Here is a Python implementation of a priority queue using a list as the underlying data structure:

```python

class PriorityQueue:

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

# Initialize an empty list to hold the queue items, each item is a tuple (priority, data)

self.queue = []

def push(self, data, priority):

# Append the data with its priority as a tuple to the queue

self.queue.append((priority, data))

# Sort the queue based on priority (highest priority first)

self.queue.sort(reverse=True, key=lambda x: x[0])

def pop(self):

# Check if the queue is empty before trying to pop

if self.is\_empty():

raise Exception("Priority queue is empty.")

# Pop the item with the highest priority (first item in the sorted list)

return self.queue.pop(0)[1]

def is\_empty(self):

# Return True if the queue is empty, False otherwise

return len(self.queue) == 0

def size(self):

# Return the number of elements in the queue

return len(self.queue)

# Example usage:

pq = PriorityQueue()

pq.push("task1", 1)

pq.push("task2", 3)

pq.push("task3", 2)

print("Size:", pq.size()) # Output: Size: 3

print("Highest priority task:", pq.pop()) # Output: task2

print("Size after pop:", pq.size()) # Output: Size after pop: 2

print("Is empty:", pq.is\_empty()) # Output: Is empty: False

```

### Explanation:

1. `\_\_init\_\_` method: Initializes an empty list to hold the items of the queue.

2. `push` method: Adds an item with a specified priority to the queue. The list is sorted by priority, keeping the highest-priority item at the start.

3. `pop` method: Returns the item with the highest priority and removes it from the queue.

4. `is\_empty` method: Checks if the queue is empty.

5. `size` method: Returns the number of items in the queue.

This structure efficiently maintains the priority order while keeping the highest-priority item at the start for easy access.

### Priority Queue Overview

A priority queue is a specialized data structure that organizes elements based on their priority. Each element in a priority queue has an associated priority, and elements are dequeued based on this priority, not in the order they were added.

In a max-priority queue, the element with the highest priority is removed first, while in a min-priority queue, the element with the lowest priority is removed first.

### Rules of a Priority Queue

1. Elements have priority levels: Each element has an assigned priority.

2. Dequeuing follows priority order: The element with the highest priority is dequeued first.

3. Insertion order is ignored: The order of insertion doesn’t matter; only priority dictates the order of processing.

### Operations on Priority Queue

1. Push (Insert): Adds an element to the queue with a specified priority.

2. Pop (Remove): Removes and returns the element with the highest (or lowest) priority.

3. Is Empty: Checks if the queue has any elements.

4. Size: Returns the number of elements in the queue.

---

### Implementing Priority Queue

#### 1. Using a List (Simple Approach)

In this approach, we use a list of tuples where each tuple contains `(priority, data)`. We can sort the list by priority each time a new element is added to keep the highest priority element at the start.

```python

class PriorityQueue:

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

self.queue = []

def push(self, data, priority):

self.queue.append((priority, data))

self.queue.sort(reverse=True, key=lambda x: x[0])

def pop(self):

if self.is\_empty():

raise Exception("Priority queue is empty.")

return self.queue.pop(0)[1]

def is\_empty(self):

return len(self.queue) == 0

def size(self):

return len(self.queue)

```

#### 2. Using a Linked List

Using a linked list for priority queue allows efficient insertion while maintaining sorted order, as elements can be inserted directly at their correct position based on priority.

```python

class Node:

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

self.data = data

self.priority = priority

self.next = None

class PriorityQueueLinkedList:

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

self.head = None

def push(self, data, priority):

new\_node = Node(data, priority)

if not self.head or self.head.priority < priority:

new\_node.next = self.head

self.head = new\_node

else:

current = self.head

while current.next and current.next.priority >= priority:

current = current.next

new\_node.next = current.next

current.next = new\_node

def pop(self):

if self.is\_empty():

raise Exception("Priority queue is empty.")

data = self.head.data

self.head = self.head.next

return data

def is\_empty(self):

return self.head is None

def size(self):

count = 0

current = self.head

while current:

count += 1

current = current.next

return count

```

#### 3. Using a Heap

Using a heap for implementing a priority queue is highly efficient. The Python `heapq` library provides a min-heap by default. We can invert priorities (multiply by `-1`) to use it as a max-heap.

```python

import heapq

class PriorityQueueHeap:

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

self.heap = []

def push(self, data, priority):

heapq.heappush(self.heap, (-priority, data))

def pop(self):

if self.is\_empty():

raise Exception("Priority queue is empty.")

return heapq.heappop(self.heap)[1]

def is\_empty(self):

return len(self.heap) == 0

def size(self):

return len(self.heap)

```

### Summary

Each implementation has its own trade-offs:

- List: Easy to implement but inefficient for frequent operations due to sorting.

- Linked List: Efficient insertion but has higher overhead for traversal and deletion.

- Heap: Most efficient for maintaining priority order with `O(log n)` complexity for insertion and removal.

Your `PriorityQueue` class implementation looks mostly correct, but there are a few small improvements that can help:

1. In the `pop` method, it’s better to return the data after popping it, so you get the dequeued item.

2. The `size` method should be indented correctly within the class.

Here’s the corrected version of your code:

```python

class PriorityQueue:

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

self.items = []

def push(self, data, priority):

index = 0

while index < len(self.items) and self.items[index][1] <= priority:

index += 1

# Insert the data and priority as a tuple at the correct position

self.items.insert(index, (data, priority))

def is\_empty(self):

return len(self.items) == 0

def pop(self):

if self.is\_empty():

raise IndexError("Priority queue is empty")

# Return the data with the highest priority (first item)

return self.items.pop(0)[0]

def size(self):

return len(self.items)

```

### Explanation of Modifications

1. `pop` method: Now returns the dequeued data (`self.items.pop(0)[0]`), so when you call `pop()`, you get the element with the highest priority.

2. Indentation for `size` method: Corrected to be part of the `PriorityQueue` class.

### Example Usage:

```python

pq = PriorityQueue()

pq.push("task1", 2)

pq.push("task2", 1)

pq.push("task3", 3)

print("Size:", pq.size()) # Output: Size: 3

print("Highest priority task:", pq.pop()) # Output: task3 (priority 3)

print("Size after pop:", pq.size()) # Output: Size after pop: 2

print("Is empty:", pq.is\_empty()) # Output: Is empty: False

```

This implementation will maintain a sorted list by priority (highest priority at the beginning), allowing efficient access to the highest-priority element.

Here’s the complete program with the `PriorityQueue` class, including an example of how to use it:

```python

class PriorityQueue:

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

# Initialize an empty list to hold the queue items, each as (data, priority)

self.items = []

def push(self, data, priority):

# Insert the data at the correct position based on priority

index = 0

while index < len(self.items) and self.items[index][1] <= priority:

index += 1

self.items.insert(index, (data, priority))

def is\_empty(self):

# Return True if the queue is empty, False otherwise

return len(self.items) == 0

def pop(self):

# Check if the queue is empty before attempting to pop

if self.is\_empty():

raise IndexError("Priority queue is empty")

# Remove and return the item with the highest priority

return self.items.pop(0)[0]

def size(self):

# Return the number of elements in the queue

return len(self.items)

# Example usage of PriorityQueue

if \_\_name\_\_ == "\_\_main\_\_":

pq = PriorityQueue()

# Adding items to the priority queue

pq.push("task1", 2)

pq.push("task2", 1)

pq.push("task3", 3)

# Checking the size of the priority queue

print("Size of priority queue:", pq.size()) # Output: 3

# Removing items based on their priority

try:

print("Highest priority task:", pq.pop()) # Output: task3

print("Highest priority task:", pq.pop()) # Output: task1

print("Highest priority task:", pq.pop()) # Output: task2

except IndexError as e:

print(e)

# Checking if the queue is empty

print("Is the priority queue empty?", pq.is\_empty()) # Output: True

# Checking the size after all items are removed

print("Size of priority queue after pops:", pq.size()) # Output: 0

```

### Explanation of the Example Code

1. Adding Items: The `push` method adds items with specified priorities, automatically inserting them into the correct position.

2. Removing Items: The `pop` method retrieves and removes the item with the highest priority.

3. Checking if Empty: `is\_empty` checks if the queue has any items left.

4. Getting the Size: `size` returns the number of items currently in the queue.

This program defines and demonstrates a fully functioning priority queue in Python using a list.

Here’s the complete program with a `Node` class and a `PriorityQueue` class implemented using a singly linked list.

```python

class Node:

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

self.item = item # The data item of the node

self.priority = priority # The priority of the item

self.next = None # The reference to the next node in the linked list

class PriorityQueue:

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

self.start = None # Start reference for the linked list

self.item\_count = 0 # Counter to keep track of the number of items in the queue

def push(self, item, priority):

# Create a new node with the given item and priority

new\_node = Node(item, priority)

# Insert the new node into the linked list based on its priority

if self.is\_empty() or self.start.priority < priority:

# Insert at the beginning if queue is empty or the new node has higher priority

new\_node.next = self.start

self.start = new\_node

else:

# Traverse the list to find the correct position for the new node

current = self.start

while current.next and current.next.priority >= priority:

current = current.next

# Insert the new node at the correct position

new\_node.next = current.next

current.next = new\_node

# Increment the item count

self.item\_count += 1

def pop(self):

# Check if the queue is empty before attempting to pop

if self.is\_empty():

raise Exception("Priority queue is empty")

# Remove the highest-priority item (first node)

item = self.start.item

self.start = self.start.next

# Decrement the item count

self.item\_count -= 1

return item

def is\_empty(self):

# Return True if the queue is empty, False otherwise

return self.start is None

def size(self):

# Return the number of items in the queue

return self.item\_count

# Example usage of PriorityQueue

if \_\_name\_\_ == "\_\_main\_\_":

pq = PriorityQueue()

# Adding items to the priority queue

pq.push("task1", 2)

pq.push("task2", 1)

pq.push("task3", 3)

# Checking the size of the priority queue

print("Size of priority queue:", pq.size()) # Output: 3

# Removing items based on their priority

try:

print("Highest priority task:", pq.pop()) # Output: task3

print("Highest priority task:", pq.pop()) # Output: task1

print("Highest priority task:", pq.pop()) # Output: task2

except Exception as e:

print(e)

# Checking if the queue is empty

print("Is the priority queue empty?", pq.is\_empty()) # Output: True

# Checking the size after all items are removed

print("Size of priority queue after pops:", pq.size()) # Output: 0

```

### Explanation of the Code

1. Node Class:

- `Node` class represents each item in the priority queue, with `item` for the data, `priority` for priority level, and `next` for the reference to the next node.

2. PriorityQueue Class:

- `push`: Inserts a new node at the correct position based on its priority. Higher-priority nodes are placed at the beginning.

- `pop`: Removes and returns the highest-priority node (head of the list).

- `is\_empty`: Checks if the queue is empty.

- `size`: Returns the number of elements in the queue.

### Example Output:

This program creates a priority queue, adds items, retrieves items based on priority, and checks if the queue is empty and its size.