Heap - I

What is Heap Data Structure ?

A heap is a binary tree-based data structure that satisfies the heap property.

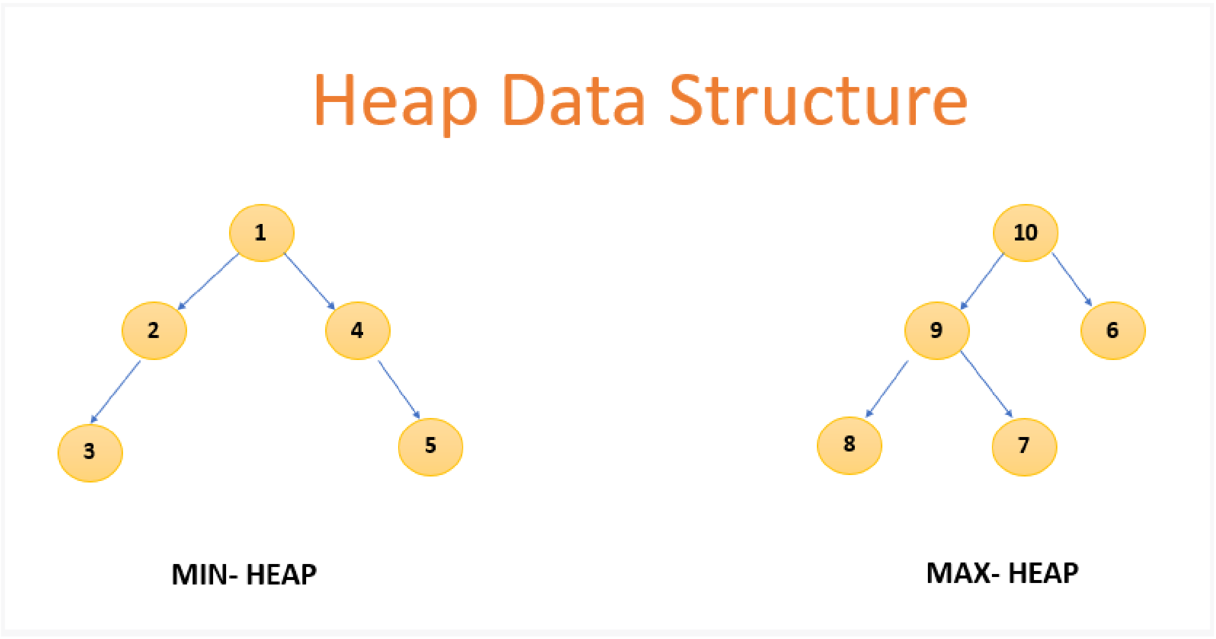
Properties of Heap:

- **Heap Property:** The main property of a heap is that it satisfies the "heap property." In a min-heap, for any node other than the root, the value of that node is greater than or equal to the values of its children. In a max-heap, the value of each node is less than or equal to the values of its children. This ensures that the minimum (or maximum) element is always at the root.
- **Complete Binary Tree:** Heaps are typically implemented using a complete binary tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. The heap property combined with the complete binary tree structure ensures that heaps are balanced.
- **Efficient Insertion and Extraction:** The heap property allows for efficient insertion and extraction of elements. Inserting a new element takes O(log n) time, where n is the number of elements in the heap. Extracting the minimum (or maximum) element also takes O(log n) time, ensuring that the root can be quickly accessed.
- **Heap Sort:** The heap property enables heaps to be used as the basis for an efficient sorting algorithm called "Heap Sort." Heap Sort has a time complexity of O(n log n) and can be used for in-place sorting.
- **Priority Queue:** Heaps are often used to implement priority queues, where elements with the highest (or lowest) priority can be efficiently retrieved and removed.
- **Dijkstra's Algorithm:** Heaps are used in Dijkstra's algorithm to find the shortest path in a weighted graph efficiently.
- **Memory Management:** Heaps are used in memory allocation systems to manage dynamic memory allocation and deallocation.

Types of heaps :

1. **Min-Heap:** In a min-heap, the value of each node is greater than or equal to the values of its children. This means that the smallest element is always at the root, and each node is smaller than its children.

2. **Max-Heap** In a max-heap, the value of each node is less than or equal to the values of its children. This means that the largest element is always at the root, and each node is greater than its children.



Operations Supported by Heap:

- **Insert:** Inserting a new element into the heap while maintaining the heap property. The new element is usually added at the end of the heap and then "bubbled up" or "percolated up" to its correct position to satisfy the heap property.
- **Extract:** Extracting the minimum (in a min-heap) or maximum (in a max-heap) element from the heap. After extraction, the heap property is restored by moving the last element to the root and then "bubbling down" or "percolating down" the element to its correct position.
- **Peek/Top:** Viewing the minimum (in a min-heap) or maximum (in a max-heap) element without removing it from the heap. This operation is usually used to access the highest (or lowest) priority element without modifying the heap's structure.

- **Heapify**: Converting an array into a valid heap by reordering its elements to satisfy the heap property. This operation is used to create a heap from an unsorted array efficiently and is commonly used as a step in heap sort.
- **Change Key:** Modifying the value of a particular element in the heap and adjusting its position to maintain the heap property. This operation is essential for updating priorities in a priority queue.
- **Merge:** Merging two heaps into a single heap. This operation is useful when combining two priority queues or merging sorted arrays into a single sorted array.

Implementation of Heap Data Structure:-

MaxHeapify function is to maintain the Max Heap property of a binary tree represented as an array, specifically when an element's value is potentially violating the Max Heap property. The Max Heap property states that each node in the heap must have a value greater than or equal to the values of its child nodes.

Here's a step-by-step explanation of the maxHeapify function:

- Given an array, arr[], that represents a complete binary tree, and an index 'i' representing the current element whose subtree needs to be adjusted to maintain the Max Heap property.
- Initially, we consider the current element at index 'i' as the 'MAXIMUM' within its subtree.
- We then compare the value of the left child of the current element, located at index 2*i+1, with the 'MAXIMUM.' If the left child's value is greater than the 'MAXIMUM,' we update the 'MAXIMUM' to be the value of the left child.
- Similarly, we compare the value of the right child of the current element, located at index 2*i+2, with the 'MAXIMUM.' If the right child's value is greater than the 'MAXIMUM,' we update the 'MAXIMUM' to be the value of the right child.
- After comparing the current element's value with its left and right children, if the 'MAXIMUM' is still the current element itself (i.e., neither left nor right child is greater than the current element), then the Max Heap property is already satisfied, and no further action is needed. We stop the process.
- However, if the 'MAXIMUM' has been updated to one of the children's values, it means that the current element's value is violating the Max Heap property. To fix this, we swap the current element with the element holding the 'MAXIMUM' value, i.e., either its left or right child.

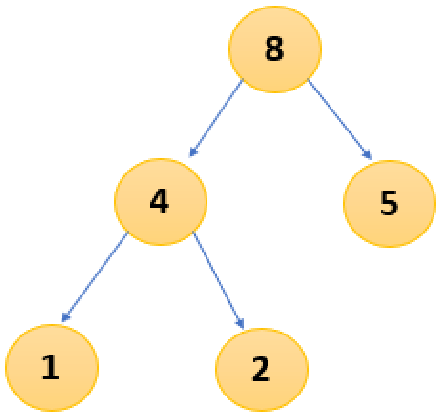
- After the swap, the current element has moved down in the tree to the position of the child that previously had the 'MAXIMUM' value. At this point, we need to check if the subtree rooted at the original child (which now holds the current element's value) also satisfies the Max Heap property. To do this, we recursively apply the maxHeapify function on the child's subtree, treating the child as the new 'i' index.
- We repeat steps 2 to 7 until the Max Heap property is restored throughout the affected subtree.

Insertion

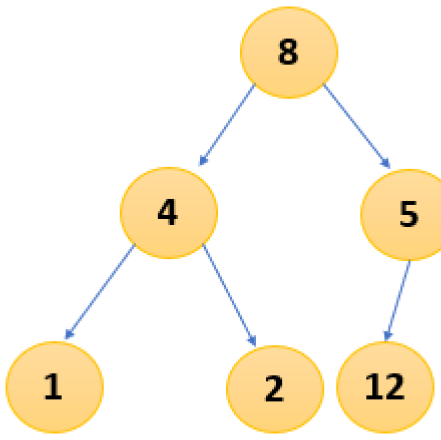
When a new element is inserted into the heap, its addition can disrupt the heap's properties. To preserve the heap's properties, we must perform the heapify operation on the heap

For example -

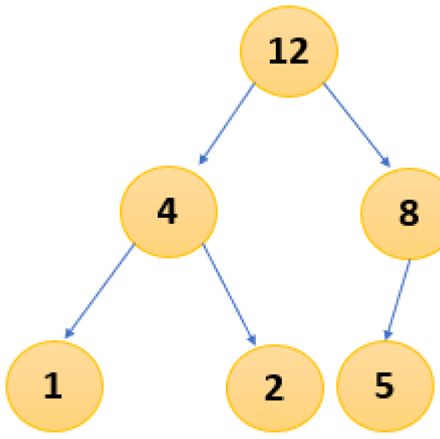
Step -1 : Given a Max- heap tree.



Step -2 : Insert "12" to the max- heap .



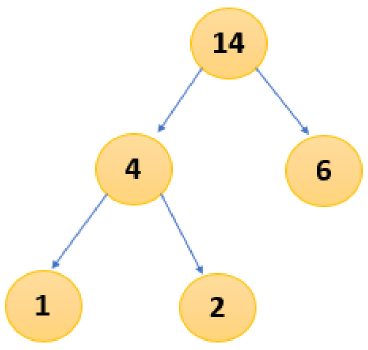
Step - 3 : After heapfiy operation final heap will be



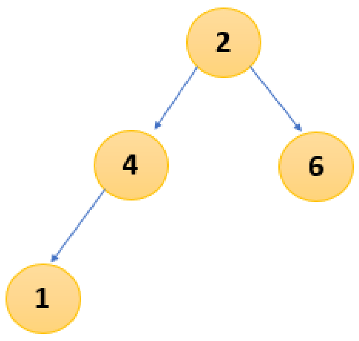
Deletion:

If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operations so that it maintains the property of the heap.

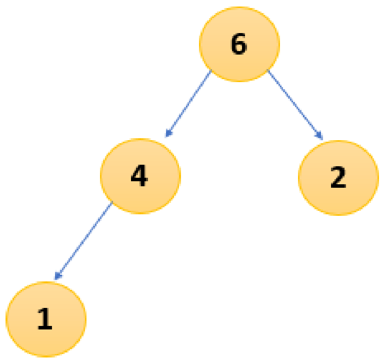
Step - 1 : Given a Max- heap tree.



Step - 2 : Delete “14” root element of the heap.



Step - 3 : after heapify final heap is



Applications of Heap Data Structure:

- Priority Queues: One of the primary applications of the Heap is in implementing Priority Queues. A Priority Queue is a data structure that allows elements with the highest priority to be served before elements with lower priority. Heaps, especially binary heaps, are commonly used to implement Priority Queues efficiently, as they provide O(log n) time complexity for both insertion and extraction of the maximum (or minimum) element.
- Dijkstra's Shortest Path Algorithm: Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph. It utilises a Priority Queue (often implemented using a Heap) to efficiently select the next vertex to visit based on the minimum distance known so far.
- Heap Sort: Heap Sort is a comparison-based sorting algorithm that uses the Heap data structure to sort elements in ascending or descending order. It is an in-place sorting algorithm with a time complexity of O(n log n) and is often used when a stable sorting algorithm is not required.
- Memory Management: In operating systems, Heaps are used for memory management. Dynamic memory allocation functions like malloc() and free() in C/C++ use the Heap to allocate and deallocate memory blocks.
- Huffman Coding: Huffman coding is a lossless data compression algorithm widely used in data compression applications. Heaps are used in constructing Huffman trees, which are then used to assign variable-length codes to characters, with shorter codes for more frequent characters and longer codes for less frequent characters, thus achieving efficient data compression.
- Event-driven Simulation: In event-driven simulations, Heaps are used to efficiently manage the events' ordering based on their timestamps. This allows the simulation to process events in chronological order efficiently.
- Merge Sorted Files: When merging multiple sorted files or streams, Heaps can be employed to efficiently select the next smallest element among all the files or streams, resulting in an overall efficient merge process.

Advantages of Heaps:

- Efficient Insertion and Extraction: Heaps offer efficient insertion and extraction of the maximum (or minimum) element, typically in O(log n) time complexity. This property makes heaps suitable for applications like Priority Queues and Dijkstra's Shortest Path Algorithm.

- Space Efficiency: Heaps can be implemented as arrays, requiring only a small constant overhead compared to linked structures. This makes them space-efficient, especially for applications with large datasets.
- Guaranteed Maximum/Minimum: Depending on whether it's a Max Heap or Min Heap, Heaps guarantee that the root node holds the maximum or minimum value, respectively. This property is useful in various scenarios, such as finding the largest or smallest element quickly.
- Sorting: Heap Sort, a sorting algorithm based on heaps, has a time complexity of O(n log n), and it sorts elements in place, making it useful when a stable sorting algorithm is not a priority.
- Memory Management: Heaps are used in memory management systems to allocate and deallocate dynamic memory blocks efficiently.
- Priority Queues: Heaps are a natural fit for implementing Priority Queues, allowing for efficient access and manipulation of elements with varying priorities.

Disadvantages of Heaps:

- Lack of Random Access: Unlike arrays, Heaps do not provide efficient random access to elements. Accessing elements in the middle of the heap requires traversing the heap from the root, resulting in a time complexity of O(n).
- Not Suitable for Dynamic Resizing: Heaps are typically implemented as arrays with a fixed size. While it is possible to create a new larger heap and copy elements to it when the original heap is full, dynamic resizing can be costly.
- Slower Search: Searching for a specific element in a Heap is not efficient. It may require traversing the entire heap, leading to a linear time complexity of O(n).
- Additional Overhead: The process of maintaining the heap property during insertions and extractions may involve swapping elements, leading to some additional overhead.
- Inefficient for Small Lists: For very small lists or datasets, the overhead of maintaining the heap property may outweigh the benefits, making other data structures more suitable.
- Complex Implementation: While the basic operations on heaps are straightforward, implementing more advanced operations or specialised heap variants (e.g., Fibonacci Heap) can be complex.