**What is a Heap?**

A **heap** is a complete binary tree that satisfies the **heap property**. It is used primarily to implement **priority queues**. A heap is efficient in operations like insertion, deletion, and retrieval of the maximum or minimum element.

**Heap Representation**

A heap is typically represented as a binary tree, where the value of each node is compared with its parent and children.

**Array Representation:**

In the array representation of a heap:

* The **root** is at index 0.
* The **left child** of node i is at index 2i + 1.
* The **right child** of node i is at index 2i + 2.
* The **parent** of node i is at index (i - 1) // 2.

**Two Types of Heaps**

1. **Max Heap**:
   * The value of each node is **greater than or equal to** the values of its children.
   * The **largest value** is at the root of the tree.

**Condition**: For any node i, the value of node i must be greater than or equal to its left and right children.

1. **Min Heap**:
   * The value of each node is **less than or equal to** the values of its children.
   * The **smallest value** is at the root of the tree.

**Condition**: For any node i, the value of node i must be less than or equal to its left and right children.

**Example Representation**

**Max Heap Example (Graphical Representation):**

10

/ \

9 8

/ \ / \

4 7 6 5

**Array Representation**: [10, 9, 8, 4, 7, 6, 5]

**Min Heap Example (Graphical Representation):**

1

/ \

3 6

/ \ / \

5 4 8 7

**Array Representation**: [1, 3, 6, 5, 4, 8, 7]

**Finding Parent and Child Nodes**

* **Parent Node**: The parent of a node at index i is located at (i - 1) // 2.
* **Left Child Node**: The left child of a node at index i is at index 2i + 1.
* **Right Child Node**: The right child of a node at index i is at index 2i + 2.

**Insertion in Heap**

The insertion process involves:

1. Adding the new element at the **end** of the heap (maintaining the complete binary tree structure).
2. **Bubbling up** (or **heapifying up**) the element to restore the heap property.

**Insertion Example in Max Heap:**

Before Insertion:

10

/ \

9 8

/ \ / \

4 7 6 5

Insert 12:

10

/ \

9 8

/ \ / \

4 7 6 5

/

12

After bubbling up:

12

/ \

10 8

/ \ / \

9 7 6 5

/

4

**Code for Insertion**:

def heapify\_up(arr, index):

parent = (index - 1) // 2

while index > 0 and arr[parent] < arr[index]:

arr[parent], arr[index] = arr[index], arr[parent]

index = parent

parent = (index - 1) // 2

def insert(arr, value):

arr.append(value) # Add the new element at the end

heapify\_up(arr, len(arr) - 1) # Heapify up to maintain heap property

# Example usage

arr = [10, 9, 8, 4, 7, 6, 5]

insert(arr, 12)

print(arr) # Output: [12, 10, 8, 9, 7, 6, 5, 4]

**Deletion in Heap**

The deletion process usually involves:

1. Replacing the **root** element with the **last element** of the heap.
2. **Bubbling down** (or **heapifying down**) the root element to restore the heap property.

**Deletion Example in Max Heap:**

Before Deletion:

12

/ \

10 8

/ \ / \

9 7 6 5

/

4

After swapping the root with the last element and bubbling down:

10

/ \

9 8

/ \ / \

4 7 6 5

**Code for Deletion**:

def heapify\_down(arr, index):

size = len(arr)

largest = index

left = 2 \* index + 1

right = 2 \* index + 2

if left < size and arr[left] > arr[largest]:

largest = left

if right < size and arr[right] > arr[largest]:

largest = right

if largest != index:

arr[index], arr[largest] = arr[largest], arr[index]

heapify\_down(arr, largest)

def delete\_root(arr):

if len(arr) == 0:

return None

# Replace root with last element

arr[0], arr[-1] = arr[-1], arr[0]

root = arr.pop() # Remove last element

heapify\_down(arr, 0) # Heapify down the root

return root

# Example usage

arr = [12, 10, 8, 9, 7, 6, 5, 4]

deleted\_element = delete\_root(arr)

print(f"Deleted Element: {deleted\_element}") # Output: 12

print(arr) # Output: [10, 9, 8, 4, 7, 6, 5]

**Summary**

* **Heap** is a complete binary tree with either the **max-heap** or **min-heap** property.
* **Max Heap**: Parent nodes are greater than or equal to their children.
* **Min Heap**: Parent nodes are less than or equal to their children.
* **Insertion**: Insert new element at the end and **heapify up**.
* **Deletion**: Replace root with the last element and **heapify down**.

### Heap and Priority Queue

A **priority queue** is an abstract data structure where each element is associated with a "priority." In this structure, elements with higher priorities are dequeued before elements with lower priorities. It is typically used in algorithms like **Dijkstra’s shortest path** and **Huffman coding**.

A **heap** is often used to implement a priority queue because it provides an efficient way to retrieve and remove the element with the highest (or lowest) priority.

### Why Heap is Used in Priority Queue?

A heap satisfies the **heap property** (either max or min heap), which allows:

* **Efficient access** to the highest or lowest priority element (O(1) time complexity for the root).
* **Efficient insertion** and **deletion** of elements (O(log n) time complexity) because elements are always inserted or removed in a way that maintains the heap property.

### Types of Priority Queues

1. **Max Priority Queue**:
   * In a max heap, the **largest element** has the highest priority.
   * The **root** node contains the largest value.
2. **Min Priority Queue**:
   * In a min heap, the **smallest element** has the highest priority.
   * The **root** node contains the smallest value.

### Operations in a Priority Queue Using Heap

1. **Insertion**: Insert a new element into the heap and maintain the heap property by **bubbling up**.
2. **Removal**: Remove the element with the highest priority (the root), replace it with the last element, and restore the heap property by **bubbling down**.

### Priority Queue Example Using a Max Heap

In a **Max Priority Queue**, elements are arranged so that the **maximum element** is always at the root. When we remove an element, the heap property is restored by moving the largest of the children to the root.

#### Max Priority Queue Example:

Assume we are inserting the following elements into a max priority queue:

5, 8, 3, 1, 10, 6

##### Step 1: Insert 5

Heap: [5]

##### Step 2: Insert 8

Heap: [8, 5] (8 is greater than 5, so it becomes the root)

##### Step 3: Insert 3

Heap: [8, 5, 3] (3 is added as the left child of 5)

##### Step 4: Insert 1

Heap: [8, 5, 3, 1] (1 is added as the left child of 3)

##### Step 5: Insert 10

Heap: [10, 8, 3, 1, 5] (10 is added as the root)

##### Step 6: Insert 6

Heap: [10, 8, 6, 1, 5, 3] (6 is added as the right child of 3)

#### Final Max Heap:

10

/ \

8 6

/ \ /

1 5 3

### Max Priority Queue Operations

1. **Insert an element**: Add the element at the end and **heapify up** to restore the heap property.
2. **Remove the highest priority element** (root):
   * Replace the root with the last element.
   * Remove the last element.
   * **Heapify down** the new root to restore the heap property.

#### Max Priority Queue Removal (Extract Max) Example:

Given the heap:

10

/ \

8 6

/ \ /

1 5 3

**Removing the root (max element 10)**:

1. Replace 10 with the last element (3).
2. Remove the last element.

Heap before heapify down:

3

/ \

8 6

/ \ /

1 5 -

1. **Heapify down**: Compare 3 with its children (8 and 6), and swap it with the larger child (8).

Heap after first swap:

8

/ \

3 6

/ \ /

1 5 -

1. Now, compare 3 with its children (1 and 5), and swap it with the larger child (5).

Final heap:

8

/ \

5 6

/ \

1 3

The root element (10) has been removed, and the heap property has been restored.

### Python Implementation of a Max Priority Queue Using a Heap

class MaxPriorityQueue:

def \_\_init\_\_(self):

self.heap = []

def insert(self, value):

self.heap.append(value) # Insert value at the end

self.\_heapify\_up(len(self.heap) - 1)

def \_heapify\_up(self, index):

parent = (index - 1) // 2

while index > 0 and self.heap[parent] < self.heap[index]:

self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]

index = parent

parent = (index - 1) // 2

def remove(self):

if len(self.heap) == 0:

return None

root = self.heap[0]

self.heap[0] = self.heap[-1]

self.heap.pop()

self.\_heapify\_down(0)

return root

def \_heapify\_down(self, index):

size = len(self.heap)

largest = index

left = 2 \* index + 1

right = 2 \* index + 2

if left < size and self.heap[left] > self.heap[largest]:

largest = left

if right < size and self.heap[right] > self.heap[largest]:

largest = right

if largest != index:

self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]

self.\_heapify\_down(largest)

def get\_max(self):

return self.heap[0] if self.heap else None

# Example usage

pq = MaxPriorityQueue()

pq.insert(10)

pq.insert(8)

pq.insert(3)

pq.insert(1)

pq.insert(12)

pq.insert(6)

print("Max Priority Queue (Heap):", pq.heap) # Output: [12, 10, 6, 1, 8, 3]

# Remove the maximum element

print("Removed Element:", pq.remove()) # Output: 12

print("Max Priority Queue after Removal:", pq.heap) # Output: [10, 8, 6, 1, 3]

### Summary

* **Heap** is an efficient data structure for implementing **priority queues** due to its ability to quickly retrieve the maximum (or minimum) element and efficiently insert or delete elements.
* A **Max Priority Queue** ensures that the maximum element is always accessible at the root, while a **Min Priority Queue** ensures that the minimum element is always at the root.
* The operations **insert** and **remove** are performed in logarithmic time, O(log n), thanks to the heap property.

Using a heap as the underlying structure of a priority queue allows for optimal performance in scenarios where frequent access to the highest (or lowest) priority element is needed.

1. Define a class Heap to implement Heap data structure with init method to create empty heap list. 2. In class Heap, define a method to create heap from a given list of elements.

3. In class Heap, define a method insert to insert a given element in the heap at appropriate position.

4. In class Heap, define a top method which returns the top element of the Heap. Raise an exception if Heap is empty.

5. Define a class EmptyHeapException to describe custom exception

6. In class Heap, define a method delete which deletes the top element and returns it Raise an exception if Heap is empty

7. In class Heap, define a method heapSort to sort a given list with the help of Heap

Here's a Python implementation of the Heap class with the required methods as specified:

# Custom exception for empty heap

class EmptyHeapException(Exception):

def \_\_init\_\_(self, message="Heap is empty"):

self.message = message

super().\_\_init\_\_(self.message)

# Heap class definition

class Heap:

def \_\_init\_\_(self):

self.heap = []

# Method to create a heap from a given list

def create\_heap(self, elements):

self.heap = elements

# Build the heap by calling heapify on each non-leaf node

for i in range(len(self.heap) // 2 - 1, -1, -1):

self.\_heapify(i)

# Method to insert a given element at the correct position

def insert(self, element):

self.heap.append(element) # Insert element at the end

self.\_heapify\_up(len(self.heap) - 1) # Restore heap property

# Method to return the top element of the heap (max or min based on heap type)

def top(self):

if not self.heap:

raise EmptyHeapException("Heap is empty, cannot access top element.")

return self.heap[0]

# Method to delete the top element (max or min) and return it

def delete(self):

if not self.heap:

raise EmptyHeapException("Heap is empty, cannot delete top element.")

top\_element = self.heap[0]

# Replace the root with the last element and heapify down

self.heap[0] = self.heap[-1]

self.heap.pop() # Remove the last element

self.\_heapify(0) # Restore heap property

return top\_element

# Method to perform heap sort

def heapSort(self):

sorted\_list = []

while self.heap:

sorted\_list.append(self.delete()) # Continuously remove the top element

return sorted\_list

# Internal helper method to restore heap property by heapifying up

def \_heapify\_up(self, index):

parent = (index - 1) // 2

while index > 0 and self.heap[parent] < self.heap[index]:

# Swap if current element is greater than parent in max heap

self.heap[parent], self.heap[index] = self.heap[index], self.heap[parent]

index = parent

parent = (index - 1) // 2

# Internal helper method to restore heap property by heapifying down

def \_heapify(self, index):

largest = index

left = 2 \* index + 1

right = 2 \* index + 2

size = len(self.heap)

# Check if left child exists and is larger than current node

if left < size and self.heap[left] > self.heap[largest]:

largest = left

# Check if right child exists and is larger than current largest node

if right < size and self.heap[right] > self.heap[largest]:

largest = right

# If largest is not the current node, swap and heapify down

if largest != index:

self.heap[index], self.heap[largest] = self.heap[largest], self.heap[index]

self.\_heapify(largest)

# Example usage

heap = Heap()

# Create heap from a list of elements

heap.create\_heap([3, 9, 2, 1, 4, 5])

# Insert an element into the heap

heap.insert(10)

# Get the top element of the heap

print("Top element:", heap.top()) # Output: 10

# Delete the top element from the heap

print("Deleted element:", heap.delete()) # Output: 10

# Perform heap sort

sorted\_list = heap.heapSort()

print("Sorted list:", sorted\_list) # Output: [9, 5, 4, 3, 2, 1]

**Explanation:**

1. **Class EmptyHeapException:**
   * This class defines a custom exception that is raised when the heap is empty and an operation is attempted that requires the heap to have elements.
2. **Class Heap:**
   * **\_\_init\_\_(self)**: Initializes an empty heap list.
   * **create\_heap(self, elements)**: This method takes a list of elements and builds a heap by calling the \_heapify method for each non-leaf node.
   * **insert(self, element)**: Inserts an element at the end of the list and calls \_heapify\_up to maintain the heap property (for max heap, the parent node is larger than its children).
   * **top(self)**: Returns the top element (root) of the heap, raises EmptyHeapException if the heap is empty.
   * **delete(self)**: Removes the top element and replaces it with the last element, then calls \_heapify to restore the heap property. Raises EmptyHeapException if the heap is empty.
   * **heapSort(self)**: Removes elements from the heap one by one and appends them to a sorted list. It performs heap sort by calling delete() repeatedly until the heap is empty.
3. **Helper Methods:**
   * **\_heapify\_up(self, index)**: Ensures that the heap property is maintained by moving an element up the tree if necessary.
   * **\_heapify(self, index)**: Ensures that the heap property is maintained by moving the element at the specified index down the tree.

**Example Output:**

Top element: 10

Deleted element: 10

Sorted list: [9, 5, 4, 3, 2, 1]

This implementation allows you to create a heap, insert elements, get the top element, delete the top element, and perform heap sort on the heap data structure.