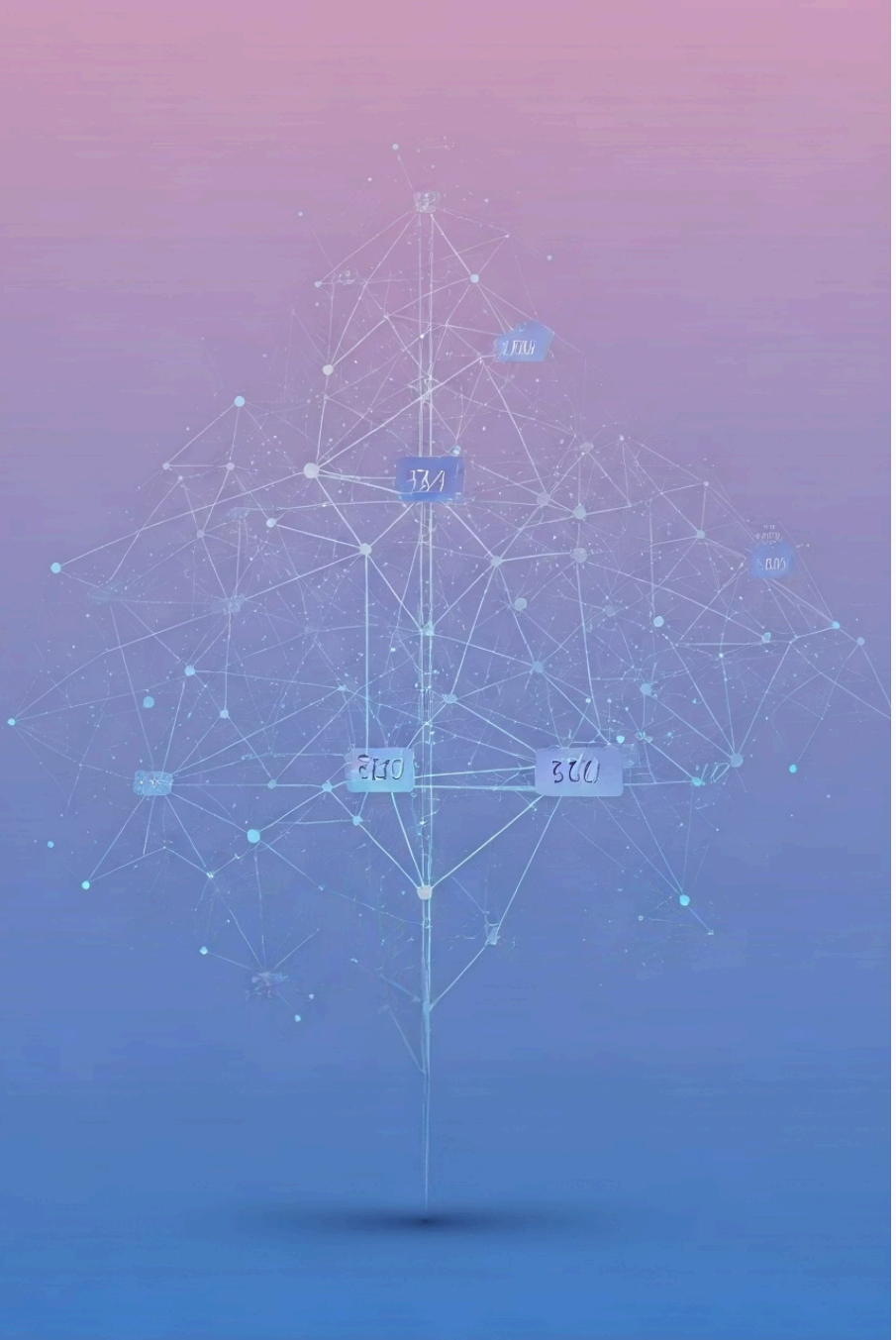


The Heap Abstract Data Type (ADT)

Today, we'll explore the Heap Abstract Data Type. It's a specialized tree-based data structure. Heaps efficiently support priority queue operations. They maintain specific shape and order properties.

Heap Properties and Applications

Shape: Complete Binary Tree

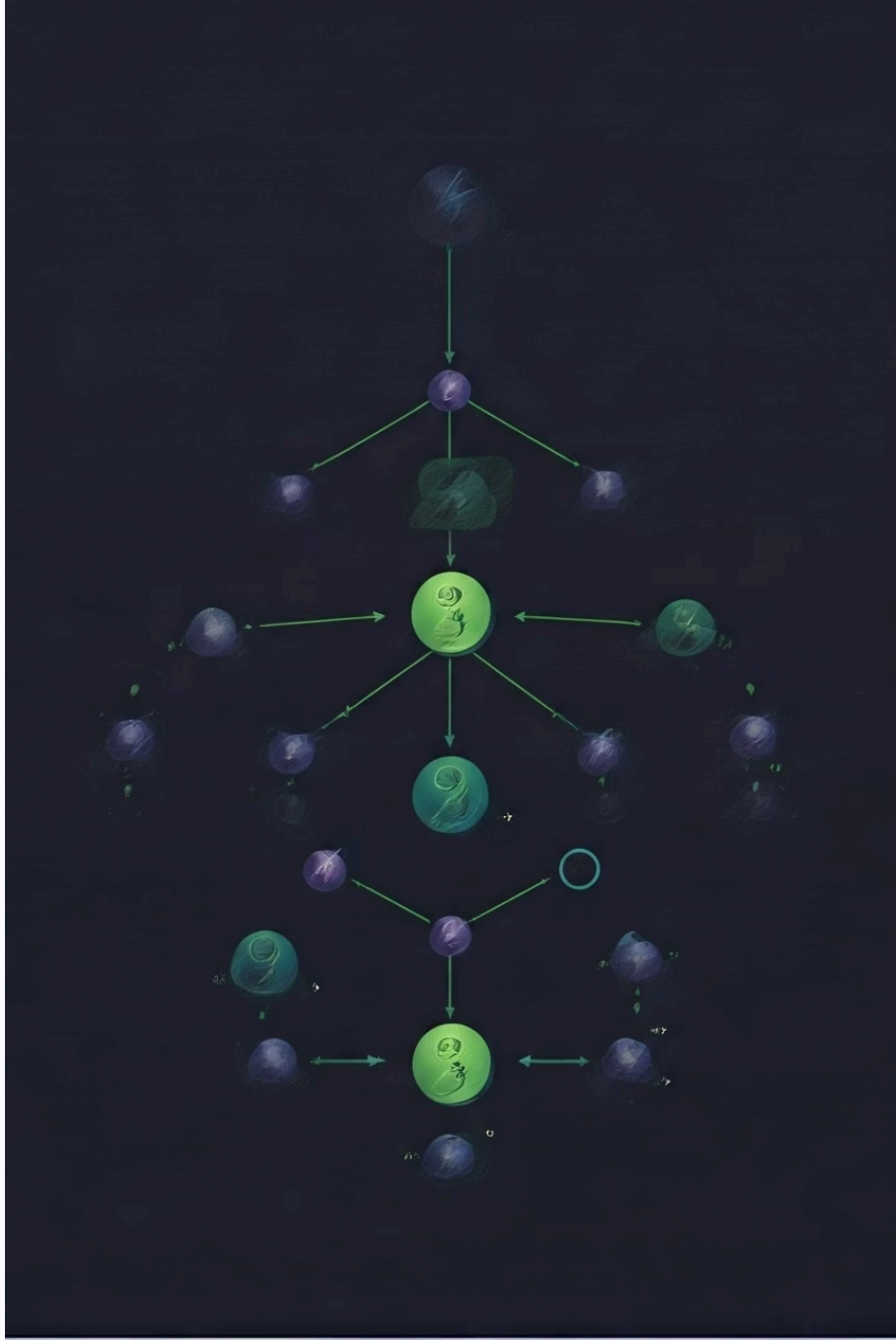
All levels are filled, and the last level is filled from left to right.

Order: Heap-Order Property

Every parent node is less than or equal to its children (for a min-heap).

Key Applications

Heaps are vital for priority queues, Dijkstra's algorithm, and Heap Sort.



Implementing Heaps with Complete Binary Trees

Complete Binary Tree

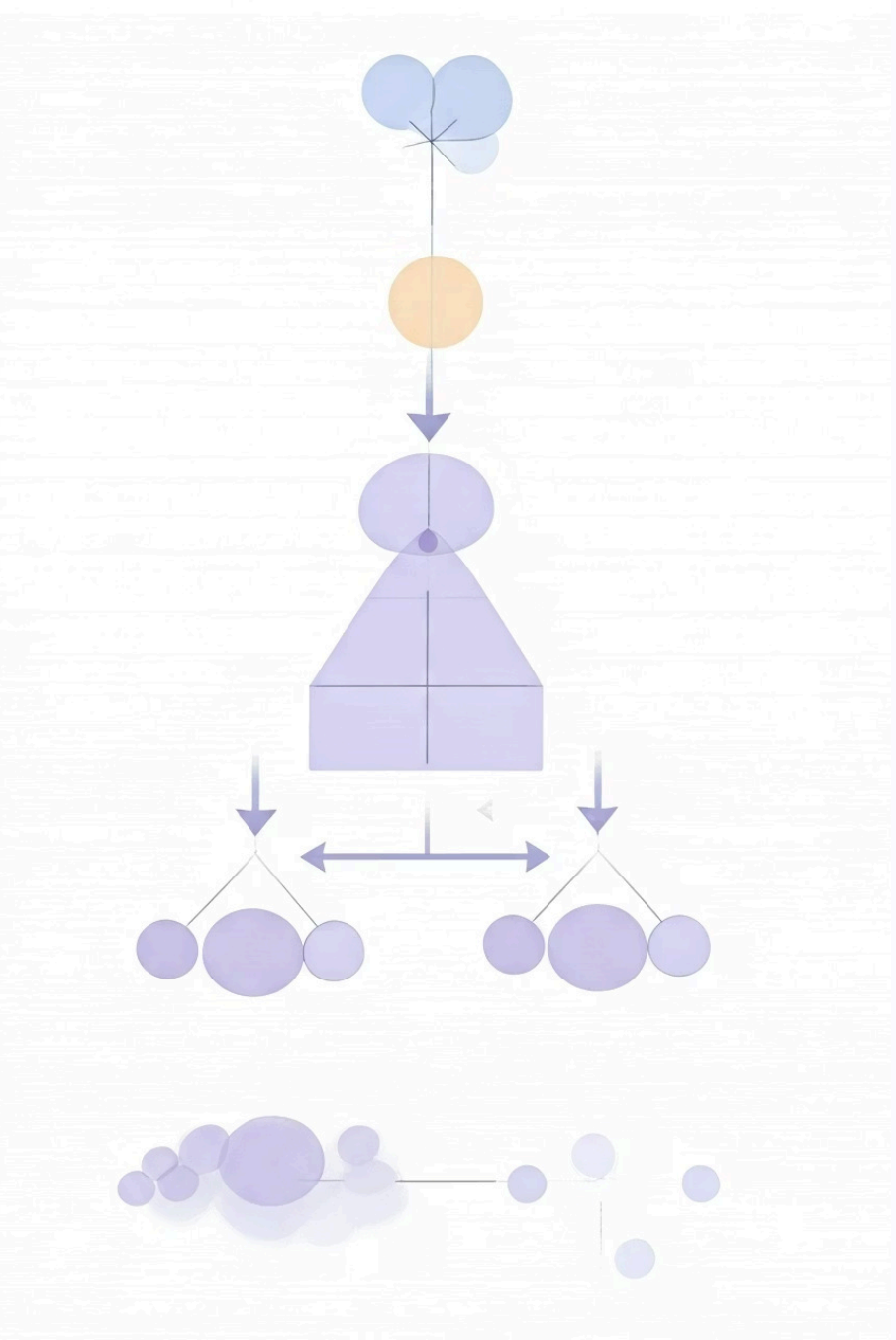
Every level must be fully filled. The bottom level fills strictly from left to right.

Efficient Operations

This structure supports very efficient insert and delete operations. This is crucial for performance.

Balanced Structure

Maintaining a balanced tree ensures O(log n) performance. This means operations are fast.



Array Representation of Complete Binary Trees

1

Heap in Array

The entire heap is stored efficiently within a simple array. The root is at index 1.

2

Child Indexing

For a node at index 'p', its left child is at '2p'. The right child is at '2p+1'.

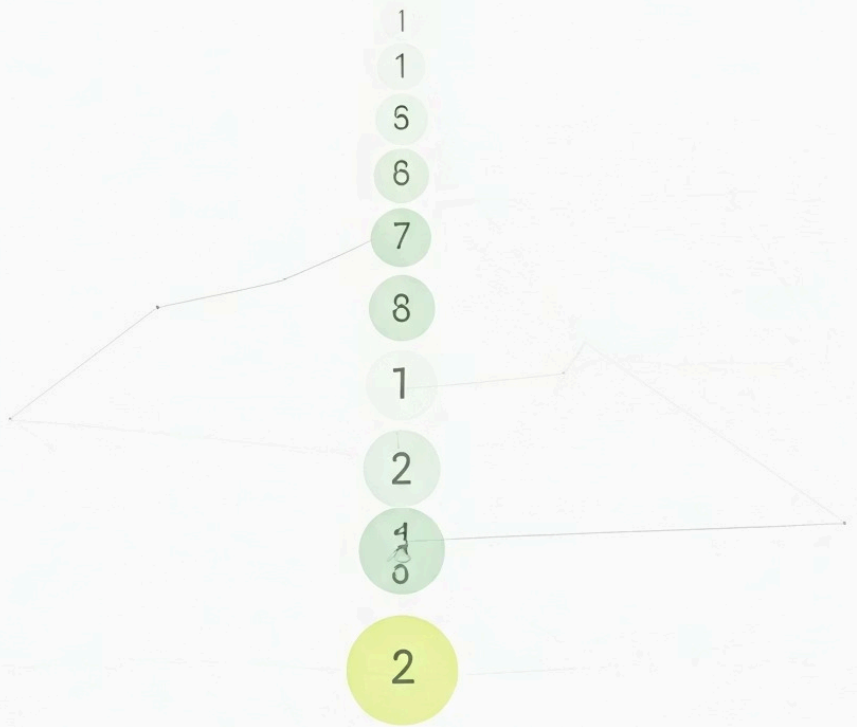
3

Parent Indexing

Conversely, the parent of a node at index 'n' is found at 'n//2'. This simplifies navigation.

Heap Order Property

The Heap Order Property is fundamental. For any node 'x' and its parent 'p', the key of the parent must be less than or equal to the key of the child. This is for a min-heap. This property guarantees the smallest element is always at the root. It simplifies finding the minimum value instantly. This consistency is crucial for heap operations.



Inserting into a Heap

Add New Node

Insert the new item into the leftmost available spot in the last level of the tree. This is usually the end of the array.

Percolate Up

The newly inserted node "percolates up." It swaps with its parent if it's smaller, until the heap property is restored.

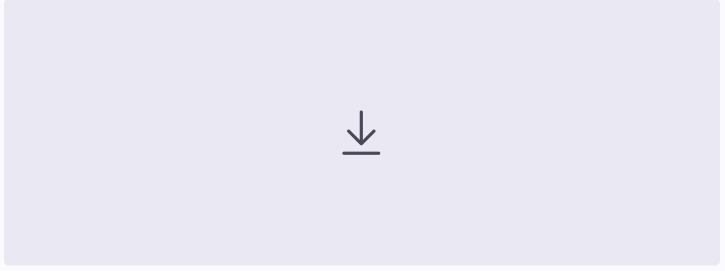
O(log n) Time

This process takes O(log n) time. The number of swaps depends on the height of the tree.

Delete-Min Operation in a Heap

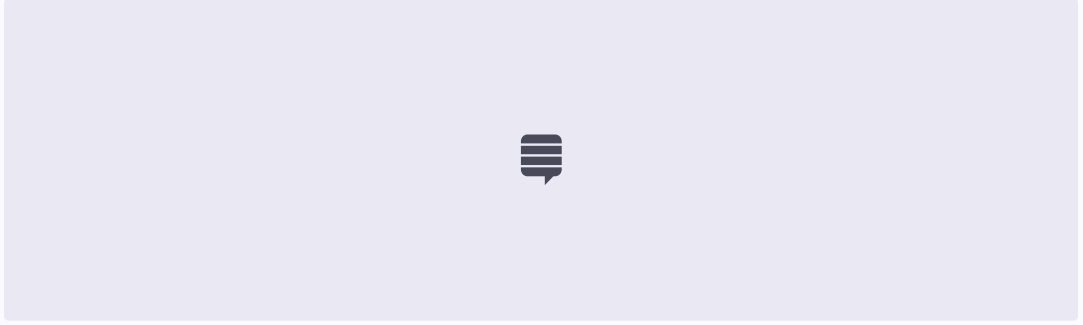
Remove Root

The smallest item, located at the root, is removed. This leaves a vacant spot.



Replace and Percolate

The very last node in the heap is moved to the root's position. It then "percolates down."



Restore Heap Order

The new root continuously swaps with its smallest child until the heap property is satisfied. The tree structure remains complete.

Building a Heap from Unordered Items

Successive Insertions

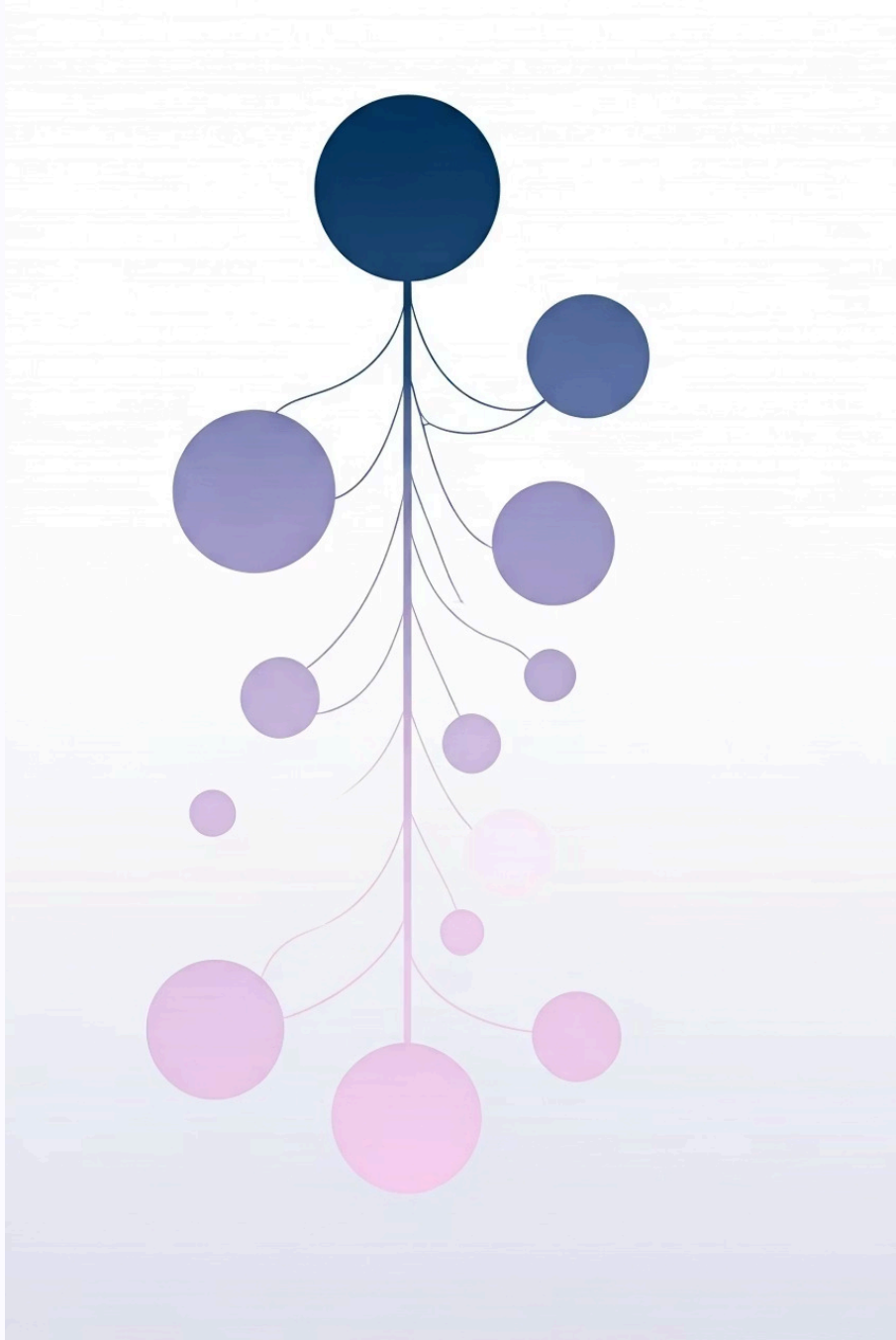
One method is to insert each item one by one. This approach takes O(n log n) time.

Heapify (Bottom-Up)

A more efficient method is "heapify." It starts from the last non-leaf node. Each node is percolated down to its correct position.

Proof: Build-Heap is Linear Time

The build-heap operation is surprisingly efficient. The number of operations needed for each node decreases significantly as you approach the leaves. This is represented by the formula: $\sum_{i=1}^{\log n} \frac{n}{2^i} * i = O(n)$. Most nodes are located near the bottom of the tree, requiring very few swaps. Therefore, the total cost for building the entire heap remains linear, or O(n). This makes heap construction very fast.



Conclusion and Key Takeaways

1

Efficient Priority Management

The Heap ADT excels at managing dynamic sets based on priorities.

2

Fast Operations

The complete binary tree structure enables rapid insertions and deletions.

3

Linear Build Time

Building a heap from scratch is provably an O(n) operation.

4

Versatile Applications

Heaps form the backbone of heapsort, efficient priority queues, and various graph algorithms.