**Lab-9: 19CSE212 Data Structure and Algorithms**

**Implement Heap ADT, and apply heap in real-world problem-solving**

**Mode: Online, Dt. 12-04-2024**

**Assignments**

1. **Write a Python code to build a MAX-HEAP from the array A of 12 elements, where A = {16, 19, 20, 25, 28, 33, 40, 42, 55, 56, 70, 85}.**

**Implement the following ADT of MAX-HEAP in Python that you have built based on the array A.**

1. **MAX(A): Print the largest element in A**
2. **EXTRACT\_MAX(A): Remove the node containing the largest element and print the removed largest element.**
3. **INCREASE\_KEY(A, 90, 40): Increase the element value from 40 to 90, and print the resultant MAX-HEAP.**
4. **HEAP\_SORT(A): implement the HEAP\_SORT algorithm to print heap elements in the non-decreasing order of their values.**

**Note: please refer to the resources shared on the heap and priority queue data structures to learn the algorithms for the given ADTs.**

1. **Write a Python code to build a MIN-HEAP from the array A of 12 elements, where A = {85, 70, 55, 56, 40, 42, 33, 16, 28, 19, 20, 25}.**

**Implement the following ADT of MIN-HEAP in Python that you have built based on the array A.**

1. **MIN(A): Print the smallest element in A**
2. **EXTRACT\_MIN(A): Remove the node containing the smallest element and print the removed smallest element.**
3. **DECREASE\_KEY(A, 10, 85): Decrease the element value from 85 to 10, and print the resultant MIN-HEAP.**

**Scenario 1: (Based on implementation of Priority Queue)**

In a bustling city, there's a centralized emergency response system responsible for **dispatching medical assistance, firefighting**, and **police intervention**. This system receives **emergency calls from various sources, each with different levels of urgency and priority.** To efficiently manage these emergencies and allocate resources effectively, the system relies on a binary heap data structure.

**System Requirements:**

**Emergency Call Reception:** The emergency response system constantly receives calls reporting emergencies. Each call is tagged with a priority level indicating the severity of the situation. For example, a medical emergency might have a higher priority than a noise complaint.

**Heap Insertion:** As calls come in, they are inserted into a binary heap data structure based on their priority. The priority queue ensures that the most critical emergencies are addressed first.

**Resource Allocation:** Emergency responders, such as ambulances, fire trucks, and police cars, are stationed strategically across the city. When a call needs to be addressed, the system retrieves the highest-priority emergency from the binary heap.

**Dispatching:** The system dispatches the appropriate resources to the emergency location based on the type and severity of the situation. For instance, a medical emergency requires an ambulance, while a fire requires a fire truck.

**Priority-Based Handling:** As new emergencies or existing ones escalate, the binary heap ensures that the highest priority incidents are always addressed first. This prevents delays in responding to critical situations and maximizes the efficiency of the emergency response system.

**Dynamic Updates:** The system continuously updates the binary heap as new emergencies are reported or resolved. This dynamic adjustment ensures that resources are allocated optimally, considering the changing landscape of emergencies throughout the city.

Based on the above scenario, answer the following questions.

1. Write an algorithm to meet the system requirements.
2. Analyze the worst-case running time of the algorithm.
3. Write the code abstractions based on your algorithm.
4. Implement your algorithm in Python.

**Solution:**

1. **Code Snippet**

**# Define the Emergency class to represent each emergency call**

**class Emergency:**

def \_\_init\_\_(self, priority, description):

self.priority = priority **# Priority of the emergency**

self.description = description **# Description of the emergency**

**# Define the EmergencyResponseSystem class**

**class Emergency\_ResponseSystem:**

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

self.heap = [] **# Initialize an empty binary heap**

**# Helper function to maintain the heap property after insertion**

def \_heapify\_up(self, index):

while index > 0:

parent\_index = (index - 1) // 2

if self.heap[index].priority < self.heap[parent\_index].priority:

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

index = parent\_index

else:

break

**# Helper function to maintain the heap property after extraction**

def \_heapify\_down(self, index):

while True:

left\_child\_index = 2 \* index + 1

right\_child\_index = 2 \* index + 2

smallest = index

if left\_child\_index < len(self.heap) and self.heap[left\_child\_index].priority < self.heap[smallest].priority:

smallest = left\_child\_index

if right\_child\_index < len(self.heap) and self.heap[right\_child\_index].priority < self.heap[smallest].priority:

smallest = right\_child\_index

if smallest != index:

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

index = smallest

else:

break

**# Function to add a new emergency to the system**

def add\_emergency(self, emergency):

self.heap.append(emergency)

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

**# Function to handle the highest priority emergency**

def handle\_emergency(self):

if not self.heap:

return None **# No emergencies in the system**

emergency = self.heap[0] **# Highest priority emergency**

self.heap[0] = self.heap[-1] **# Replace root with the last element**

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

self.\_heapify\_down(0) **# Restore heap property**

return emergency

**# Example usage:**

emergency\_system = EmergencyResponseSystem()

**# Add emergencies to the system**

emergency\_system.add\_emergency(Emergency(3, "Medical emergency"))

emergency\_system.add\_emergency(Emergency(2, "Fire outbreak"))

emergency\_system.add\_emergency(Emergency(1, "Burglary"))

**# Handle emergencies based on priority**

while True:

emergency = emergency\_system.handle\_emergency()

if emergency:

print(f"Handling emergency: {emergency.description} (Priority: {emergency.priority})")

else:

print("No more emergencies in the system")

break

1. **Time Complexity Analysis**

**Adding an Emergency (add\_emergency):**

Appending an emergency to the end of the heap takes **O(1) time**.

Heapifying up to maintain the heap property may take up to O(log n) time, where n is the number of emergencies in the heap. However, in the worst case, it might need to traverse the height of the heap, which is logarithmic. So, the time complexity of adding an emergency is **O(log n).**

**Handling an Emergency (handle\_emergency):**

Retrieving the highest priority emergency (at the root of the heap) takes O(1) time.

Swapping the root with the last element and removing the last element takes O(1) time.

Heapifying down to restore the heap property may take up to O(log n) time, where n is the number of emergencies in the heap. Similar to adding an emergency, it might need to traverse the height of the heap. So, the time complexity of handling an emergency is **O(log n).**

**Overall:**

If there are m emergency calls added to the system, the time complexity to handle all emergencies would be O(m \* log n), where n is the maximum number of emergencies that can be in the heap at any given time.

In scenarios where m is significantly larger than n, the time complexity can be approximated to **O(m \* log n).**

**Assignments**

1. **Scenario 2: Priority Queue-based Project Management Application**

In a large organization, various departments and teams handle a multitude of projects, each with different priorities and deadlines. To ensure efficient project management and timely completion of projects, the company requires a robust project scheduling application. This application should prioritize projects based on their importance, allocate resources effectively, and ensure that critical projects receive prompt attention from the appropriate teams. The application must cater to the following system requirements.

**System Requirements:**

**Project Submission:** The enterprise must have a system for clients to submit projects, each tagged with a priority level indicating its importance and deadline. This system should handle a continuous flow of project submissions from multiple clients.

**Priority Queue Management:** Implementation of a priority queue data structure to manage incoming projects and prioritize them based on their importance and deadlines. The system should insert projects into the queue in real-time as they are submitted by clients.

**Resource Allocation:** Allocation of human and material resources, including skilled employees, equipment, and budget, to handle projects effectively. The system should consider factors such as the availability of specialized teams and the criticality of the project.

**Project Assignment:** Assignment of projects to appropriate teams or individuals based on their expertise and availability. The system should ensure that projects are delegated to the most suitable team to maximize efficiency and productivity. **Dynamic Updates:** Dynamic updating of the priority queue as new projects are submitted or existing ones are completed. The system should adjust the priority of projects based on changes in their importance, deadlines, and the overall workload of the organization.

**Deadline Management:** Monitoring and managing project deadlines to ensure that critical projects are completed on time. The system should provide alerts and notifications for approaching deadlines and facilitate prioritization of projects accordingly.

Based on the above scenario, answer the following questions.

1. Write an algorithm to meet the system requirements based on the priority queue ADTs.
2. Analyze the worst-case running time of the algorithm.
3. Write the code abstractions based on your algorithm.
4. Implement your algorithm in Python.
5. **Scenario 3: Priority Queue-based Hospital Emergency Management System**

In a metro city, a major hospital operates an emergency department that receives patients with a wide range of medical conditions, from minor injuries to life-threatening emergencies. The hospital needs an efficient system to manage patient intake, prioritize cases based on medical urgency, allocate resources effectively, and ensure timely treatment for critical patients. Following are the system requirements.

**System Requirements:**

**Emergency Call Reception:** The hospital must have a system in place to receive incoming patients, categorizing each case based on the severity of the medical condition. This system should handle a continuous stream of patients efficiently, ensuring that no emergencies go unnoticed.

**Priority Queue Management:** Implementation of a priority queue data structure to manage patient intake and prioritize cases based on medical urgency. The system should insert patients into the queue in real-time as they arrive at the emergency department.

**Resource Allocation:** Allocation of hospital staff, including doctors, nurses, and medical equipment, to address the most critical cases promptly. The system should consider factors such as the availability of specialized medical personnel and critical care units.

**Dispatching:** Efficient dispatching of hospital resources to patient rooms or treatment areas based on the severity and type of medical emergency. The system should facilitate communication and coordination among hospital staff to ensure timely responses.

**Dynamic Updates:** Dynamic updating of the priority queue as new patients arrive or existing cases are resolved. The system should adjust the priority of patients based on changes in their medical condition and the overall workload of the emergency department.

Based on the above scenario, answer the following questions.

1. Write an algorithm to meet the system requirements based on the priority queue ADTs.
2. Analyze the worst-case running time of the algorithm.
3. Write the code abstractions based on your algorithm.
4. Implement your code abstractions based on your algorithm in Python.