HEAP – COMPLETE NOTES

1. Definition

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

✔ Max-Heap:

Parent node ≥ children
The **largest element is at the root**.

✔ Min-Heap:

Parent node ≤ children
The **smallest element is at the root**.

2. Properties of Heap

✔ Property 1: Complete Binary Tree

All levels must be filled completely except possibly the last level, which should be filled **from left to right**.

✔ Property 2: Heap Order Property

- **Min heap:** parent ≤ child
- **Max heap:** parent ≥ child

✔ Property 3: Height of Heap

For **n** nodes, height = **O(log n)** (since tree is always complete).

✔ Property 4: Stored in Arrays

Heap does **not need pointers**; stored in array.

Index relations (1-based index):

- Parent(i) = i/2
- Left(i) = 2*i
- Right(i) = 2*i + 1

Index relations (0-based index):

- Parent(i) = (i–1)/2
- Left(i) = 2*i + 1
- Right(i) = 2*i + 2

3. Types of Heaps

1. **Binary Heap** (Min/Max) → most commonly used
2. **Binomial Heap**
3. **Fibonacci Heap**
4. **DAry Heap** (k-ary heap)
5. **Pairing Heap**
6. **Leftist Heap**
7. **Skew Heap**

4. Heap Operations & Time Complexity

Operation	Time
Insert	**O(log n)**
Delete (root)	**O(log n)**
Get Min/Max	**O(1)**
Heapify (down)	**O(log n)**
Build Heap	**O(n)**

5. Creating a Heap (Build Heap)

There are 2 methods:

✔ METHOD 1: Insert one-by-one (O(n log n))

1. Insert element at end
2. Compare with parent
3. **Percolate Up / Bubble Up** until heap property is satisfied.

✔ METHOD 2: Build heap using Heapify (O(n))

Most efficient.

Algorithm:

```
BuildHeap(A):
    for i = n/2 down to 1:
        heapify(A, i)
```

Why from n/2?
Nodes from n/2+1 to n are leaf nodes → already heap.

6. Heapify Algorithms

✔ Max-Heapify

Fixes the heap property by moving element down.

```
MaxHeapify(A, i):
    l = left(i)
    r = right(i)
    largest = i

    if l ≤ n and A[l] > A[largest]:
        largest = l
    if r ≤ n and A[r] > A[largest]:
        largest = r

    if largest != i:
        swap(A[i], A[largest])
        MaxHeapify(A, largest)
```

✔ Min-Heapify

```
MinHeapify(A, i):
    l = left(i)
    r = right(i)
    smallest = i
```

```
    if l ≤ n and A[l] < A[smallest]:
        smallest = l
    if r ≤ n and A[r] < A[smallest]:
        smallest = r
    if smallest != i:
        swap(A[i], A[smallest])
        MinHeapify(A, smallest)
```

7. Example: MAX HEAP CREATION

Given array: **10, 20, 15, 30, 40**

Build heap bottom-up:

Step 1: Start from i = ⌊n/2⌋ = 2

Index:
1:10
2:20
3:15
4:30
5:40

Heapify at index 2:

```
     10
   /     \
  20      15
 /  \
30   40
```

Children of 20 → 30, 40 (largest = 40)
Swap 20 ↔ 40

Now:

```
     10
   /     \
  40      15
 /  \
30   20
```

Step 2: Heapify at index 1

Children: 40, 15 → largest = 40
Swap 10 ↔ 40

Final Max Heap:

```
     40
   /      \
  30       15
 /  \
10   20
```

8. Example: MIN HEAP CREATION

Array: 10, 50, 40, 20, 30

Final Min Heap after heapify:

```
     10
   /      \
  20       40
 /  \
50   30
```

9. Deletion in Heap(Time: O(log n))

Delete always removes the **root** (min/max).

Steps:

1. Replace root with last node.
2. Remove last node.
3. Apply **Heapify Down**.

10. Insertion in Heap

Steps:

1. Add element at **end of array**.
2. **Bubble up** / percolate up to restore heap property.

Time: **O(log n)**

11. Heap Sort

Using Max Heap:

1. Build max heap
2. Swap root with last node
3. Heapify remaining tree
4. Repeat

Time:

- Build heap: **O(n)**
- Each delete: **O(log n)**
  Total: **O(n log n)**

12. Advantage of Heap

1. **Fast get max/min** → O(1)
2. Insert/Delete → O(log n)
3. Efficient priority queue implementation
4. Guaranteed height = log n
5. Used in many key algorithms

13. Disadvantages of Heap

1. Searching an arbitrary element → **O(n)**
2. Not efficient for deletions other than root
3. Not used for sorted traversal (unlike BST)
4. Uses array → fixed-size unless dynamically grown

14. Applications of Heap

✔ 1. Priority Queues

- CPU scheduling
- Dijkstra's algorithm
- Prim's MST algorithm

✔ 2. Heap Sort Algorithm

✔ 3. Job Scheduling

- Process with smallest time / highest priority

✔ 4. Graph Algorithms

- Dijkstra uses Min-Heap
  Prim's algorithm uses Min-Heap

✔ 5. Data Stream Algorithms

- Running median using two heaps
  (max heap for lower half, min heap for upper half)

✔ 6. Bandwidth & Network Management

✔ 7. Event Simulation Systems

- The next event has the highest priority

✔ 8. K Largest / K Smallest Element

- Using a heap efficiently

15. Difference Between Min Heap & Max Heap

Feature	Min Heap	Max Heap
Root	Smallest element	Largest element
Usage	Dijkstra, priority queue	Heap sort, priority queue
Comparison	parent ≤ children	parent ≥ children
Extract	extractMin()	extractMax()

16. Difference Between Heap & BST

Feature	Heap (Binary Heap)	BST
Shape	Complete binary tree	No shape restriction
Order	Only parent-child rule	Left < Root < Right
Search	O(n)	O(log n)
Insert/Delete	O(log n)	O(log n)
Traversal	No sorted output	Inorder gives sorted
Use	Priority queue	Searching/Sorting

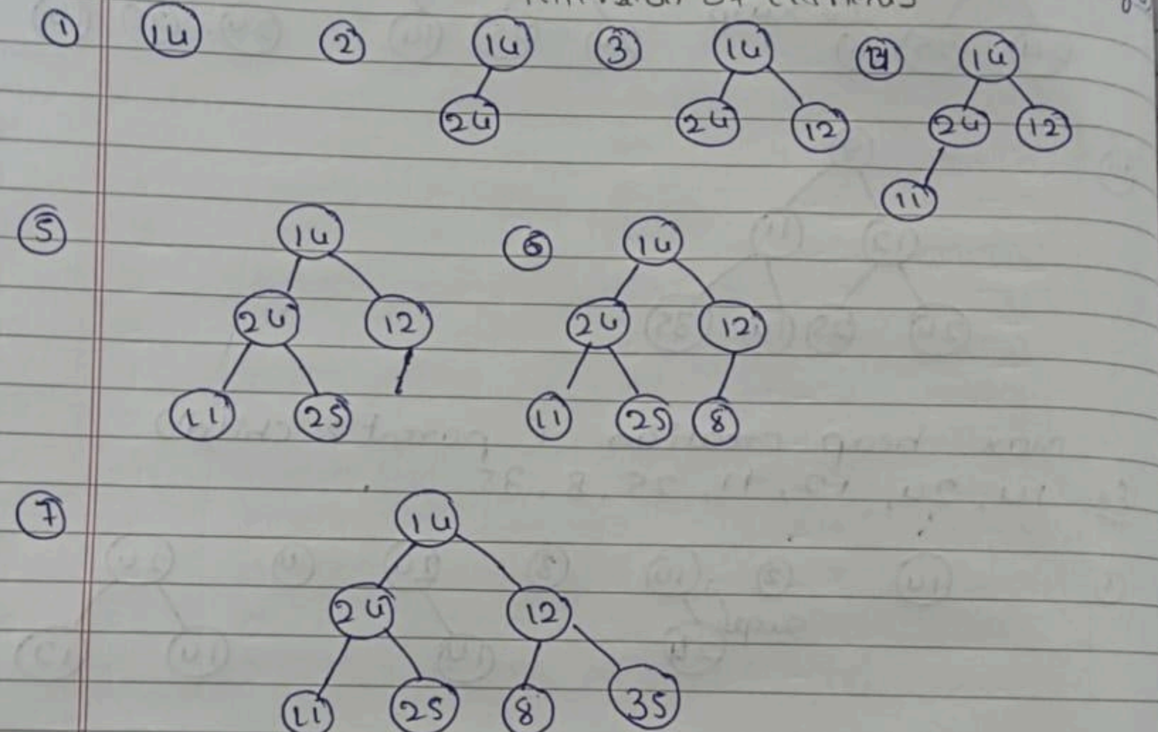
17. Interview Questions

1. What is a heap?
2. Difference between min heap & max heap?
3. Why heapify bottom-up is O(n)?
4. Why are heaps always complete binary trees?
5. Applications of a heap?
6. Build a heap from an array example.
7. Difference between a heap and BST?
8. Explain heap sort.
9. Extract-min / extract-max algorithm.
10. Why is searching slow in the heap?

Heap

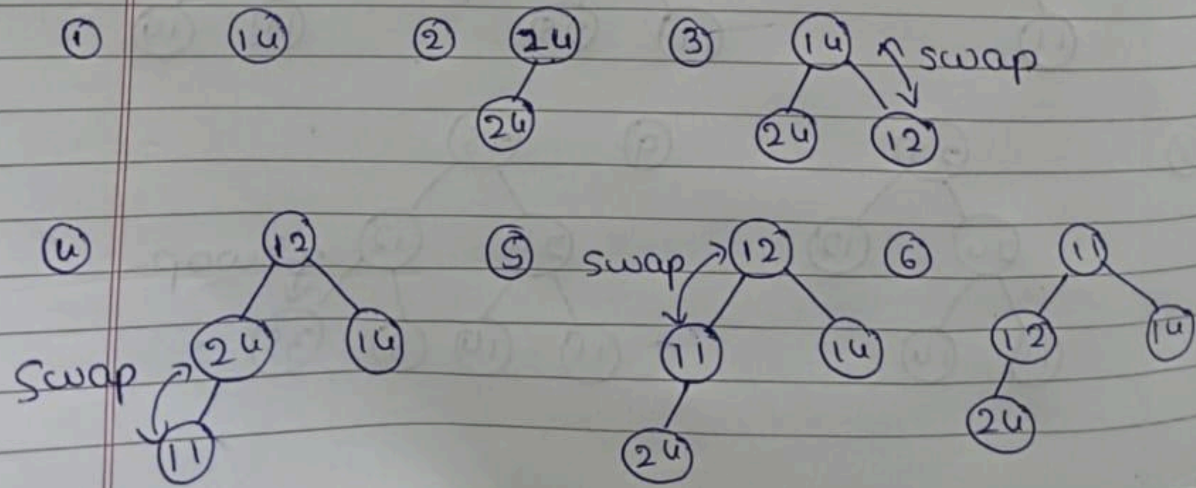
Ex 14, 24, 12, 11, 25, 8, 35

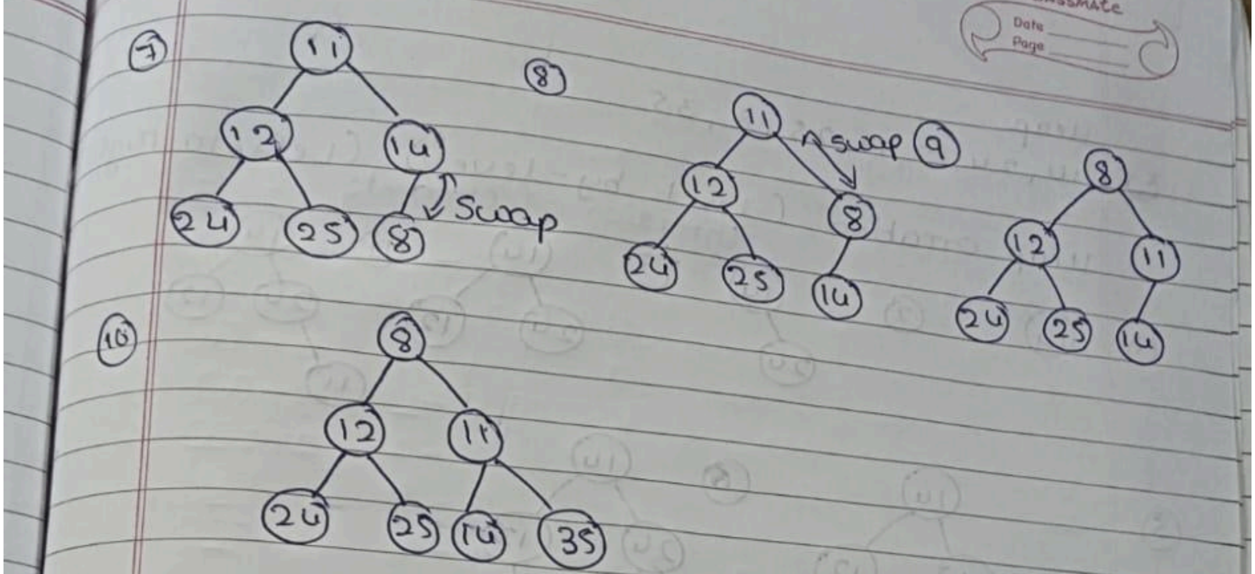
Heap creation (level-by-level) (Left to Right) interstion of elements



Min Heap Creation (parent < child)

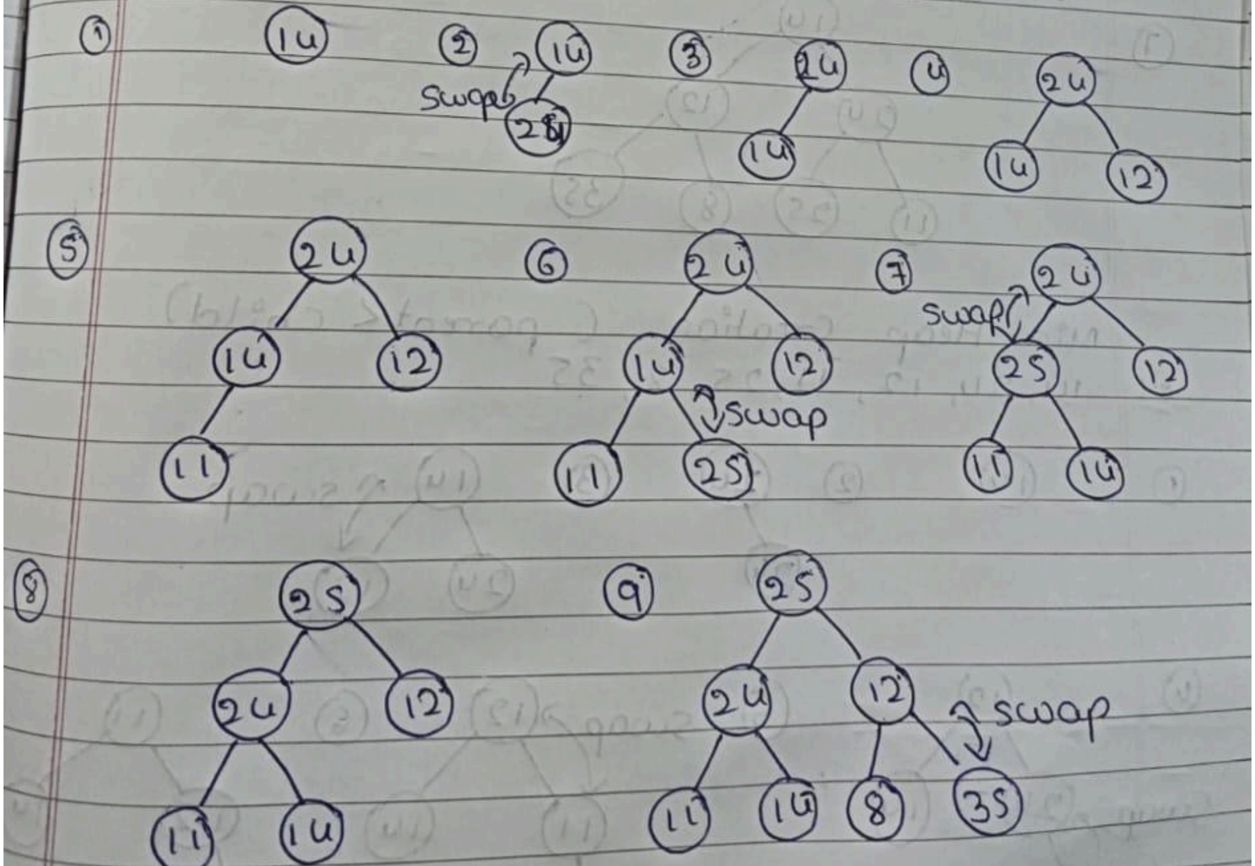
Ex 14, 24, 12, 11, 25, 8, 35

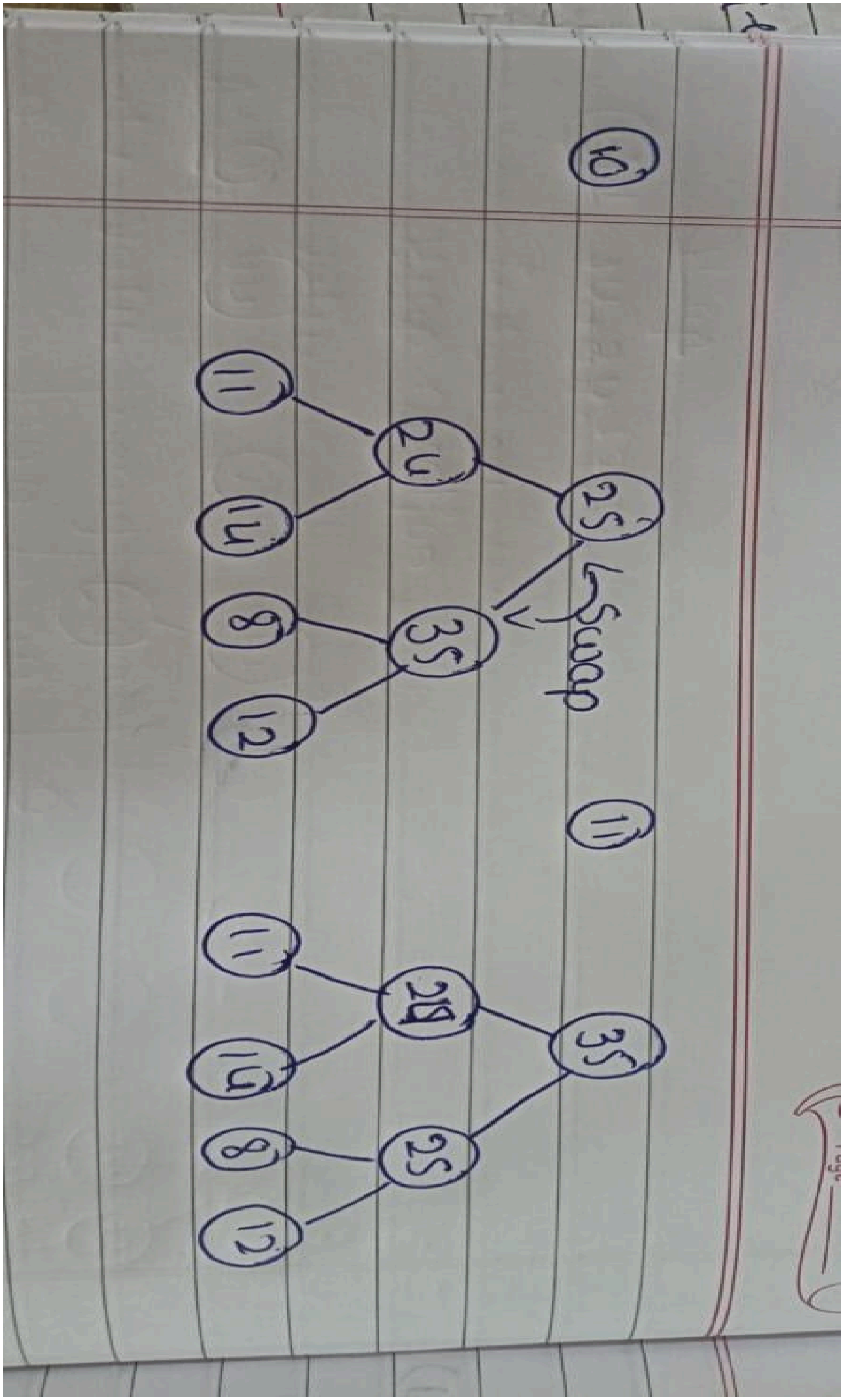




Max heap creation (parent > child)

Ex 14, 24, 12, 11, 25, 8, 35




10
25
24
35
11
14
8
12
swap
11
35
25
11
14
8
12