**DSAD – Heap Structures**

1. **How is a Heap used to implement a Priority- Queue?**

* A heap is a common data structure used to implement a priority queue due to its efficient support for the essential priority queue operations, particularly insertion and removal of the highest or lowest priority element. Here's how it works:
  + **Properties of a Heap**

A heap is a specialized binary tree that satisfies two main properties:

1. **Complete Binary Tree:** All levels are fully filled except possibly the last, which is filled from left to right.

Heap Property:

1. **Max-Heap:** Every parent node is greater than or equal to its children, meaning the maximum element is at the root.
2. **Min-Heap:** Every parent node is less than or equal to its children, meaning the minimum element is at the root.

Priority queues can be implemented using either a max-heap or min-heap, depending on whether you want the highest or lowest priority element to be accessible at all times.

* + **Operations in a Priority Queue Using a Heap**

1. **Insertion:** To insert an element, it is added at the end of the heap (preserving the complete binary tree property) and then "bubbled up" or "heapified up" to maintain the heap property. This operation takes \(O(\log n)\) time.
2. **Find Highest/Lowest Priority:** In a max-heap or min-heap, the root node (top of the heap) always contains the highest or lowest priority element, respectively. Accessing this element is an (O(1)) operation.
3. **Remove Highest/Lowest Priority (Extract Max/Min):** The root element is removed, and the last element in the heap is moved to the root. The heap then undergoes a "bubble down" or "heapify down" operation to restore the heap property. This operation takes (O(log n)) time.
   * **Efficiency**

Using a heap for a priority queue is efficient because:

1. Both insertion and deletion (extract max or min) are (O(log n)) operations due to the need to maintain the heap property.
2. Accessing the highest or lowest priority element is (O(1)) because it's always at the root.
   * **Implementation**

In practice, a binary heap is often implemented as an array.

For example, the children of a node at index (i) are located at indices (2i + 1) and (2i + 2), while the parent is located at frac(i – 1)/2. This array-based structure is both memory-efficient and faster to access.

1. **How can a Heap be utilised in Dijkstra’s Algorithm for shortest path finding?**

* In Dijkstra’s Algorithm for finding the shortest path in a weighted graph, a min-heap (or priority queue) is used to efficiently select the next vertex with the smallest tentative distance. Here’s a breakdown of how a heap is utilized within Dijkstra's algorithm:
  + **Purpose of the Heap in Dijkstra’s Algorithm**

The heap (min-heap Priority queue) allows us to:

1. Efficiently retrieve the node with the smallest tentative distance, which is crucial since Dijkstra’s algorithm continuously selects the “closest” unvisited vertex.
2. Update distances for neighbouring vertices and reorder them in the heap as needed.

Using the min-heap provides a faster way to manage and access nodes by their distance, as it enables both:

1. Fast insertion (when new distances are calculated for neighbouring nodes)
2. Fast extraction of the node with the smallest distance (i.e., the node to be processed next)
   * **Steps in Dijkstra’s Algorithm using a min-heap**
3. **Initialize Distances and Heap:**
4. Set the distance to the source node as 000 and all other nodes as ∞ (indicating they are not yet reachable).
5. Insert all nodes into the min-heap, ordered by their current distances.
6. **Process Nodes in the Min-Heap:**
   1. Extract the node with the smallest distance from the heap. This is the next node to be processed, as it has the shortest known path from the source.
   2. For each neighbour of this node, calculate the tentative distance (current distance + edge weight).
   3. If the tentative distance to a neighbouring node is smaller than its current distance:
      1. Update the neighbour's distance.
      2. Decrease the key (priority) of this neighbour in the min-heap to reflect the updated shorter distance.
7. **Repeat until the Heap is Empty:**
   1. Continue extracting the minimum distance node and updating its neighbours until the heap is empty, meaning all reachable nodes have been processed.
   * **Efficiency of the Dijkstra’s Algorithm**

Using a min-heap significantly reduces the time complexity of Dijkstra's algorithm compared to a simple list or array. The operations involved are:

1. **Extracting the minimum distance node**: O(log V) time (where V is the number of vertices).
2. **Updating distances (decrease key)**: O(log V) time for each edge relaxation.

The overall time complexity of Dijkstra’s algorithm with a min-heap (implemented as a binary heap) is O((V+E)log V), where V is the number of vertices and E is the number of edges. This is much more efficient than using unstructured data structures like a simple array, where finding the minimum would take O(V) time per extraction.

1. **What role does a heap play in the Heap Sort Algorithm?**

* In the Heap Sort algorithm, a heap is used to efficiently sort elements by leveraging the heap’s properties to repeatedly extract the maximum (or minimum) element and place it in the correct sorted position. Heap Sort is typically implemented using a max-heap to produce a sorted array in ascending order.
  + **Steps of Heap Sort Using a Heap**

1. **Building the Max-Heap:**
2. Convert the input array into a max-heap. In a max-heap, each parent node is greater than or equal to its children, ensuring that the maximum element is at the root (index 0).
3. This can be achieved by performing a series of "heapify" operations from the middle of the array to the beginning, which ensures each subtree satisfies the max-heap property.
4. **Sorting Process:**
5. Once the max-heap is built, the sorting phase begins by repeatedly extracting the largest element (the root of the heap) and moving it to the end of the array.
6. After moving the maximum element to its correct position, reduce the size of the heap by one (ignoring the sorted portion at the end) and restore the max-heap property on the remaining elements.
7. Repeat this extraction and heapify process until all elements are sorted.

* **Example of Heap Sort Steps**

1. **Initial Max-Heap Construction:** Convert the unsorted array into a max-heap structure so that the maximum element is at the root.
2. **Heap Extraction and Re-Heapifying:** 
   1. Swap the root element (the maximum) with the last element of the heap.
   2. Exclude the last element from the heap, reducing its size.
   3. Re-heapify the root to maintain the max-heap property.
   4. Repeat until all elements are sorted in ascending order.

* **Role of the Heap in Heap Sort**
  1. **Organizing Elements for Sorting:** The heap structure is used to keep the largest element at the root, facilitating the extraction of the next largest element with each iteration.
  2. **Efficient Selection of Maximum Elements:** By maintaining the max-heap property, Heap Sort ensures (O(log n)) time complexity for re-heapifying after each extraction.
  3. **In-Place Sorting:** Heap Sort does not require additional storage, as the elements are sorted within the array itself, making it memory efficient.

1. **How does a heap support the efficient implementation of a median finding algorithm?**
2. **How are heaps used in the process of merging k sorted arrays or lists?**
3. **What is the difference between using a min-heap and a max-heap for job scheduling problems?**
4. **How can a heap be used to find the kth largest (or smallest) elements in an unsorted array?**
5. **How does a heap help in maintaining the top k elements in a stream of data?**
6. **How can a heap be used to solve the problem of finding the kth smallest element in a matrix sorted by rows and columns?**
7. **How can heaps be used in graph algorithms like Prim’s Minimum Spanning Tree Algorithm?**