Heap Data Structure

What is a Heap?

A **Heap** is a special type of binary tree that satisfies two main properties:

1. **Complete Binary Tree**: It is a complete binary tree, meaning that all levels of the tree are fully filled except possibly for the last level, which is filled from left to right.
2. **Heap Property**: In a heap, each node must satisfy the heap property, which can be either a Max-Heap or a Min-Heap:
   - **Max-Heap**: The value of each node is greater than or equal to the values of its children, with the maximum value at the root.
   - **Min-Heap**: The value of each node is less than or equal to the values of its children, with the minimum value at the root.

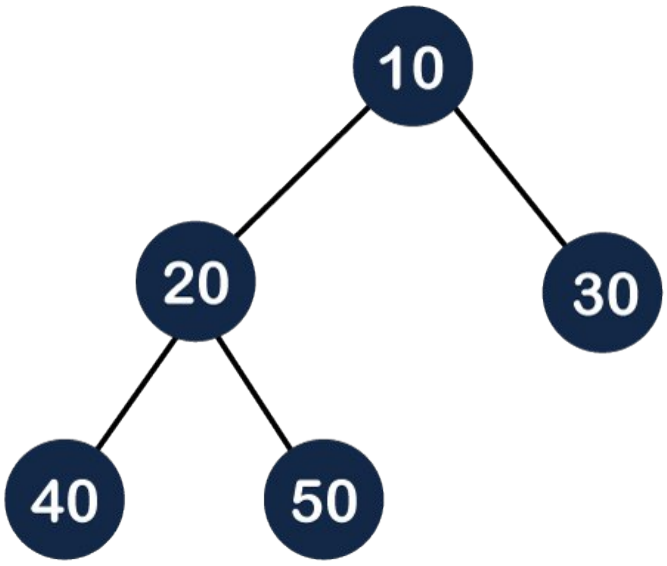
Before diving deeper into heaps, it's essential to understand the concept of a complete binary tree.

What is a Complete Binary Tree?

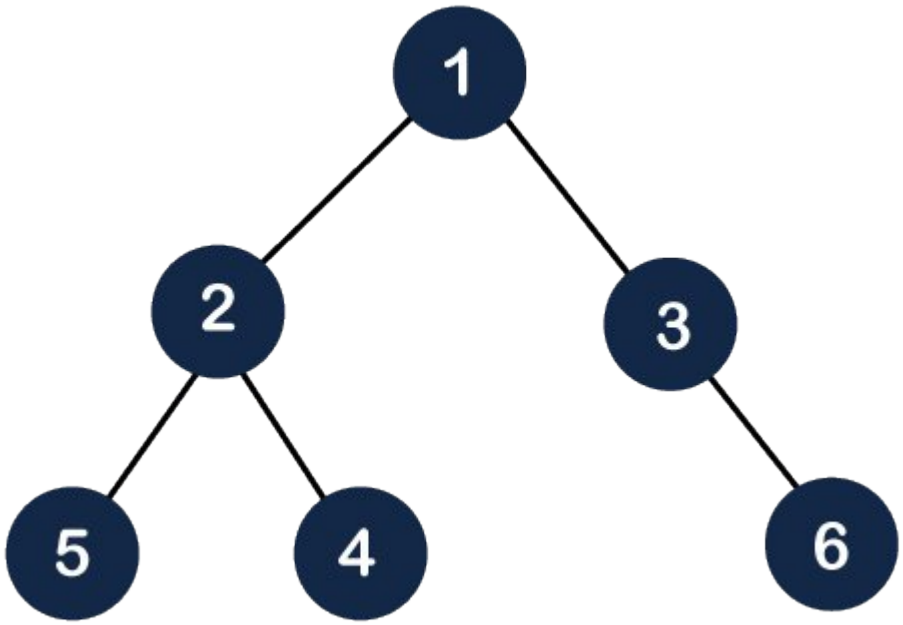
A **Complete Binary Tree** is a type of binary tree where:

- All levels are fully filled, except possibly the last level.
- The last level (leaf nodes) is filled from left to right.

In other words, in a complete binary tree, every level, except possibly the last, has the maximum number of nodes, and all nodes at the last level are positioned as far left as possible. This structure is crucial for implementing heaps efficiently.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Types of Heaps

Heaps are categorized into two types based on how they maintain their internal structure:

1. **Min Heap**
2. **Max Heap**

Min Heap

In a **Min Heap**, the value of each parent node is less than or equal to the value of its children. This ensures that the smallest element is always at the root of the tree.

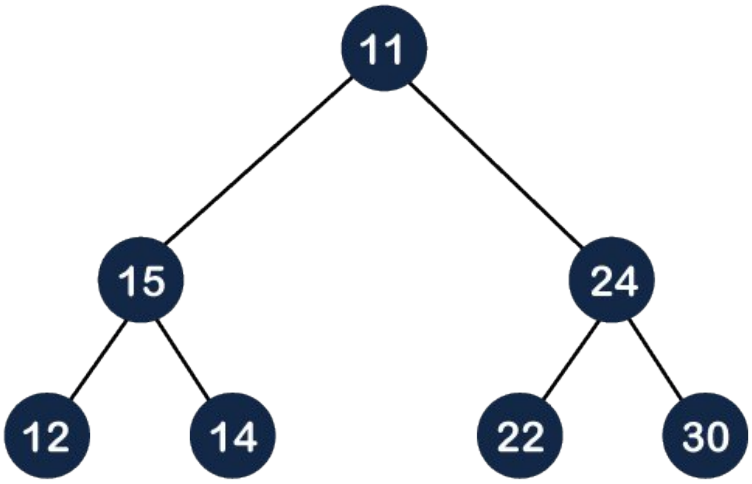
Key Property:

- For every node `i`, the value of node `i` is greater than or equal to the value of its parent node, except for the root node, which has no parent.

Mathematically, this property can be expressed as: $A[Parent(i)] \leq A[i]$

This means that in a Min Heap, the minimum element is always found at the root node, making it efficient for operations like finding the minimum value in a collection.

Let's understand the min-heap through an example.



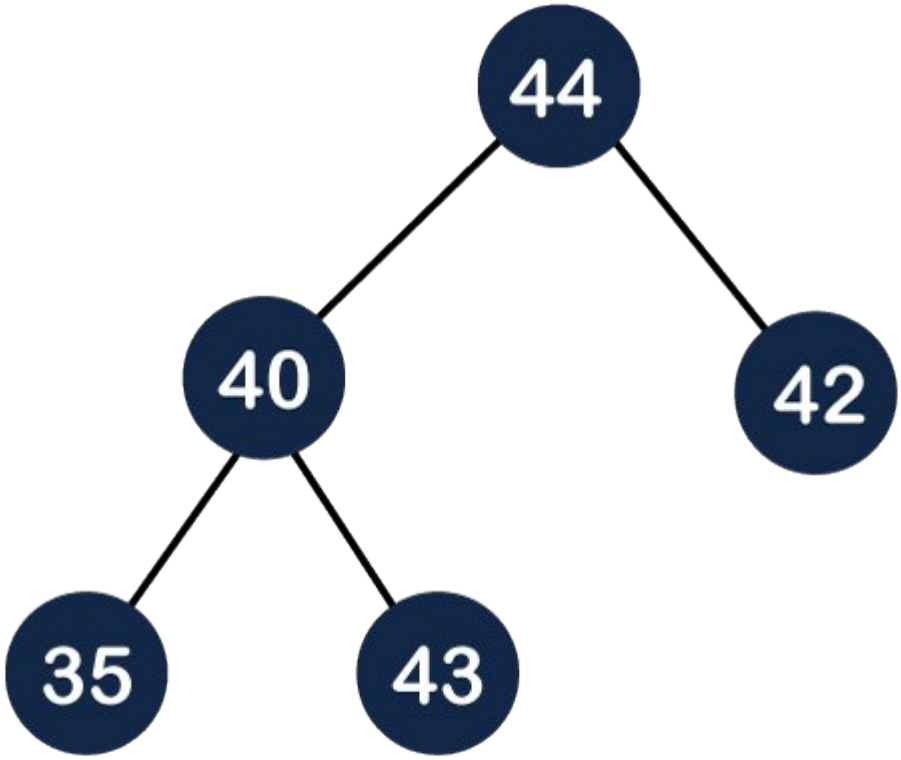
In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time Complexity in Max Heap

Understanding the time complexity of various operations in a Max Heap is crucial for analyzing its efficiency. Below is a breakdown of the time complexity for common operations:

1. Insertion

- **Process**: When inserting an element into a Max Heap, the element is initially added at the end of the heap (which maintains the complete binary tree property). The new element is then "heapified up" (compared with its parent and possibly swapped) to restore the Max Heap property.
- **Time Complexity**: O(log n)
- **Explanation**: In the worst case, the newly inserted element might need to be moved up to the root, which would require O(log n) comparisons and swaps. The height of the heap, which is log n for n elements, determines this complexity.

2. Deletion

- **Process**: The deletion operation typically involves removing the root element (the maximum value in a Max Heap). The last element in the heap is moved to the root position, and then "heapified down" (compared with its children and possibly swapped) to restore the Max Heap property.
- **Time Complexity**: O(log n)
- **Explanation**: After moving the last element to the root, it may need to be moved down to maintain the heap property, requiring O(log n) comparisons and swaps.

3. Peek (Get Maximum)

- **Process**: The maximum value in a Max Heap is always at the root.
- **Time Complexity**: O(1)
- **Explanation**: Accessing the root element is a constant-time operation since it is stored at index 0 in the array representation of the heap.

4. Heapify

- **Process**: Heapify is the process of transforming an arbitrary binary tree into a heap. In a Max Heap, this is typically done by starting from the last non-leaf node and performing "heapify down" operations.
- **Time Complexity**: O(log n) for a single node, O(n) for the entire array
- **Explanation**: The heapify operation on a single node involves traversing the height of the heap (log n). When building a heap from an unsorted array, the total complexity is O(n) because the number of operations decreases as you move up the tree.

5. Build Heap

- **Process**: Building a heap from an unordered array of n elements is done by applying the heapify operation from the bottom-up.
- **Time Complexity**: O(n)
- **Explanation**: While heapifying a single element takes O(log n) time, the build heap process efficiently uses the fact that lower levels of the tree require fewer operations. The aggregate time for heapifying all elements is O(n).

6. Heapsort

- **Process**: Heapsort involves building a Max Heap from an unsorted array and then repeatedly removing the maximum element (root) and placing it at the end of the array, followed by a heapify operation.

- **Time Complexity**: O(n log n)
- **Explanation**: Building the heap takes O(n) time, and each of the n deletions takes O(log n) time, resulting in an overall time complexity of O(n log n).

Operation	Time Complexity	Explanation
Insertion	O(log n)	The new element may need to be moved up to the root, requiring log n comparisons and swaps.
Deletion	O(log n)	After removing the root, the last element may need to be moved down to restore the heap property.
Peek (Get Maximum)	O(1)	Accessing the maximum value (root element) is a constant-time operation.
Heapify (Single Node)	O(log n)	Heapifying a single node involves traversing the height of the heap, which is log n.
Build Heap	O(n)	Building a heap from an array involves heapifying each element, with an overall complexity of O(n).
Heapsort	O(n log n)	Building the heap takes O(n) time, and each of the n deletions takes O(log n) time.

Algorithm to Insert an Element in a Max Heap

The insertion operation in a Max Heap involves placing the new element at the end of the heap and then restoring the Max Heap property by "bubbling up" the new element until it is in the correct position.

Here is the algorithm for inserting an element into a Max Heap:

```
insertHeap(A, n, value)
{
    n = n + 1;               // Increment the size of the heap to insert the new element
    A[n] = value;            // Place the new value at the end of the heap (at index n)
    i = n;                   // Set i to point to the last element (the new element)

    // Loop to restore the Max Heap property by bubbling up the new element
    while (i > 1)            // Continue until the element reaches the root or the heap property is satisfied
    {
        parent = floor(i / 2); // Calculate the parent index of the current element

        // If the value of the parent is less than the value of the current node, swap them
        if (A[parent] < A[i])
        {
            swap(A[parent], A[i]); // Swap the values
            i = parent;            // Move up to the parent node
        }
        else
        {
            return;              // If the parent is larger, the heap property is satisfied, so stop
        }
```

Explanation:

1. **Increment the Heap Size**: The size of the heap (`n`) is increased by 1 to accommodate the new element.
2. **Insert the New Element**: The new value is placed at the end of the heap array (`A[n]`).
3. **Bubble Up (Heapify Up)**:
   - The algorithm compares the newly added element with its parent.
   - If the parent is smaller (in a Max Heap), the values are swapped to maintain the Max Heap property.
   - This process repeats, moving the new element up the heap until it either reaches the root or is smaller than its parent.
1. **Stop Condition**: The loop stops when the element is in the correct position, satisfying the Max Heap property, or when it becomes the root node.

This algorithm ensures that the heap remains a Max Heap after inserting a new element.

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

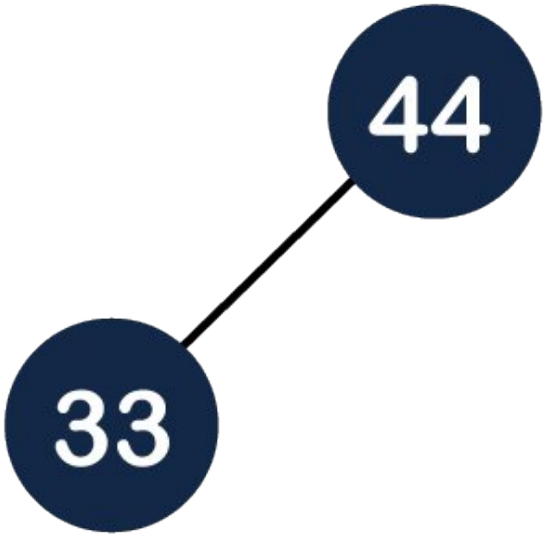
44, 33, 77, 11, 55, 88, 66

Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

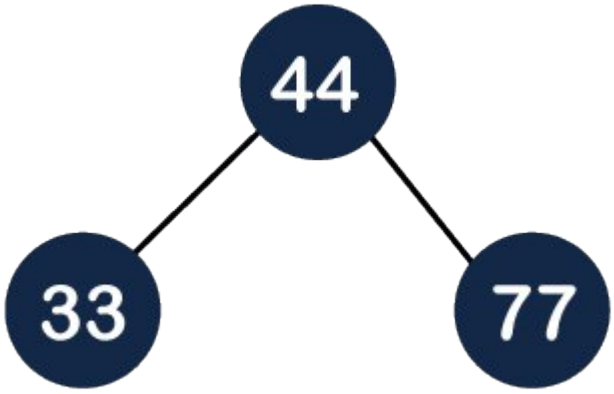
- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

Step 1: First we add the 44 element in the tree as shown below:

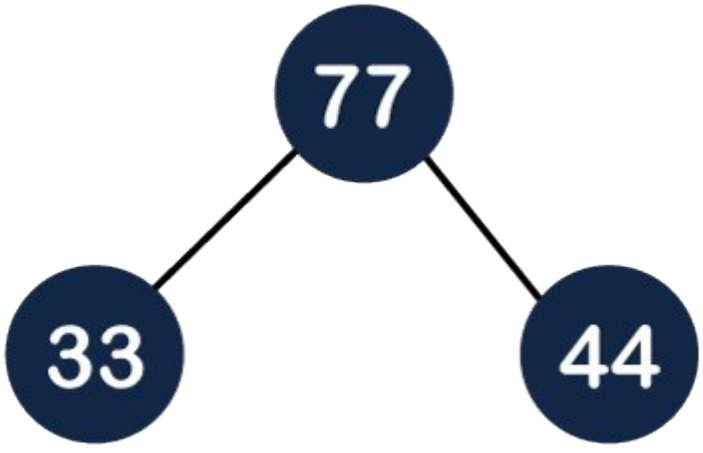
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



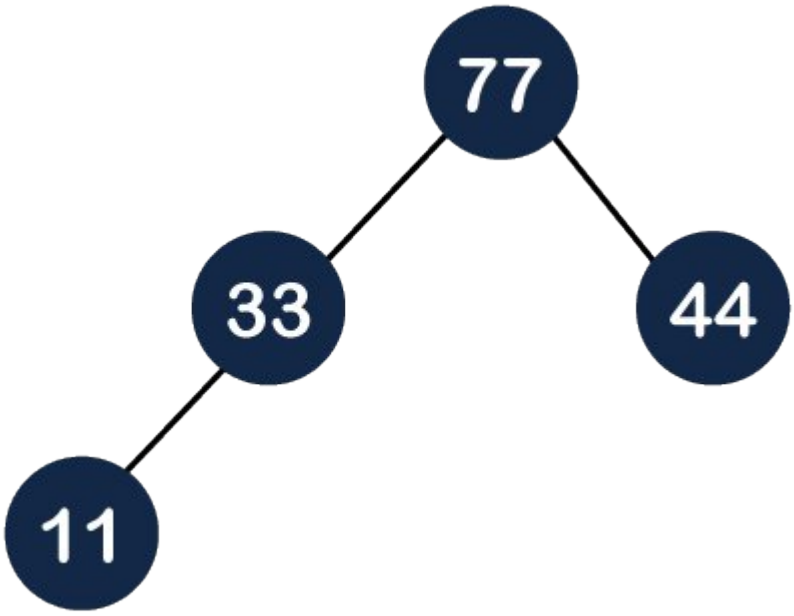
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



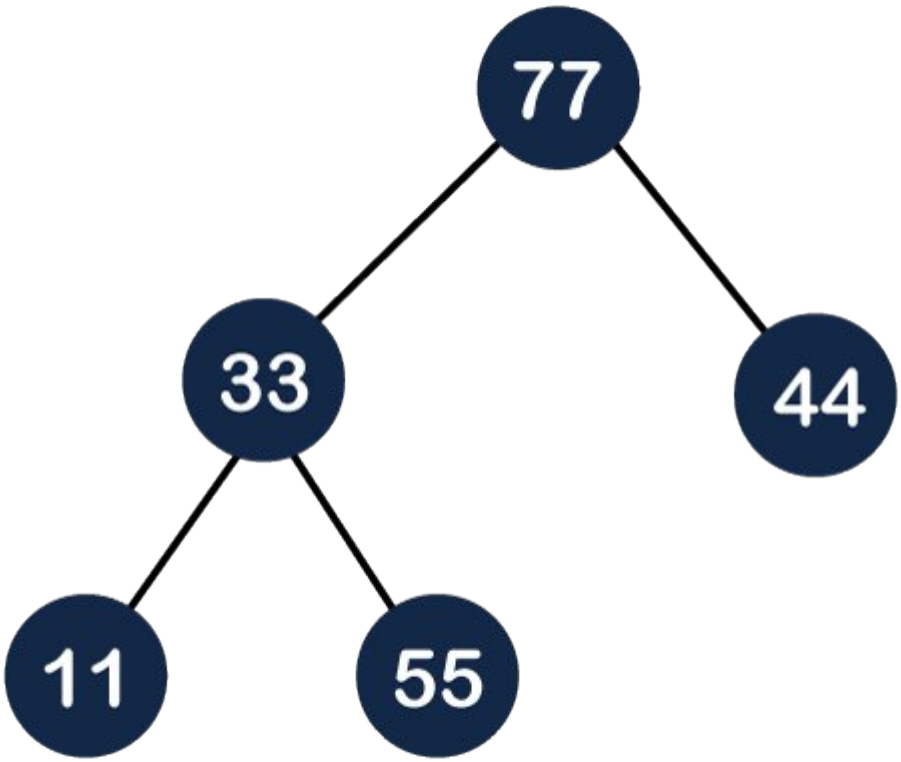
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



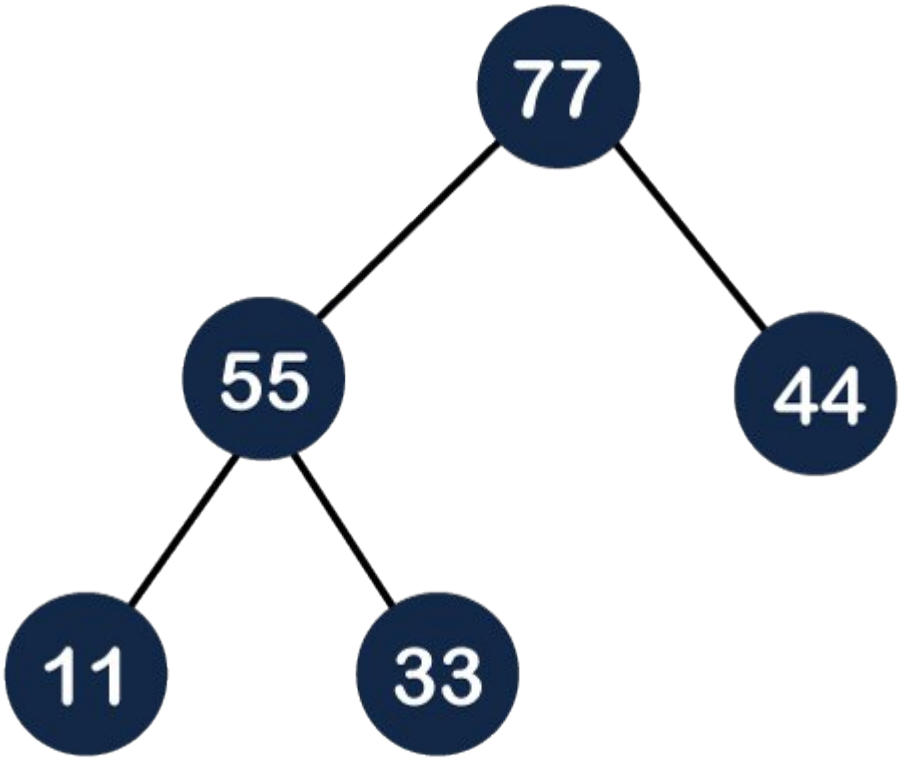
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



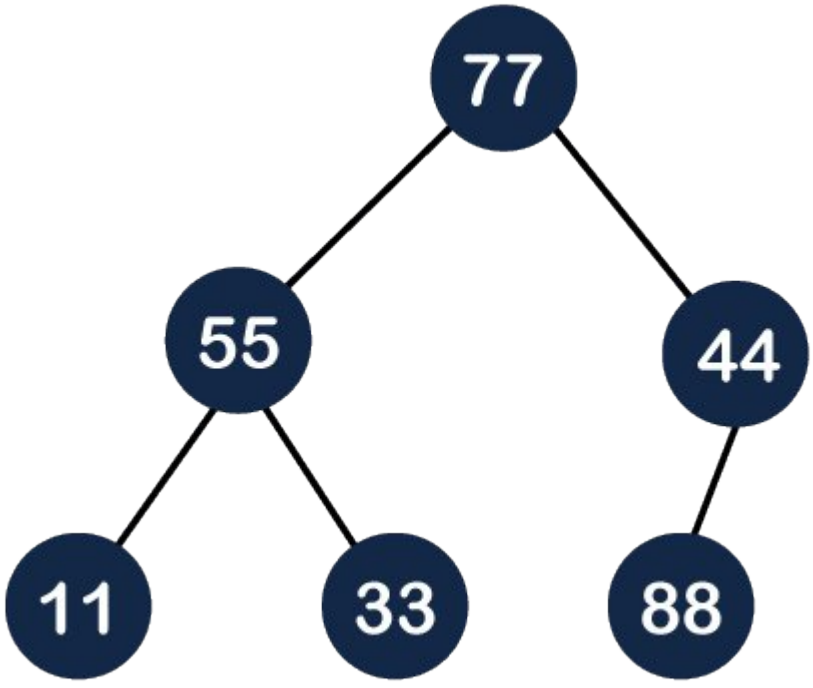
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 33<55, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because 44<88, so we will swap these two values as shown below:

Again, it is violating the max heap property because 88>77 so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1: In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to Heapify a Subtree in a Max Heap

```
// Algorithm to heapify a subtree rooted at index i in a Max Heap
MaxHeapify(A, n, i)
{
    int largest = i;       // Initialize largest as root
    int l = 2 * i;         // Left child index (2 * i)
    int r = 2 * i + 1;     // Right child index (2 * i + 1)

    // If the left child exists and is greater than the root
    if (l <= n && A[l] > A[largest])
    {
        largest = l;       // Update largest to the left child
    }

    // If the right child exists and is greater than the current largest
    if (r <= n && A[r] > A[largest])
    {
        largest = r;       // Update largest to the right child
    }

    // If the largest element is not the root
    if (largest != i)
    {
        swap(A[largest], A[i]); // Swap the root with the largest child
        MaxHeapify(A, n, largest); // Recursively heapify the affected subtree
    }
}
```

Max Heap Implementation in

Here's a class `Max_Heap` that implements a max-heap data structure. This class includes methods for inserting elements, deleting the maximum element, and maintaining the heap property using heapify operations.

1. Initialization (`__init__`)

```
class Max_Heap:
    def __init__(self) -> None:
        self.heap = []
```

Explanation:

- The `__init__` method initializes an empty list named `heap`, which will be used to store the elements of the heap.

2. Insert Operation (`insert`)

```
def insert(self, value):
    Insert a new value into the max-heap."""
self.heap.append(value)  # Add the value to the end of the heap
self.heapify_up(len(self.heap) - 1)  # Restore the heap property
```

Explanation:

- Adds the new `value` to the end of the heap.
- Calls `heapify_up` to ensure the new element is in the correct position to maintain the max-heap property.

3. Heapify Up (`heapify_up`)

```
def heapify_up(self, index):
        Restore the heap property by bubbling up the element at
    index."""     parent_index = (index - 1) // 2
    while index > 0 and self.heap[parent_index] < self.heap[index]:
        # Swap the element with its parent if it's larger
            self.heap[parent_index], self.heap[index] = self.heap[index],
            self.heap[parent_index]
            index = parent_index
     parent_index = (index - 1) // 2
```

Explanation:

- **Calculate the Parent Index:** Determine the parent index of the current node using `(index - 1) // 2`.
- **Compare and Swap:** While the current node is larger than its parent, swap the current node with its parent.
- **Update Indexes:** Move the index up to the parent’s index and repeat the process until the heap property is restored or the root is reached.

4. Delete Operation (`delete`)

```
def delete(self):
        Remove and return the maximum element (root) from the max-heap."""     if len(self.heap) == 0:
        raise ValueError("Delete from empty heap!")
    if len(self.heap) == 1:
        return self.heap.pop()
    root = self.heap[0]
    self.heap[0] = self.heap.pop()  # Move the last element to the root
    self.heapify_down(0)  # Restore the heap property
    return root
```

Explanation:

- **Check for Empty Heap:** Raise an error if the heap is empty.
- **Single Element Case:** If there is only one element, remove and return it.
- **Replace Root:** Move the last element to the root position.
- **Heapify Down:** Call `heapify_down` to restore the heap property after removing the root.

5. Heapify Down (`heapify_down`)

```
def heapify_down(self, index):
  Restore the heap property by bubbling down the element at index."""
   last_index = len(self.heap) - 1
   while True:
      left_index = 2 * index + 1
      right_index = 2 * index + 2
      largest = index

      # Find the largest among the current node, left child, and right child
      if left_index <= last_index and self.heap[left_index] > self.heap[largest]:
         largest = left_index
      if right_index <= last_index and self.heap[right_index] > self.heap[largest]:
         largest = right_index

      # If the largest is not the current node, swap and continue heapifying
      if largest == index:
         break
      self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]
```

```
        index = largest
```

Explanation:

- **Calculate Child Indices:** Determine the indices of the left and right children.
- **Find Largest:** Compare the current node with its children to find the largest element.
- **Swap and Continue:** If the largest element is not the current node, swap the current node with the largest child and continue heapifying down the subtree.

Example Usage

```
if __name__ == "__main__":
    obj = Max_Heap()
    print("Heap before insertion:", obj.heap)
    obj.insert(10)
    obj.insert(20)
    obj.insert(30)
    obj.insert(40)
    print("Heap after insertion:", obj.heap)
    obj.delete()
    print("Heap after deletion:", obj.heap)
```

Output:

```
mustafa@MSI:~/Learning/DSA/Heaps & Hashing$ /bin/python3 "/home/mustafa/Learning/DSA/Heaps & Hashing/MaxHeap.py"
Heap before insertion: []
Heap after insertion: [40, 30, 20, 10]
Heap before insertion: [30, 10, 20]
```

Here’s a detailed explanation of each function in the `MinHeap` class:

1. `__init__`

```
def __init__(self) -> None:
    self.heap = []
```

Purpose: Initializes a new instance of the `MinHeap` class.

Details:

- Creates an empty list `self.heap` which will store the elements of the heap.

Certainly! Here’s a detailed explanation of each function in the `MinHeap` class:

2. `insert`

```
def insert(self, value):
  Insert a new value into the min-heap."""
   self.heap.append(value)  # Add the value to the end of the heap
   self.heapify_up(len(self.heap) - 1)  # Restore the heap property
   return value
```

Purpose: Inserts a new value into the min-heap and maintains the heap property.

Details:

Add the Value: The new value is appended to the end of the list, maintaining the complete binary tree structure.

Heapify Up: After adding the new value, it may violate the min-heap property. The `heapify_up` method is called to move the new value up the heap until the heap property is restored.

Return: Returns the inserted value for confirmation.

3. `heapify_up`

```
def heapify_up(self, index):
  Restore the heap property by bubbling up the element at index."""
   parent_index = (index - 1) // 2
   while index > 0 and self.heap[parent_index] > self.heap[index]:
      # Swap the element with its parent if it's smaller
      self.heap[parent_index], self.heap[index] = self.heap[index],
self.heap[parent_index]
      index = parent_index
      parent_index = (index - 1) // 2
```

Purpose: Ensures that the heap property is maintained by moving the element at a given index up the heap.

Details:

Find Parent: Calculates the index of the parent of the current node.

Bubble Up: Compares the current node with its parent. If the current node's value is smaller (which violates the min-heap property), it swaps the values and moves up to the parent's position.

Loop: Continues until the current node is the root (index 0) or the heap property is restored (i.e., the current node is not smaller than its parent).

4. `delete`

```
def delete(self):
  Remove and return the minimum element (root) from the min-
heap."""
   if len(self.heap) == 0:
      raise ValueError("Delete from empty heap!")
   if len(self.heap) == 1:
      return self.heap.pop()
   root = self.heap[0]
   self.heap[0] = self.heap.pop()  # Move the last element to the root
   self.heapify_down(0)  # Restore the heap property
   return root
```

Purpose: Removes and returns the minimum element (root) from the min-heap and maintains the heap property.

Details:

Check Empty: Raises an error if the heap is empty.

Single Element: If there's only one element, it is removed and returned directly.

Remove Root: If there are multiple elements, the root (minimum value) is removed. The last element in the heap is moved to the root position.

Heapify Down: Calls `heapify_down` to move the new root down the heap to restore the heap property.

Return: Returns the removed root value.

5. `heapify_down`

```
def heapify_down(self, index):
  Restore the heap property by bubbling down the element at index."""
   last_index = len(self.heap) - 1
   while True:
      left_index = 2 * index + 1
      right_index = 2 * index + 2
      smallest = index

      # Find the smallest among the current node, left child, and right
child
      if left_index <= last_index and self.heap[left_index] <
self.heap[smallest]:
```

```
            smallest = left_index
        if right_index <= last_index and self.heap[right_index] <
self.heap[smallest]:
            smallest = right_index

        # If the smallest is not the current node, swap and continue
heapifying
        if index == smallest:
            break
        self.heap[smallest], self.heap[index] = self.heap[index],
self.heap[smallest]
        index = smallest
```

Purpose: Ensures that the heap property is maintained by moving the element at a given index down the heap.

Details:

Find Children: Calculates the indices of the left and right children of the current node.

Find Smallest: Determines which of the current node and its children has the smallest value.

Swap and Bubble Down: If the smallest value is not the current node, it swaps values with the smallest child and continues the process to ensure the entire subtree satisfies the min-heap property.

Example Usage

```
if __name__ == "__main__":
    obj = MinHeap()
    print("Heap before insertion:", obj.heap)
    obj.insert(10)
    obj.insert(20)
    obj.insert(30)
    obj.insert(40)
    print("Heap after insertion:", obj.heap)
    obj.delete()
    print("Heap after deletion:", obj.heap)
```

Before Insertion: Shows the empty heap.

After Insertion: Shows the heap after inserting several elements.

After Deletion: Shows the heap after removing the minimum element (root).

Output:

```
mustafa@MSI:~/Learning/DSA/Heaps & Hashing$ /bin/python3 "/home/mustafa/Learning/DSA/Heaps & Hashing/MinHeap.py"
Heap before insertion: []
Heap after insertion: [10, 20, 30, 40]
Heap before insertion: [20, 40, 30]
```

Heapsort Algorithm

Heapsort is a comparison-based sorting algorithm that leverages the heap data structure to sort an array. It involves two main phases:

1. **Heap Construction**: Convert the array into a heap (typically a max-heap for ascending order).
2. **Sorting**: Repeatedly extract the maximum element from the heap and rebuild the heap until the array is sorted.

Here's a step-by-step explanation and the Python implementation of heapsort:

1. Build a Max Heap

A max-heap is a complete binary tree where each node's value is greater than or equal to its children's values. Building a max-heap from an array involves calling `heapify` on each internal node, starting from the last non-leaf node up to the root node.

2. Extract Elements from the Heap

Once the max-heap is built, the root node of the heap (which contains the maximum element) is repeatedly extracted and placed at the end of the array. The heap size is reduced by one each time, and the heap property is restored for the reduced heap.

```
Algorithm Heapsort(A):
    Input: Array A with n elements

    // Step 1: Build a min-heap from the input array
    for i from floor(n / 2) - 1 down to 0:
        MinHeapify(A, n, i)

    // Step 2: Extract elements from the heap one by one
    for i from n - 1 down to 1:
        swap(A[0], A[i])         // Move current root (min) to end
        MinHeapify(A, i, 0)      // Restore the min-heap property for reduced heap

// MinHeapify function to ensure the subtree rooted at index i is a min-heap
Algorithm MinHeapify(A, n, i):
    smallest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and A[left] < A[smallest]:
        smallest = left

    if right < n and A[right] < A[smallest]:
        smallest = right

    if smallest != i:
        swap(A[i], A[smallest])
        MinHeapify(A, n, smallest)
```

Python Implementation

Here's the complete Python code for **heapsort**:

```
def heapify(arr, n, i):
"""Ensure the subtree rooted at index i is a min-heap."""
smallest = i        # Initialize smallest as root
left = 2 * i + 1    # Left child index
right = 2 * i + 2   # Right child index

# If the left child exists and is smaller than root
if left < n and arr[left] < arr[smallest]:
    smallest = left

# If the right child exists and is smaller than the smallest so far
if right < n and arr[right] < arr[smallest]:
    smallest = right

# If the smallest is not the root
if smallest != i:
    arr[i], arr[smallest] = arr[smallest], arr[i]  # Swap root with smallest
    heapify(arr, n, smallest)  # Recursively heapify the affected subtree

def heapsort(arr):
"""Perform heapsort using a min-heap."""
n = len(arr)

# Build a min-heap from the array
for i in range(n // 2 - 1, -1, -1):
    heapify(arr, n, i)

# Extract elements from the min-heap one by one
for i in range(n - 1, 0, -1):
    arr[0], arr[i] = arr[i], arr[0]  # Move current root to the end
    heapify(arr, i, 0)  # Call heapify on the reduced heap

if __name__ == "__main__":
arr = [12, 11, 13, 5, 6, 7]
print("Original array:", arr)
heapsort(arr)
print("Sorted array:", arr)
```

Output:

```
mustafa@MSI:~/Learning/DSA/Heaps & Hashing$ /bin/python3 "/home/mustafa/Learning/DSA/Heaps & Hashing/MinHeap.py"
arr = [5, 6, 7, 11, 12, 13]
```

Explanation:

1. **Heapify Function**:
   - Ensures that a subtree rooted at index `i` is a min-heap.
   - Compares the current node with its children and swaps if necessary to maintain the min-heap property.
   - Recursively heapifies the affected subtree.
1. **Heapsort Function**:

- **Build Min-Heap**: Calls `heapify` starting from the last non-leaf node up to the root to build a min-heap.
- **Sort**: Repeatedly swaps the root of the heap (the minimum element) with the last element in the array and then heapifies the reduced heap.

Time Complexity:

- **Building the Heap**: O(n)
- **Heap Sort**: O(n log n)
- Each of the `n` extractions involves `O(log n)` operations.

Space Complexity:

- **Heapsort**: O(1) (in-place sorting)

This implementation sorts the array in ascending order using a min-heap, which is an alternative to the more common approach using a max-heap.