**Heap**

Table of Contents

[**Heap Concepts** 2](#_Toc208549893)

[What is heap? 2](#_Toc208549894)

[Heap implementation in python 3](#_Toc208549895)

[Custom Heap Implementation 4](#_Toc208549896)

[LEVEL 1: **EASY** 5](#_Toc208549897)

[1. Kth Largest Element in an Array 5](#_Toc208549898)

[2. Kth Largest Element in a stream 5](#_Toc208549899)

[3. Final Array State After K Multiplication Operations I 5](#_Toc208549900)

[4. Sort Array by Increasing Frequency 5](#_Toc208549901)

[5. Minimum Amount of time to Fill Cups 5](#_Toc208549902)

[6. Find Subsequence of Length K with Largest Sum 5](#_Toc208549903)

[7. Find K closest elements 5](#_Toc208549904)

[**LEVEL** 2: **Medium** 6](#_Toc208549905)

[1. Minimum cost of ropes 6](#_Toc208549906)

[LEVEL 3: **Difficult** 7](#_Toc208549907)

[**SOLUTIONS:** 8](#_Toc208549908)

[**LEVEL 1:** 8](#_Toc208549909)

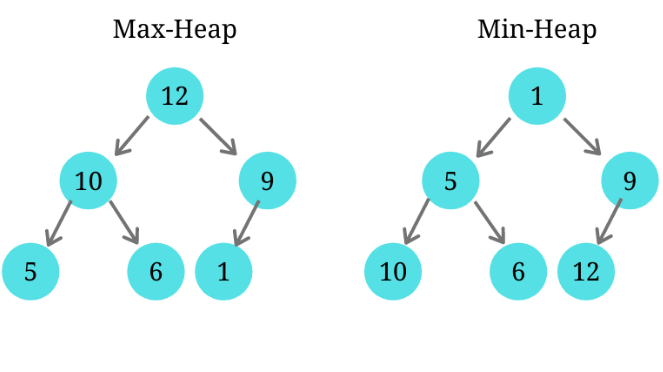
[**LEVEL 2:** 12](#_Toc208549910)

# **Heap Concepts**

## What is heap?

A **Heap** is a specialized **complete binary tree** that satisfies the **heap property**:

* **Max Heap**: Parent is **greater than or equal** to its children.
* **Min Heap**: Parent is **less than or equal** to its children.

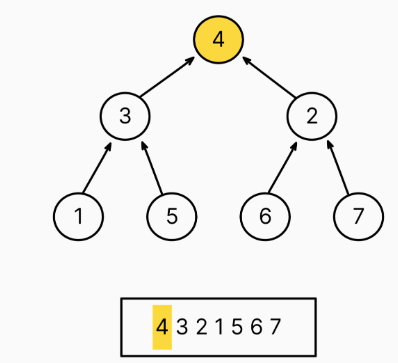


**Key Properties:**

1. **Complete Binary Tree**: All levels are filled except maybe the last, which is filled left to right.
2. Usually implemented as **array** (index-based binary tree).
3. Commonly used in:

* Priority Queues
* Heap Sort
* Dijkstra’s Algorithm
* Median finding
* Scheduling Systems

**Heap can be expressed as array:**



For node at index i:

* Left child: 2\*i + 1
* Right child: 2\*i + 2
* Parent: (i - 1) // 2

## Heap implementation in python

Python uses **Min Heap** by default. For max heap make values negative.

import heapq

#min heap working

heap = []

heapq.heappush(heap, 5)

heapq.heappush(heap, 1)

heapq.heappush(heap, 3)

print(heapq.heappop(heap))  # 1 (min element)

#max heap, implement by making values negative

heap = []

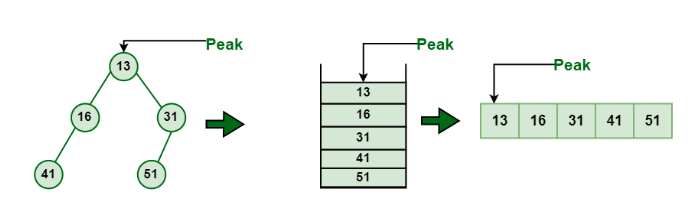
heapq.heappush(heap, -5)

heapq.heappush(heap, -1)

heapq.heappush(heap, -3)

print(-heapq.heappop(heap))  # 5 (max element)

Heap push and pop can be imagined as stack, for min heap top value at stack is minimum. Pop from min heap gives min value



**Time Complexities:**

1. Insert : O(logn)
2. Get Min/Max: O(1)
3. Delete Min/Max : O(logn)
4. Heapify: O(logn)

* **Insertion**(heapify-up): Add at the end, swap with parent if violating the property
* **Deletion**(heapify-down): Remove root, replace with last element, restore heap property

**When to use heap:**

1. If asked kth largest: use min heap (of size k, if heap size exceeds k, pop the smallest)
2. If asked kth smallest: use max heap

Can use sorting too in above cases, but sorting is O(nlogn). Whereas using heap it will be O(nlogk).

## Custom Heap Implementation

class MinHeap:

    def \_\_init\_\_(self):

        self.heap = []

    def insert(self, val):

        self.heap.append(val)

        self.\_heapify\_up(len(self.heap) - 1)

    def pop(self):

        if len(self.heap) == 0:

            return None

        val = self.heap[0]

        self.heap[0] = self.heap.pop()

        self.\_heapify\_down(0)

        return val

    def \_heapify\_up(self, i):

        parent = (i - 1) // 2

        if i > 0 and self.heap[i] < self.heap[parent]:

            self.heap[i], self.heap[parent] = self.heap[parent], self.heap[i]

            self.\_heapify\_up(parent)

    def \_heapify\_down(self, i):

        smallest = i

        left = 2 \* i + 1

        right = 2 \* i + 2

        if left < len(self.heap) and self.heap[left] < self.heap[smallest]:

            smallest = left

        if right < len(self.heap) and self.heap[right] < self.heap[smallest]:

            smallest = right

        if smallest != i:

            self.heap[i], self.heap[smallest] = self.heap[smallest], self.heap[i]

            self.\_heapify\_down(smallest)

**Heap Sort**

def heap\_sort(arr):

    heapq.heapify(arr)  # In-place min-heap

    return [heapq.heappop(arr) for \_ in range(len(arr))]

print(heap\_sort([4, 1, 3, 2, 16, 9]))  # [1, 2, 3, 4, 9, 16]

# LEVEL 1: **EASY**

### Kth Largest Element in an Array

Link: <https://leetcode.com/problems/kth-largest-element-in-an-array/>

### Kth Largest Element in a stream

Link: <https://leetcode.com/problems/kth-largest-element-in-a-stream/>

### Final Array State After K Multiplication Operations I

Link: <https://leetcode.com/problems/final-array-state-after-k-multiplication-operations-i/>

### Sort Array by Increasing Frequency

Link: <https://leetcode.com/problems/sort-array-by-increasing-frequency/>

### Minimum Amount of time to Fill Cups

Link: <https://leetcode.com/problems/minimum-amount-of-time-to-fill-cups/>

### Find Subsequence of Length K with Largest Sum

Link: <https://leetcode.com/problems/find-subsequence-of-length-k-with-the-largest-sum/>

### Find K closest elements

Link: <https://leetcode.com/problems/find-k-closest-elements/description/>

# **LEVEL** 2: **Medium**

### Minimum cost of ropes

Link: <https://www.geeksforgeeks.org/problems/minimum-cost-of-ropes-1587115620/1>

# LEVEL 3: **Difficult**

# **SOLUTIONS:**

## **LEVEL 1:**

1. **Kth largest Element in an Array**

import heapq

class Solution:

    def findKthLargest(self, nums: List[int], k: int) -> int:

        # for kth largest, use min heap

        heap = []

        for num in nums:

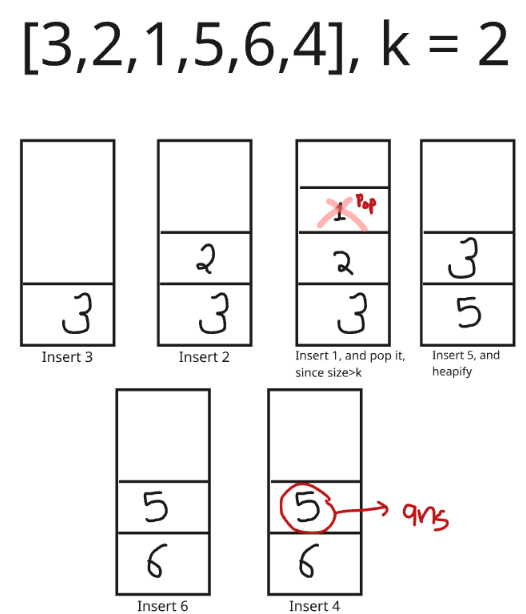
            heapq.heappush(heap,num)

            if len(heap)>k:

                heapq.heappop(heap)

        # Can also do heap[0]

        return heapq.heappop(heap)



Whenever there is K and ask about smallest or largest, then use heap

It changes problem complexity from nlog(n)🡪 nlog(k)

**2. Kth Largest Element in a Stream**

class KthLargest(object):

    def \_\_init\_\_(self, k, nums):

        self.k = k

        self.nums = nums

        heapq.heapify(self.nums)  # Convert nums into a heap

        # If the heap is larger than k, remove the smallest elements until it has exactly k elements

        while len(self.nums) > k:

            heapq.heappop(self.nums)

    def add(self, val):

        heapq.heappush(self.nums, val)

        # If after adding the new value, the heap has more than k elements, pop the smallest

        if len(self.nums) > self.k:

            heapq.heappop(self.nums)

        # The smallest element in the heap is now the k-th largest element

        return self.nums[0]

**3. Final Array State After K Multiplication Operations |**

import heapq

class Solution:

    def getFinalState(self, nums: List[int], k: int, multiplier: int) -> List[int]:

        mh = [[nums[i],i] for i in range(len(nums))]

        #if pair is passed to heapify, it executes logic considering first value from pair

        heapq.heapify(mh)     #can directly convert array to heap using heapify

        for \_ in range(k):

            v,ind = heapq.heappop(mh)

            v = v\*multiplier

            nums[ind] = v

            heapq.heappush(mh, [v,ind])

        return nums

**4. Sort Array by Increasing Frequency**

If in heap we push 2 values [x,y] , if x is same for 2 values, y is considered for sorting.

class Solution:

    def frequencySort(self, nums: List[int]) -> List[int]:

        c = Counter(nums)

        heap = []

        # min heap based on frequecy

        # if freq is same, larger value should come first

        for k,v in c.items():

            heapq.heappush(heap,[v,-k])

        ans=[]

        while heap:

            freq,key = heapq.heappop(heap)

            ans += [-key]\*freq

        return ans

**5. Minimum Amount of Time to Fill Cups**

We use a max-heap to always take the two largest cup amounts at each step. Reduce them by 1 (if nonzero), increase time, and push back. Repeat until all values are zero. This ensures the minimum time to fill all cups.

class Solution:

    def fillCups(self, amount: List[int]) -> int:

        time = 0

        amount = [-x for x in amount] #since here we need max heap

        heapq.heapify(amount)

        while True:

            v = heapq.heappop(amount)

            v2 = heapq.heappop(amount)

            if v==v2==0: #means all values are 0, so work is done

                return time #all ans will be returned from here

            if v!=0:

                v = v+1

            if v2!=0:

                v2 = v2+1

            time+=1

            heapq.heappush(amount,v)

            heapq.heappush(amount,v2)

        return time

**6. Find Subsequence of Length K With the Largest Sum**

**Approach:**

1. **Pair each element with its index** → [value, index].
2. **Sort in descending order** of values.
3. **Select the top** k **elements** (largest values).
4. **Sort these** k **elements by their original indices** to maintain order.
5. **Extract the values** to form the result.

class Solution:

    def maxSubsequence(self, nums: List[int], k: int) -> List[int]:

        heap=[]

        #using max heap, will fetch top k  elements from heap

        for i in range(len(nums)):

            heapq.heappush(heap,[-nums[i],i])

        #don't take top k like this, you have to pop it

#as heap just guranetees that top value is sorted

        # a = heap[:k] (wrong way)

        a = [heapq.heappop(heap) for \_ in range(k)]

        a.sort(key=lambda x:x[1])

        return [-val[0] for val in a]

**7. Find K closest elements**

**Approach:** The code builds a max-heap of size k, storing elements by their distance from x. If heap grows beyond k, farthest element is removed. Finally, the remaining k closest elements are extracted, converted back, and sorted before returning.

class Solution:

    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:

        # sort on basis of difference between num and x

        heap = []

        #need k elements with minimum difference with x

        for num in arr:

            heapq.heappush(heap,[-abs(num-x),-num])

            if len(heap)>k:

                heapq.heappop(heap)

        ans = []

        for \_ in range(k):

            val = heapq.heappop(heap)

            ans.append(-val[1])

        return sorted(ans)

## **LEVEL 2:**

1. Minimum Cost of Ropes

#Approach, on joining top 2 minimum length ropes, we can get min cost

# 4,3,2,6  Join 2,3  =>   [4,5,6] cost=5

#          Join 4,5  =>   [9,6]  cost=5+9

#          Join 9,6  =>   [15]  cost= 5+9+15 = 29

#Try other approach, only this gives best answer

import heapq

class Solution:

    def minCost(self, arr):

        heapq.heapify(arr)

        cost=0

        while(len(arr)!=1):

            x = heapq.heappop(arr)

            y = heapq.heappop(arr)

            cost += (x+y)

            heapq.heappush(arr,(x+y))

        return cost