—Dictionary keeps a complete binary tree, which is used to check data efficiently to get min/max element inside the tree structure.
— Each element is $\text{dict}[k]$
— Indices : $\text{idx}[\,]$ [1,4,2,3,5] → indices parent node
$\text{idx}[\,]$ [2,4,1,3,5] → → left child node
$\text{idx}[\,]$ [3,4,1,2,5] → → right child node
—Operations on the heap
$\text{update}(\,)$ → adjusts child elements of a subtree. $O(1)$
$\text{pop}(\,)$ → → → → → → min/max. $O(1)$
$\text{extract}(k)$ → converts value element from heap. $O(\log N)$
this operation needs to maintain the heap property by calling $\text{bubble}(\,)$.
$\text{decrease}(k)$ → decrease the value of key. $O(\log N)$
if decreased key makes it a child to parent then parent of child, then we don't need to do anything. otherwise, need to traverse up to fix violating properties.
$\text{insert}(\,)$ → add a new key-val at the tree. $O(\log N)$
if key is greater than parent value, then we don't need to perform heapify function.
$\text{delete}(\,)$ → implies the key-val, also subtree, which by calling $\text{decrease}(\text{key})$, etc. could be extracted(key), to remove the key. $O(\log N)$
—Applications: —Dijkstra → uses binary heap to get min or max in $O(\log n)$ time.
—Priority queue → efficiently implemented using queues here because it supports $\text{insert}(\,)$, $\text{delete}(\,)$, $\text{extract}(\,)$, $\text{decrease}(\,)$ in $O(\log n)$ time.
— Heap sort → implement efficient time complexity heap.
— Graph algo → especially used in graph algo like: Dij, Prim's, shortest path, and Kruskal's MST algorithm.
— Min heap → implement min in binary heap. Dij's algo, shortest path, and Kruskal's MST algorithm.

Implement Monotonic/Binary Stack using array

Heap: Usually complete binary tree, represented using array in computer and stored in array.

Monotonic: Maintaining a stack, which follows O(N) order of N elements.

Binary Search: Searching for a target in sorted array and performs O(log N) time.

Solution: optimized by maintaining the last track index and calculating height. Idea is to find the position of last smaller/greater index in stack. Then, height calculation is easier. Consider ind be last index, idx be current index, ht be current ht. Then, $\text{ht} = (\text{idx} - \text{ind} - 1)$.

```
stack = [-1]
for i in range(len(heights)):
    while stack[-1] != -1 and heights[i] < heights[stack[-1]]:
        ind = stack.pop()
        ht = (i - ind - 1)
        ans = max(ans, heights[ind] * ht)
    stack.append(i)
while stack[-1] != -1:
    ind = stack.pop()
    ht = (len(heights) - ind - 1)
    ans = max(ans, heights[ind] * ht)
return ans
```

Find an array using Heap (Heapify)

—Implicitly binary tree, represented using array. Parent node always is greater than its children and vice versa.

Heapify: Convert an array into a max/min heap. Start from last non-leaf node and perform sink down operation to each other-sized node, in reverse order. This heapify operation is more efficient than sorting, which takes $O(n^2)$ time.

— Last non-leaf node, can be found at $\text{last node idx} = \left\lfloor \frac{(n-1)}{2} \right\rfloor$, $n = \left\lfloor \frac{(n-1)}{2} \right\rfloor$.

```
def heapify(self):
    for i in range(self.size // 2, -1, -1):
        self.sink_down(i)
def sink_down(self, i):
    while True:
        left = self.left(i)
        right = self.right(i)
        largest = i
        if left < self.size and self.heap[left] > self.heap[largest]:
            largest = left
        if right < self.size and self.heap[right] > self.heap[largest]:
            largest = right
        if largest == i:
            break
        self.swap(i, largest)
        i = largest
def left(self, i):
    return 2 * i + 1
def right(self, i):
    return 2 * i + 2
def swap(self, i, j):
    self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
```

Maximum of all subarrays of size K

(Sliding window maximum)

1. Initialize an empty priority queue, heap element in decreasing order of their value, along with their index.
2. Place the first k elements into priority queue along with their indices.
3. Maximum element in the window is obtained by accessing the top element. Print this element to the
4. During the remaining elements of array, perform following:
   — add current element along with its index, to priority queue heap.
   — remove element from priority queue heap that are not within current window. This is done comparing index of top element in heap with $i - k$.
   If index of element top element is less than or equal to $(i - k)$, it means element is outside current window and should be removed.
   — Print element in current window by obtained by accessing the top element of priority queue heap after this new element is added.
   — Gradually slide the window forward, continuing this process until the end of the array is reached.

```
def maxSlidingWindow(nums, k):
    ans = []
    heap = []
    for i in range(k):
        heapq.heappush(heap, (-nums[i], i))
    ans.append(-heap[0][0])
    for i in range(k, len(nums)):
        while heap[0][1] <= i - k:
            heapq.heappop(heap)
        heapq.heappush(heap, (-nums[i], i))
        ans.append(-heap[0][0])
    return ans
```

Using Set

—To remove another value in O(1)
Return Maximum in $O(\log n)$ time. Time and add remove overall $O(n)$ are $O(\log n)$. Find $(k-1)$-th element from set using iteration.

Find K-th largest element in unsorted array

Using Sorting
— Sort the array in descending order in first line. Find k-th element by the array index. $O(n \log n)$

Using Binary Search $O(\log(\max - \min))$
— Idea is to find k^{th} largest element of array and then find all elements which are greater than or equal to k^{th} largest element.
— k^{th} largest element can be found using binary search by defining a search range based on min and max, where to search array.
— in case function of binary search, check for, keep than mid-point and update the search range accordingly.
— This process continues until the range collapses to a single element, which is k^{th} largest element.

Using Quickselect Partitioning $O(n^2)$ in worst case, $O(n)$ in average
$O(n)$ space.

Using Priority Queue (Min-Heap) $O(n \log(k))$ time.
— Initialize a min-heap $O(n \log(k))$ space.
— For each element
   — Push the element into heap
   — If size of heap exceeds k, pop the smallest element from the smallest element from the heap. Ensures that only the k largest elements remain in the heap.
— After processing the all elements, the heap will contain k largest elements of array.

```
heapq = []
heap = []
num = []
heapq.heapify()
for i in range(len(nums)):
    heapq.heappush(heap, nums[i])
    if len(heap) > k:
        heapq.heappop(heap)
while len(heap) > 0:
    num.append(heapq.heappop(heap))
return num
```

Find Maximum Sum Combination

Heap: Complete binary tree, represented using array. Parent node always is greater than its children.

Idea: Find k largest sum combinations from two arrays. Sort both arrays in descending order. Use a max-heap to track potential sum combinations. Start with the largest sum combination (the sum of the first elements of both arrays). Extract the largest sum from the heap and add it to the result. Generate new sum combinations by adding the next elements from both arrays to the extracted sum. Ensure uniqueness by using a set to track visited indices. Continue until k sum combinations are found.

```
def getMaxSum(A, B, k):
    A.sort(reverse=True)
    B.sort(reverse=True)
    heap = [(-A[0] + B[0], 0, 0)]
    visited = {(0, 0)}
    result = []
    while len(result) < k:
        max_sum, i, j = heapq.heappop(heap)
        result.append(-max_sum)
        if (i + 1, j) not in visited:
            heapq.heappush(heap, (-A[i + 1] + B[j], i + 1, j))
            visited.add((i + 1, j))
        if (i, j + 1) not in visited:
            heapq.heappush(heap, (-A[i] + B[j + 1], i, j + 1))
            visited.add((i, j + 1))
    return result[:k]
```

K-th largest element in a stream

Keep the results in heap instead of continuous sorting within the array.

k^{th} largest element is the root of the heap. Use a min-heap to maintain the top k elements. When a new element arrives, compare it with the root. If the new element is larger than the root, replace the root with the new element and adjust the heap to maintain the heap property. This ensures that the root always contains the kth largest element.

```
class KthLargest:
    def __init__(self, k, nums):
        self.k = k
        self.heap = []
        for num in nums:
            self.add(num)
    def add(self, val):
        import heapq
        if len(self.heap) < self.k:
            heapq.heappush(self.heap, val)
        else:
            if val > self.heap[0]:
                heapq.heappop(self.heap)
                heapq.heappush(self.heap, val)
        return self.heap[0]
```

Heap: K-sorted Sorted Set

Idea: Initially insert k elements into heap. Then iterate, add new element into heap and remove the smallest element from heap. Continue this process until all elements are processed. The heap maintains the k smallest elements at any point, ensuring efficient retrieval of the kth smallest element.

Time and Complexity: — Insert: O(log k) for each element, resulting in O(n log k) for n elements.
— Extract: O(1) to access the kth smallest element.
— Space: O(k) for storing k elements.

```
import heapq
def kthSmallest(nums, k):
    heap = []
    for num in nums:
        heapq.heappush(heap, -num)
        if len(heap) > k:
            heapq.heappop(heap)
    return -heap[0]
```

Find Kth Smallest Element in Two Sorted Arrays

Idea: Combine two arrays into a single array. Sort the combined array. The kth element in the sorted array is the kth smallest element.

```
def kthSmallest(arr1, arr2, k):
    combined = arr1 + arr2
    combined.sort()
    return combined[k - 1]
```

Heap: K-sorted Array

Heapify: Convert an array into a heap. Start from last non-leaf node and perform sink down operation to each other-sized node, in reverse order.

Idea: Initialize a min-heap with the first k elements from the array. Extract the smallest element from the heap and add it to the result. Replace the extracted element with the next element from the array. Continue this process until all elements are processed.

```
import heapq
def kSortedArray(arr, k):
    heap = []
    for i in range(k):
        heapq.heappush(heap, arr[i])
    for i in range(k, len(arr)):
        smallest = heapq.heappop(heap)
        heapq.heappush(heap, arr[i])
        print(smallest, end=" ")
    while heap:
        smallest = heapq.heappop(heap)
        print(smallest, end=" ")
```

Heap: Min Heap Max Heap

Idea: Create an array to store the values. Copy data values into the array with indexes. Build min heap using heapify function. Create a max heap by inverting the values. Print min heap and max heap.

```
class MinHeap:
    def __init__(self, arr):
        self.heap = arr
        self.size = len(arr)
        self.build_heap()
    def build_heap(self):
        for i in range(self.size // 2, -1, -1):
            self.heapify(i)
    def heapify(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        smallest = i
        if left < self.size and self.heap[left] < self.heap[smallest]:
            smallest = left
        if right < self.size 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(smallest)
    def get_min(self):
        return self.heap[0]
    def extract_min(self):
        if self.size == 0:
            raise IndexError("Heap is empty")
        min_value = self.heap[0]
        self.heap[0] = self.heap[self.size - 1]
        self.size -= 1
        self.heapify(0)
        return min_value
    def insert(self, value):
        self.heap.append(value)
        self.size += 1
        self.build_heap()
    def print_heap(self):
        print(self.heap[:self.size])

class MaxHeap:
    def __init__(self, arr):
        self.heap = [-x for x in arr]
        self.size = len(arr)
        self.build_heap()
    def build_heap(self):
        for i in range(self.size // 2, -1, -1):
            self.heapify(i)
    def heapify(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        largest = i
        if left < self.size and self.heap[left] > self.heap[largest]:
            largest = left
        if right < self.size and self.heap[right] > self.heap[largest]:
            largest = right
        if largest != i:
            self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]
            self.heapify(largest)
    def get_max(self):
        return -self.heap[0]
    def extract_max(self):
        if self.size == 0:
            raise IndexError("Heap is empty")
        max_value = -self.heap[0]
        self.heap[0] = self.heap[self.size - 1]
        self.size -= 1
        self.heapify(0)
        return max_value
    def insert(self, value):
        self.heap.append(-value)
        self.size += 1
        self.build_heap()
    def print_heap(self):
        print([-x for x in self.heap[:self.size]])
```

K-th largest sum combinations subarray

Idea: Initialize a max-heap with the sum of the last elements from each array. Use a set to track visited indices. Extract the largest sum from the heap and add it to the result. Generate new combinations by decreasing the indices of the last elements. Ensure uniqueness by checking if the new combination has been visited. Continue until k combinations are found.

```
import heapq
def kthSum(A, B, k):
    A.sort(reverse=True)
    B.sort(reverse=True)
    heap = [(-A[-1] - B[-1], len(A) - 1, len(B) - 1)]
    visited = {(len(A) - 1, len(B) - 1)}
    result = []
    while len(result) < k:
        sum, i, j = heapq.heappop(heap)
        result.append(-sum)
        if (i - 1, j) not in visited:
            heapq.heappush(heap, (-A[i - 1] - B[j], i - 1, j))
            visited.add((i - 1, j))
        if (i, j - 1) not in visited:
            heapq.heappush(heap, (-A[i] - B[j - 1], i, j - 1))
            visited.add((i, j - 1))
    return result[:k]
```

Sorting Array

—Sorting array in descending order is from first. Find k-th element by the array index. $O(n \log n)$

Using Binary Search $O(\log(\max - \min))$
— Idea is to find k^{th} largest element of array and then find all elements which are greater than or equal to k^{th} largest element.
— k^{th} largest element can be found using binary search by defining a search range based on min and max, where to search array.
— in case function of binary search, check for, keep than mid-point and update the search range accordingly.
— This process continues until the range collapses to a single element, which is k^{th} largest element.

Using Quickselect Partitioning $O(n^2)$ in worst case, $O(n)$ in average
$O(n)$ space.

Using Priority Queue (Min-Heap) $O(n \log(k))$ time.
— Initialize a min-heap $O(n \log(k))$ space.
— For each element
   — Push the element into heap
   — If size of heap exceeds k, pop the smallest element from the smallest element from the heap. Ensures that only the k largest elements remain in the heap.
— After processing the all elements, the heap will contain k largest elements of array.

```
import heapq
heap = []
heap = []
num = []
heapq.heapify()
for i in range(len(nums)):
    heapq.heappush(heap, nums[i])
    if len(heap) > k:
        heapq.heappop(heap)
while len(heap) > 0:
    num.append(heapq.heappop(heap))
return num
```

— not quickselect algorithm
— use sorting instead

Heap: K-sorted Array

Heapify: Convert an array into a heap. Start from last non-leaf node and perform sink down operation to each other-sized node, in reverse order.

Idea: Initialize a min-heap with the first k elements from the array. Extract the smallest element from the heap and add it to the result. Replace the extracted element with the next element from the array. Continue this process until all elements are processed. The heap maintains the k smallest elements at any point, ensuring efficient retrieval of the kth smallest element.

```
import heapq
def kSortedArray(self, arr, k):
    n = len(arr)
    if n == 0:
        return []
    heap = [arr[i] for i in range(min(k, n))]
    heapq.heapify(heap)
    result = []
    for i in range(k, n):
        smallest = heapq.heappop(heap)
        result.append(smallest)
        heapq.heappush(heap, arr[i])
    while heap:
        smallest = heapq.heappop(heap)
        result.append(smallest)
    return result
```

Dijkstra: Shortest Path in Weighted Graph

Idea: Use a min-heap to keep track of the shortest distances to vertices. Initialize distances with infinity except for the source vertex. Extract the vertex with the smallest distance from the heap. Update distances to its neighbors if a shorter path is found. Continue until all vertices are processed.

```
import heapq
def dijkstra(graph, start):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
```

Find Minimum Sum Combinations

Idea: Create two arrays. Sort both arrays in ascending order. Use a min-heap to track potential sum combinations. Start with the smallest sum (the sum of the first elements of both arrays). Extract the smallest sum from the heap and add it to the result. Generate new sum combinations by adding the next elements from both arrays to the extracted sum. Ensure uniqueness by using a set to track visited indices. Continue until k sum combinations are found.

```
import heapq
def minSumCombinations(A, B, k):
    A.sort()
    B.sort()
    heap = [(A[0] + B[0], 0, 0)]
    visited = {(0, 0)}
    result = []
    while len(result) < k:
        sum, i, j = heapq.heappop(heap)
        result.append(sum)
        if (i + 1, j) not in visited:
            heapq.heappush(heap, (A[i + 1] + B[j], i + 1, j))
            visited.add((i + 1, j))
        if (i, j + 1) not in visited:
            heapq.heappush(heap, (A[i] + B[j + 1], i, j + 1))
            visited.add((i, j + 1))
    return result[:k]
```

Dijkstra: Shortest Path in Weighted Graph

Idea: Use a min-heap to keep track of the shortest distances to vertices. Initialize distances with infinity except for the source vertex. Extract the vertex with the smallest distance from the heap. Update distances to its neighbors if a shorter path is found. Continue until all vertices are processed.

```
import heapq
def dijkstra(graph, start):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]
    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)
        if current_distance > distances[current_vertex]:
            continue
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))
    return distances
```

Heap: Sliding Window Maximum

Idea: Use a max-heap to keep track of the maximum elements in the sliding window. Initialize the heap with the first k elements. Extract the maximum element from the heap and add it to the result. Replace the extracted element with the next element from the array. Continue this process until all elements are processed.

```
import heapq
def maxSlidingWindow(nums, k):
    n = len(nums)
    if n == 0:
        return []
    heap = [(-nums[i], i) for i in range(k)]
    heapq.heapify(heap)
    result = [-heap[0][0]]
    for i in range(k, n):
        while heap[0][1] <= i - k:
            heapq.heappop(heap)
        heapq.heappush(heap, (-nums[i], i))
        result.append(-heap[0][0])
    return result
```

Heap: K-sorted Array

Heapify: Convert an array into a heap. Start from last non-leaf node and perform sink down operation to each other-sized node, in reverse order.

Idea: Initialize a min-heap with the first k elements from the array. Extract the smallest element from the heap and add it to the result. Replace the extracted element with the next element from the array. Continue this process until all elements are processed. The heap maintains the k smallest elements at any point, ensuring efficient retrieval of the kth smallest element.

```
import heapq
def kSortedArray(self, arr, k):
    n = len(arr)
    if n == 0:
        return []
    heap = [arr[i] for i in range(min(k, n))]
    heapq.heapify(heap)
    result = []
    for i in range(k, n):
        smallest = heapq.heappop(heap)
        result.append(smallest)
        heapq.heappush(heap, arr[i])
    while heap:
        smallest = heapq.heappop(heap)
        result.append(smallest)
    return result
```

Heap: Kth Largest Element in Stream

Idea: Create an array to store the values. Copy data values into the array with indexes. Build min heap using heapify function. Add new elements to the heap. Print min heap.

```
class MinHeap:
    def __init__(self, arr):
        self.heap = arr
        self.size = len(arr)
        self.build_heap()
    def build_heap(self):
        for i in range(self.size // 2, -1, -1):
            self.heapify(i)
    def heapify(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        smallest = i
        if left < self.size and self.heap[left] < self.heap[smallest]:
            smallest = left
        if right < self.size 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(smallest)
    def get_min(self):
        return self.heap[0]
    def extract_min(self):
        if self.size == 0:
            raise IndexError("Heap is empty")
        min_value = self.heap[0]
        self.heap[0] = self.heap[self.size - 1]
        self.size -= 1
        self.heapify(0)
        return min_value
    def insert(self, value):
        self.heap.append(value)
        self.size += 1
        self.build_heap()
    def print_heap(self):
        print(self.heap[:self.size])
```

Heap: Kth Largest Element in Unsorted Array

—using sorting function
—using binarysearch $O(n \log(\max - \min))$
—using priority queue/min-heap
heapq
$O(n \log n)$ time
$O(k)$ space.
def kth_smallest(self, nums, k):
    heapq.heapify(nums)
    for num in nums:
        heapq.heappush(heapq, -num)
    if len(heapq) > k:
        heapq.heappop(heapq)
    return -heapq[0]
— not quickselect algorithm
— use sorting instead

Heap: K-sorted Array

Heapify: Convert an array into a heap. Start from last non-leaf node and perform sink down operation to each other-sized node, in reverse order.

Idea: 1. Initialize a min-heap with the first k elements from the array.
2. Extract the smallest element from the heap and add it to the result.
3. Replace the extracted element with the next element from the array.
4. Continue this process until all elements are processed.
5. Print the remaining elements in the heap.

```
import heapq
def kSortedArray(self, arr, k):
    n = len(arr)
    if n == 0:
        return []
    heap = [arr[i] for i in range(min(k, n))]
    heapq.heapify(heap)
    result = []
    for i in range(k, n):
        smallest = heapq.heappop(heap)
        result.append(smallest)
        heapq.heappush(heap, arr[i])
    while heap:
        smallest = heapq.heappop(heap)
        result.append(smallest)
    return result
```

Heap: Kth Largest Element in Stream

Idea: Create an array to store the values. Copy data values into the array with indexes. Build min heap using heapify function. Add new elements to the heap. Print min heap.

```
class MinHeap:
    def __init__(self, arr):
        self.heap = arr
        self.size = len(arr)
        self.build_heap()
    def build_heap(self):
        for i in range(self.size // 2, -1, -1):
            self.heapify(i)
    def heapify(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        smallest = i
        if left < self.size and self.heap[left] < self.heap[smallest]:
            smallest = left
        if right < self.size 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(smallest)
    def get_min(self):
        return self.heap[0]
    def extract_min(self):
        if self.size == 0:
            raise IndexError("Heap is empty")
        min_value = self.heap[0]
        self.heap[0] = self.heap[self.size - 1]
        self.size -= 1
        self.heapify(0)
        return min_value
    def insert(self, value):
        self.heap.append(value)
        self.size += 1
        self.build_heap()
    def print_heap(self):
        print(self.heap[:self.size])
```

Heap: Kth Largest Element in Unsorted Array

—using sorting function
—using binarysearch $O(n \log(\max - \min))$
—using priority queue/min-heap
heapq
$O(n \log n)$ time
$O(k)$ space.
def kth_smallest(self, nums, k):
    heapq.heapify(nums)
    for num in nums:
        heapq.heappush(heapq, -num)
    if len(heapq) > k:
        heapq.heappop(heapq)
    return -heapq[0]
— not quickselect algorithm
— use sorting instead

Heap: K-sorted Array

Heapify: Convert an array into a heap. Start from last non-leaf node and perform sink down operation to each other-sized node, in reverse order.

Idea: Initialize a min-heap with the first k elements from the array. Extract the smallest element from the heap and add it to the result. Replace the extracted element with the next element from the array. Continue this process until all elements are processed. Print the remaining elements in the heap.

```
import heapq
def kSortedArray(self, arr, k):
    n = len(arr)
    if n == 0:
        return []
    heap = [arr[i] for i in range(min(k, n))]
    heapq.heapify(heap)
    result = []
    for i in range(k, n):
        smallest = heapq.heappop(heap)
        result.append(smallest)
        heapq.heappush(heap, arr[i])
    while heap:
        smallest = heapq.heappop(heap)
        result.append(smallest)
    return result
```

Heap: Kth Largest Element in Stream

Idea: Create an array to store the values. Copy data values into the array with indexes. Build min heap using heapify function. Add new elements to the heap. Print min heap.

```
class MinHeap:
    def __init__(self, arr):
        self.heap = arr
        self.size = len(arr)
        self.build_heap()
    def build_heap(self):
        for i in range(self.size // 2, -1, -1):
            self.heapify(i)
    def heapify(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        smallest = i
        if left < self.size and self.heap[left] < self.heap[smallest]:
            smallest = left
        if right < self.size 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(smallest)
    def get_min(self):
        return self.heap[0]
    def extract_min(self):
        if self.size == 0:
            raise IndexError("Heap is empty")
        min_value = self.heap[0]
        self.heap[0] = self.heap[self.size - 1]
        self.size -= 1
        self.heapify(0)
        return min_value
    def insert(self, value):
        self.heap.append(value)
        self.size += 1
        self.build_heap()
    def print_heap(self):
        print(self.heap[:self.size])
```