WEEK 4

Aim: Implement Priority Queues using Heaps

Description:

A Priority Queue is an abstract data type where each element is associated with a priority, and elements are removed based on priority rather than insertion order. A Binary Heap is the most efficient and commonly used data structure to implement a priority queue. A binary heap is a complete binary tree that satisfies the heap property.
There are two types of binary heaps:

- **Max Heap: Parent node ≥ its children → highest priority at the root**
- **Min Heap: Parent node ≤ its children → lowest priority at the root**

In heap-based priority queues, the root element is always the highest (or lowest) priority, enabling efficient access and deletion.

Heap Properties

1. Complete Binary Tree
   - All levels are completely filled except possibly the last.
2. Heap Order Property
   - Max Heap: Parent ≥ children
   - Min Heap: Parent ≤ children
3. Array Representation
   - Parent index = $\lfloor (i-1)/2 \rfloor$
   - Left child = $2i + 1$
   - Right child = $2i + 2$

Algorithms :

Algorithm: Max Heap (Priority Queue)

Insert Operation (Max Heap)

1. Insert the new element at the end of the heap.
2. Set i to the index of the inserted element.
3. While $i > 0$ and heap[parent(i)] < heap[i]:
   - Swap heap[i] and heap[parent(i)]
   - Update i = parent(i)
4. Stop when heap property is satisfied.

Logic:
Move the element upward until parent is larger.

Delete Operation (Max Heap)

1. If heap is empty, print “Queue is empty”.
2. Store the root element (maximum).
3. Replace root with the last element.
4. Remove the last element.
5. Heapify down from root:
   - Compare with left and right children

   - Swap with the largest child
6. Repeat until heap property is restored.
7. Return the deleted element.

Examples:

Given Elements (Insertion Order)
[15, 40, 10, 60, 25, 55, 5, 30, 45, 20]

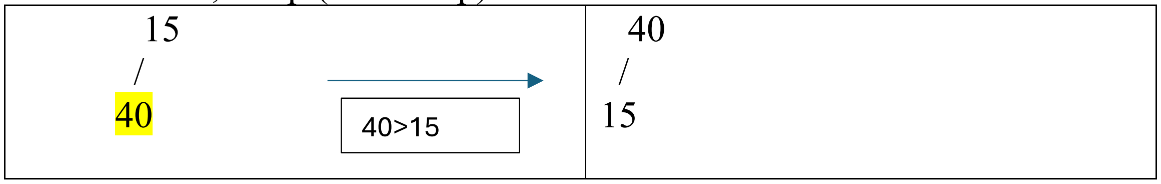
Step 1: Insert 15

- Heap is empty
- 15 becomes the root

15
Heap array: [15]

Step 2: Insert 40

- Insert at next available position (left child of 15)
- Since 40 > 15, swap (bubble up)



Heap array: [40, 15]

Step 3: Insert 10

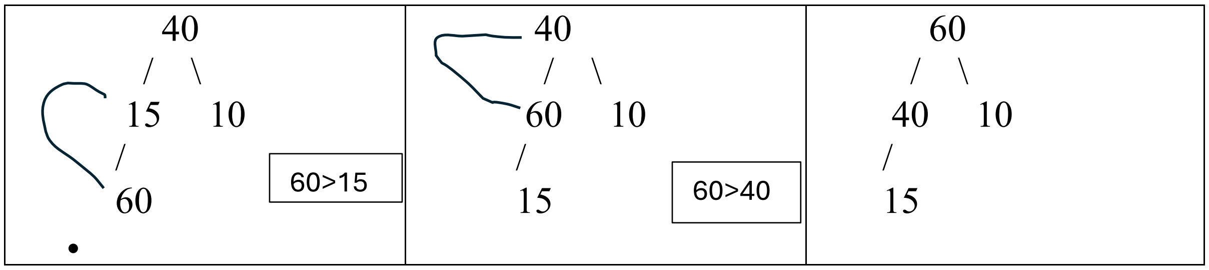
- Insert as right child of 40
- 10 < 40, no swap

```
  40
 /  \
15   10
```

Heap array: [40, 15, 10]

Step 4: Insert 60

- Insert as left child of 15
- 60 > 15 → swap
- 60 > 40 → swap again (bubble up to root)



Heap array: [60, 40, 10, 15]

Step 5: Insert 25

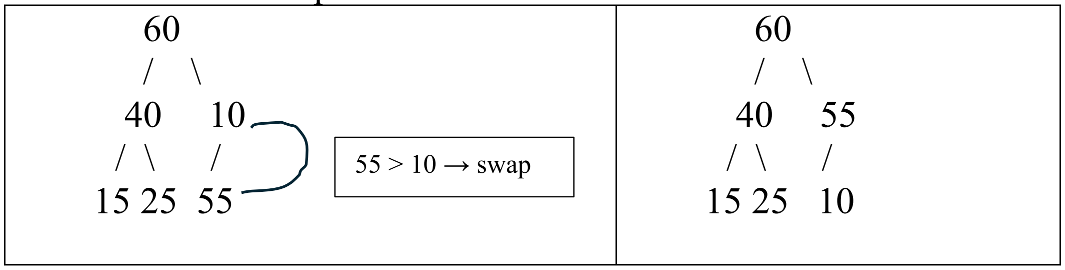
- Insert as right child of 40
- 25 < 40, heap property satisfied

```
    60
  /   \
 40    10
/  \
15 25
```

Heap array: [60, 40, 10, 15, 25]

Step 6: Insert 55

- Insert as left child of 10
- 55 > 10 → swap
- 55 < 60 → stop



Heap array: [60, 40, 55, 15, 25, 10]

Step 7: Insert 5

- Insert as right child of 55
- 5 < 55, no swap

```
      60
    /        \
 40          55
 /  \        /   \
15 25     10      5
```

Heap array: [60, 40, 55, 15, 25, 10, 5]

Step 8: Insert 30

- Insert as left child of 15
- 30 > 15 → swap
- 30 < 40 → stop

```
      60                                      60
    /      \                                /      \
  40        55                            40        55
 /  \      /   \      ┌───────────────┐   / \      /   \
15  25    10    5     │ 30 > 15 → swap │  30 25    10    5
 /                    └───────────────┘   /
30                                       15
```

Heap array: [60, 40, 55, 30, 25, 10, 5, 15]

Step 9: Insert 45

- Insert as right child of 30
- 45 > 30 → swap
- 45 > 40 → swap
- 45 < 60 → stop

```
|         60               |         60               |    60                    |
|       /      \           |       /      \           |   /      \               |
|     40        55         |     40        55         |  45        55            |
|    / \       /   \       |    / \       /   \       |  / \       /  \          |
|   30 25    10    5       |   45 25    10    5       | 40 25    10    5         |
|   / \                    |   / \                    | /  \                     |
| 15     45                | 15     30                | 15 30                    |
```

Heap array: [60, 45, 55, 40, 25, 10, 5, 15, 30]

Step 10: Insert 20

- Insert as left child of 25
- 20 < 25, no swap

```
     60
   /     \
 45       55
 /  \        /   \
40 25     10     5
/ \  /   \
15 30 20
```

Heap array:
[60, 45, 55, 40, 25, 10, 5, 15, 30, 20]

Final Max Heap

Tree Representation

```
       60
    /        \
  45          55
 /  \        /   \
40 25     10     5
/ \  /   \
15 30 20
```

Heap Array Representation

[60, 45, 55, 40, 25, 10, 5, 15, 30, 20]

Max Heap DELETION

Deletion 1: Delete 60

D1-Step 1: Remove root (60), move last element (20) to root

```
                    60                              |                    20
           /                 \                      |           /                 \
        45                        55                |        45                        55
     /        \               /        \            |     /        \               /        \
   40         25          10         5              |   40         25          10         5
   /    \      /                                    |   /    \
   15 30  20                                        |   15 30
```

D1-Step 2: Heapify down (swap 20 with larger child 55)

```
                   55
           /                 \
        45                        20
     /        \               /        \
      40         25          10         5
   /     \
        15 30
```

Heap after deleting 60
Heap array: 55,45,20,40,25,10,5,15,30

Deletion 2: Delete 55

D2-Step 1: Remove root (55), move last element (30) to root

```
                    55
            /                 \
          45                    20
      /        \            /       \
        40        25        10        5
     /    \
     15 30
```

```
      30
    /     \
  45         20
 / \       /     \
40  25    10    5
/
15
```

D2-Step 2: Heapify down (swap 30 with larger child 45)

```
      45
   /       \
  30         20
 /  \       /  \
40  25    10    5
/
15
```

D2-Step 3: Continue heapify (swap 30 with larger child 40)

```
      45
   /       \
  40         20
 /  \       /  \
30  25     10    5
/
15
```

Heap after deleting 55

Heap array: [45, 40, 20, 30, 25, 10, 5, 15]

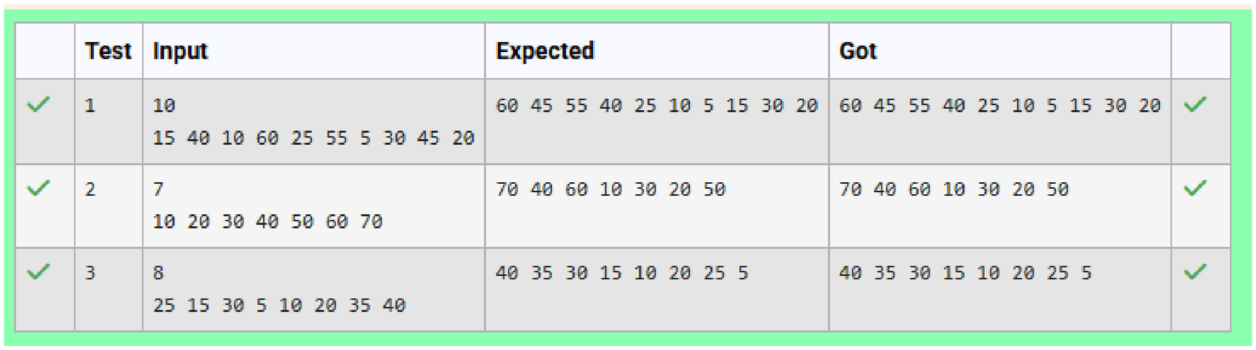
Program

Max-Heap Program:

```
def insert_max_heap(heap, value):
    heap.append(value)
    i = len(heap) - 1

    while i > 0:
        parent = (i - 1) // 2
        if heap[i] > heap[parent]:
            heap[i], heap[parent] = heap[parent], heap[i]
            i = parent
        else:
            break


# -------- Main Program --------
n = int(input())
arr = list(map(int, input().split()))

heap = []
for x in arr[:n]:
    insert_max_heap(heap, x)

print(*heap)
```

Test cases

Max-Heap

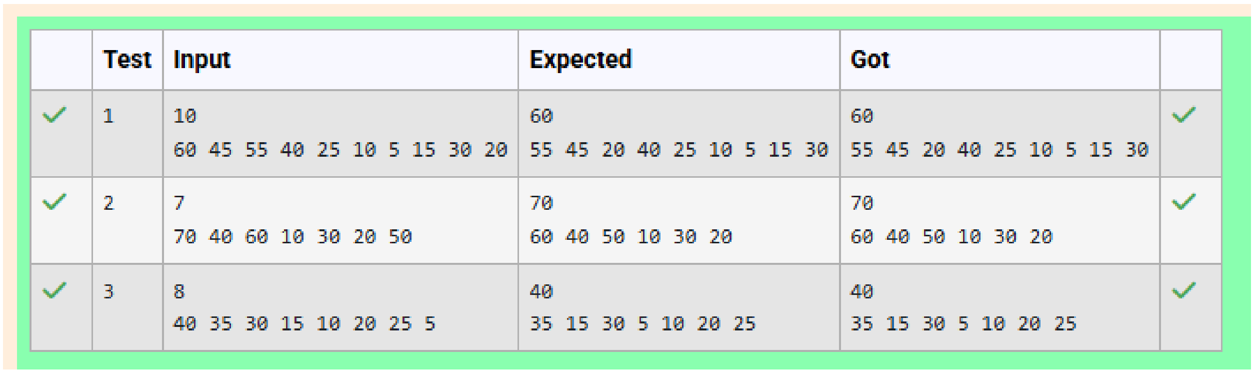


	Test	Input	Expected	Got	
✓	1	10 15 40 10 60 25 55 5 30 45 20	60 45 55 40 25 10 5 15 30 20	60 45 55 40 25 10 5 15 30 20	✓
✓	2	7 10 20 30 40 50 60 70	70 40 60 10 30 20 50	70 40 60 10 30 20 50	✓
✓	3	8 25 15 30 5 10 20 35 40	40 35 30 15 10 20 25 5	40 35 30 15 10 20 25 5	✓

Deletion: Max-Heap

```
def heapify_down(heap, i):
    n = len(heap)
    while True:
        largest = i
        left = 2*i + 1
        right = 2*i + 2

        if left < n and heap[left] > heap[largest]:
            largest = left
        if right < n and heap[right] > heap[largest]:
            largest = right

        if largest != i:
            heap[i], heap[largest] = heap[largest], heap[i]
            i = largest
        else:
            break


def delete_max(heap):
    if not heap:
        return None
    maximum = heap[0]
    heap[0] = heap[-1]
    heap.pop()
    heapify_down(heap, 0)
    return maximum


n = int(input())
heap = list(map(int, input().split()))
deleted = delete_max(heap)
print(deleted)
print(*heap)
```

test cases:



	Test	Input	Expected	Got	
✓	1	10 60 45 55 40 25 10 5 15 30 20	60 55 45 20 40 25 10 5 15 30	60 55 45 20 40 25 10 5 15 30	✓
✓	2	7 70 40 60 10 30 20 50	70 60 40 50 10 30 20	70 60 40 50 10 30 20	✓
✓	3	8 40 35 30 15 10 20 25 5	40 35 15 30 5 10 20 25	40 35 15 30 5 10 20 25	✓

Result

The Priority Queue operations were implemented using a Max Heap and executed successfully. All elements were inserted in the correct order while maintaining the heap property. The Delete-Max operation was also performed correctly on the priority queue. Both insertion and deletion operations were completed without any errors in the LMS.