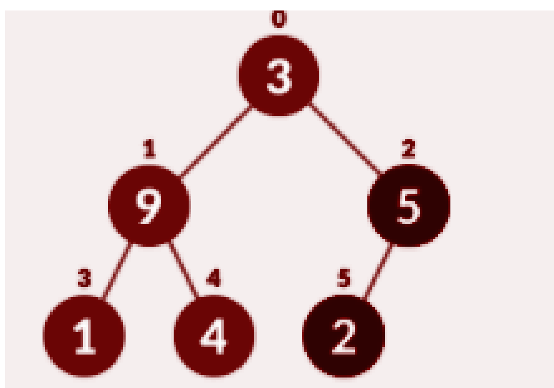
3	9	2	1	4	5
0	1	2	3	4	5

- Create a complete binary tree from the array
- Start from the first index of the non-leaf node whose index is given by **n/2 - 1**.

- Set current element **i** as **largest**.



- The index of the left child is given by **2i + 1** and the right child is given by **2i + 2**.
- If **leftChild** is greater than **currentElement** (i.e. element at the ith index), set **leftChildIndex** as largest. `#Condition1`



- If **rightChild** is greater than element in **largest**, set **rightChildIndex** as **largest**. `#Condition2`
- Swap **largest** with **currentElement.** `#Condition3`
- Repeat steps 3-7 until the subtrees are also heapified.
- For Min-Heap, both **leftChild** and **rightChild** must be smaller than the parent for all nodes.

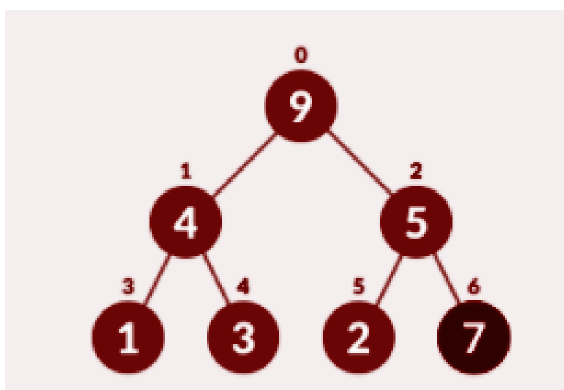
Python Code

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1 #Index of Left Child
    r = 2 * i + 2 #Index of Right Child

    if l < n and arr[i] < arr[l]: #Condition1
        largest = l

    if r < n and arr[largest] < arr[r]: #Condition2
        largest = r

    if largest != i: #Condition3
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)
```

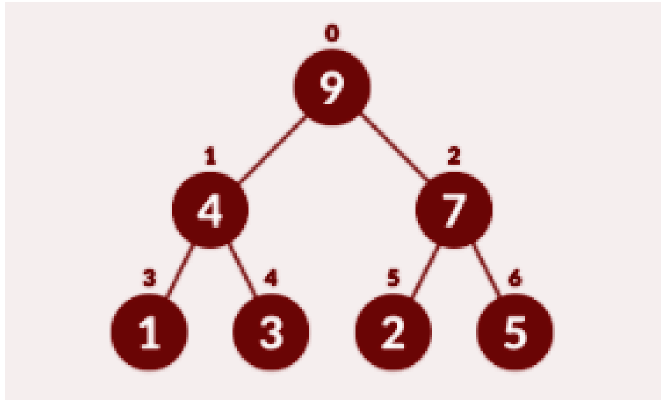
Insert Element into Heap

Insertion into a heap can be done in two steps:

- Insert the new element at the end of the tree. `#Step1`



- Heapify the tree. `#Step2`



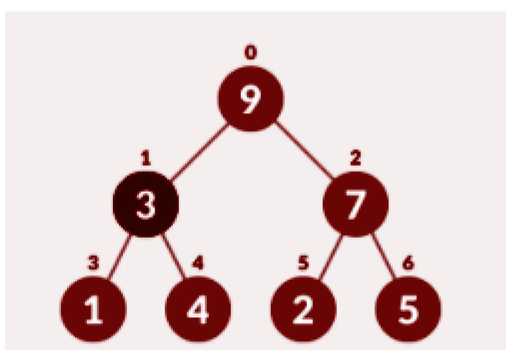
Python Code

```
def insert(array, newNum):
    size = len(array)
    if size == 0:#If empty heap initially
        array.append(newNum) #Simply add the newNum
    else:
        array.append(newNum);#Step1
        for i in range((size//2)-1, -1, -1):
            heapify(array, size, i) #Step2
```

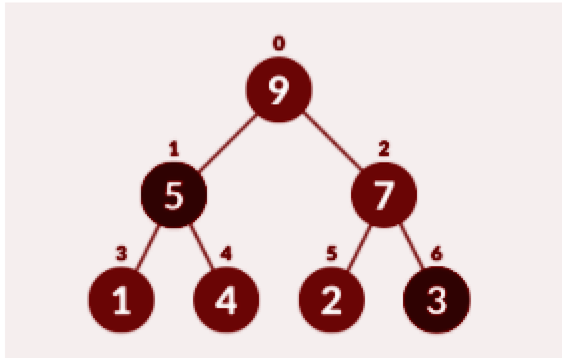
Delete Element from Heap

Follow the given steps to delete an element from a Heap:

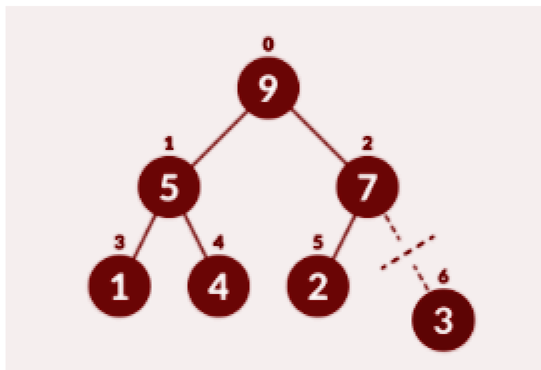
- Select the element to be deleted. `#Step1`



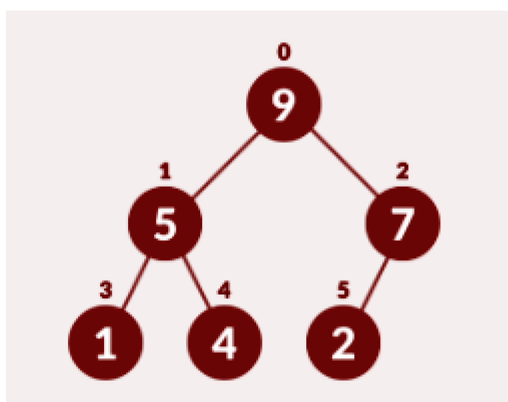
- Swap it with the last element.`#Step2`



- Remove the last element .#Step3



- Heapify the tree. #Step4



Python Code

```
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]: #Step1
            break

    array[i], array[size-1] = array[size-1], array[i] #Step2
    array.remove(size-1) #Step3
    for i in range((len(array)//2)-1, -1, -1):
        heapify(array, len(array), i) #Step4
```

Implementation of Priority Queue

- Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.
- Among these data structures, heap data structure provides an efficient implementation of priority queues.
- Hence, we will be using the heap data structure to implement the priority queue in this tutorial.

Insertion and Deletion in a Priority Queue

The first step would be to represent the priority queue in the form of a max/min-heap. Once it is heapified, the insertion and deletion operations can be performed similar to that in a Heap. Refer to the codes discussed above for more clarity.

Priority Queues- 2

Heap Sort

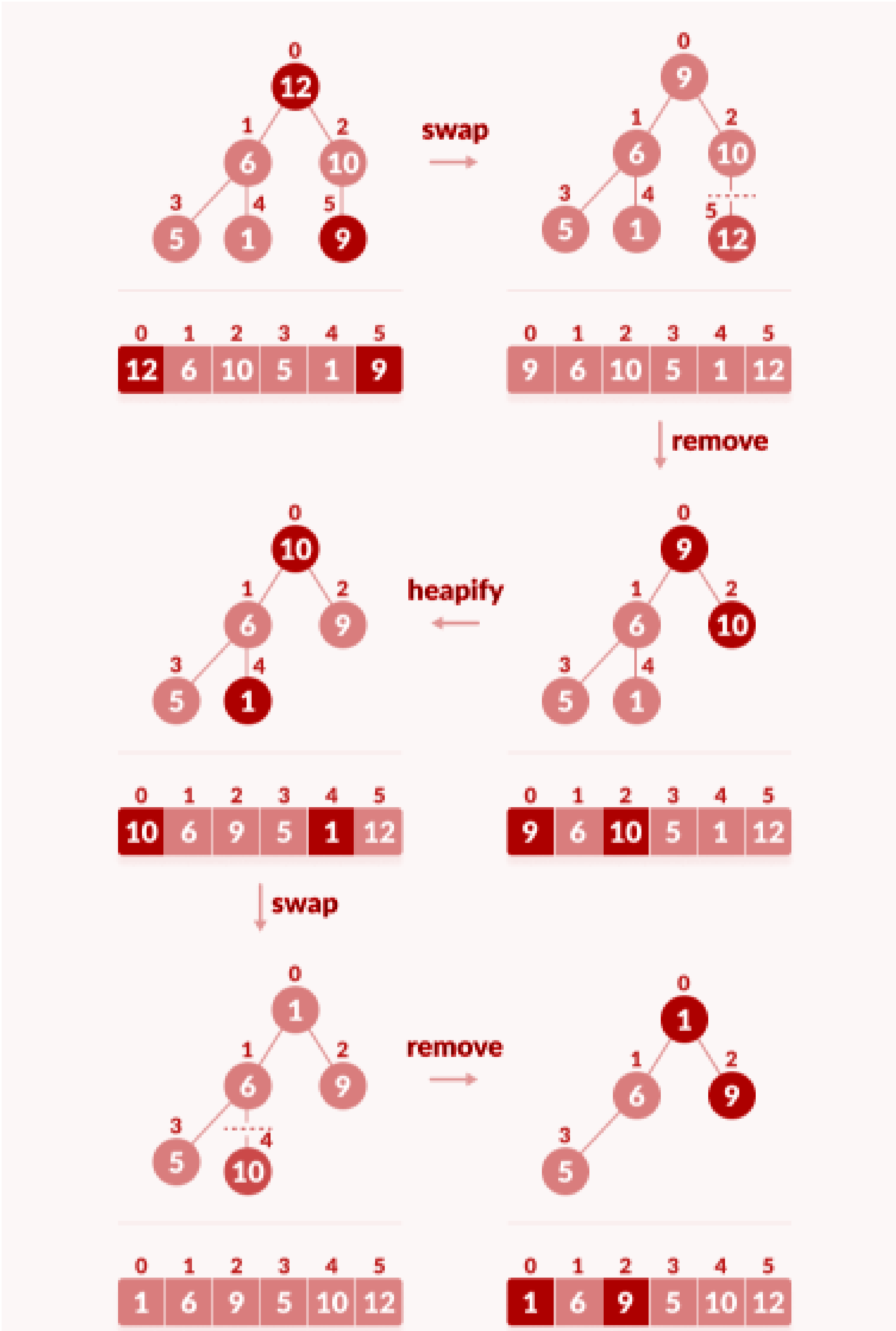
- Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a great worst-case runtime of **O(nlog(n))** regardless of the input data.
- As the name suggests, Heap Sort relies heavily on the heap data structure - a common implementation of a Priority Queue.
- Without 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.

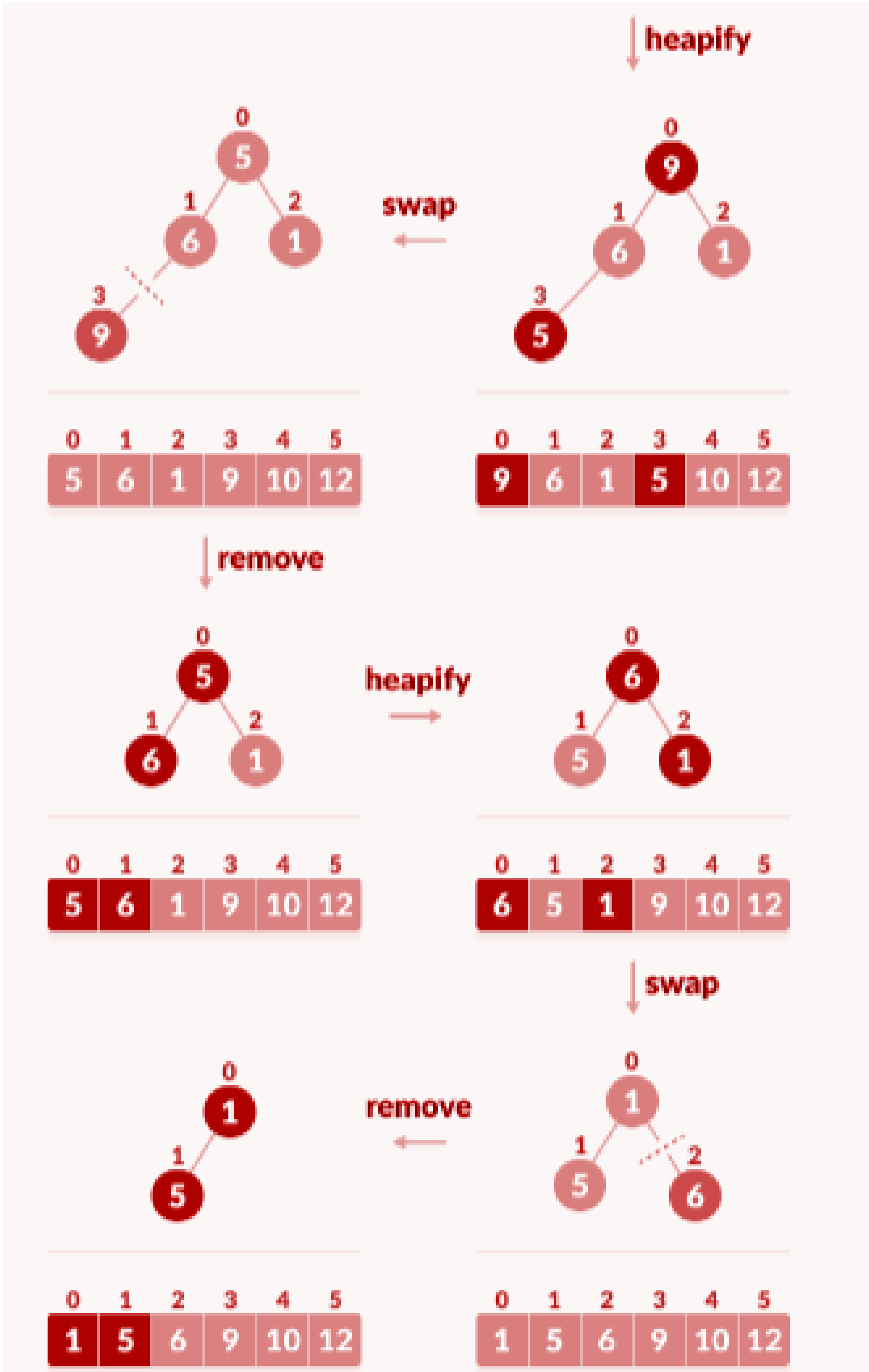
Algorithm

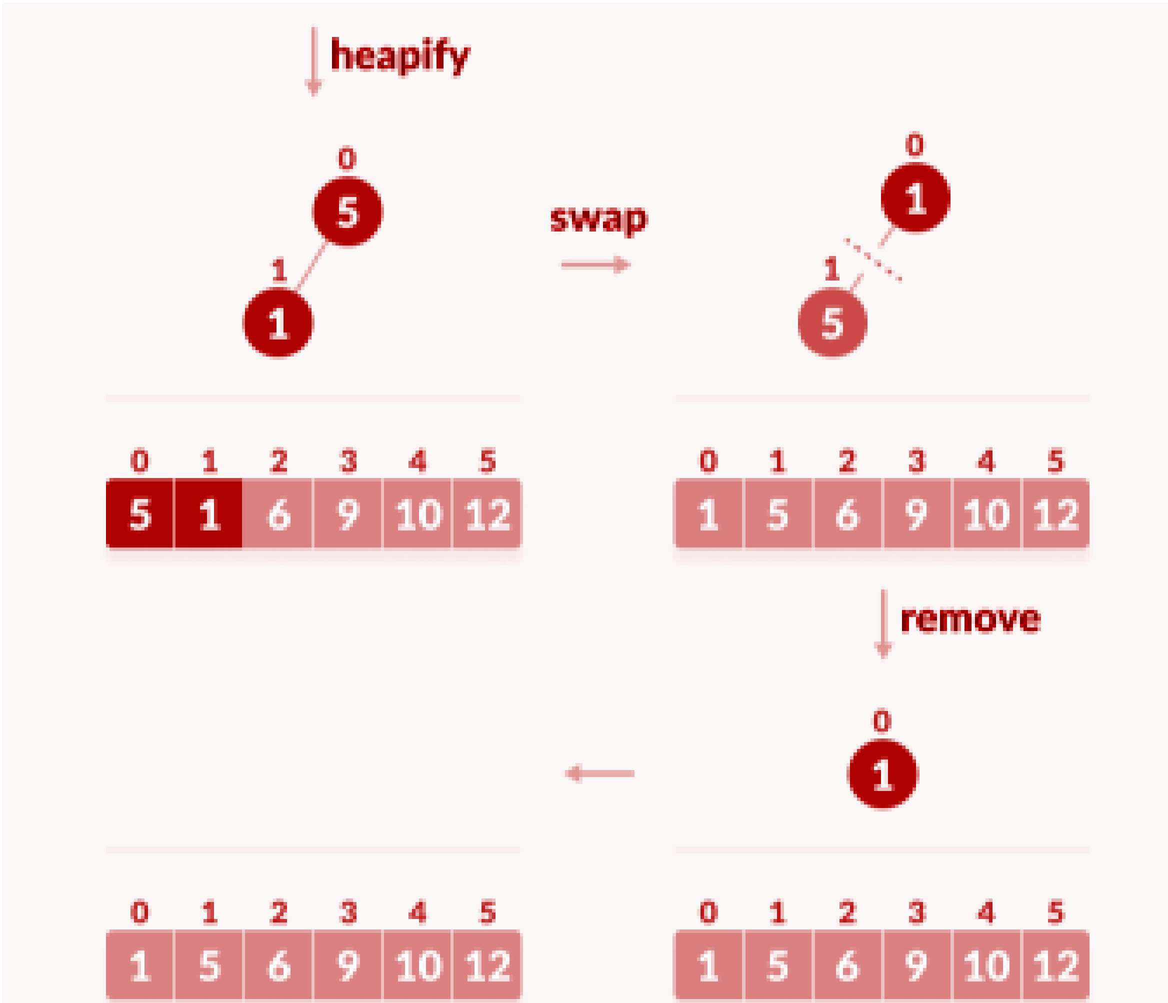
- The heap-sort algorithm inserts all elements (from an unsorted array) into a maxheap.
- Note that heap sort can be done **in-place** with the array to be sorted.
- Since the tree satisfies the Max-Heap property, then the largest item is stored at the root node.
- **Swap**: Remove the root element and 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.