CMPSC 413 - Lab 4

Priority queue, and Heap Sort

Lab Exercises

The priority queue is an abstract data type that contains the following methods:

- insert(item, priorityValue) Inserts item into the priority queue with priority value priorityValue.
- peek() Returns (but does not remove) the item with highest priority in the priority queue.
- delete() Removes and returns the item with highest priority in the priority queue.
- changePriority(item, newPriority) Changes the priority of an item to a new priority value.

Exercise 1

Write down the algorithm and implement a priority queue (both min and max) using an array of elements. Determine the runtime for each of the following:

1. In the worst case, describe the runtime to insert an item into the priority queue.
2. In the worst case, describe the runtime to remove the element with highest priority.
3. In the worst case, describe the runtime to change the priority of an element (find an element and change the priority of the element).

Show an example for each.

Runtime complexities are:

1. Insert: When inserting an item, the worst-case runtime occurs when the item needs to be moved to the root of the heap (due to higher priority) or all the way down to the last level. This results in a time complexity of O(log N), where N is the number of elements in the priority queue.
2. Delete: In the worst case, when you delete an item, you need to perform the heapify operation to maintain the heap property. This operation also has a time complexity of O(log N) as it involves moving elements within the heap.
3. Change: In the worst case, you might need to move an element up to the root or all the way down to the last level, which is also O(log N).

These time complexities assume a balanced heap, and in practice, heaps provide efficient operations for priority queues. However, it's important to note that these are worst-case complexities, and the average-case performance might be better, depending on the distribution of priorities and items.

Exercise 2

Write down the algorithm and implement a priority queue (both min and max) using a linked list of elements. Determine the runtime for each of the following:

1. In the worst case, describe the runtime to insert an item into the priority queue.
2. In the worst case, describe the runtime to remove the element with highest priority.
3. In the worst case, describe the runtime to change the priority of an element.

Show an example for each.

Runtime complexities are:

1. Insert: In the worst case, for insertion, you may need to traverse the entire linked list to find the correct position for the new element. This results in a time complexity of O(N), where N is the number of elements in the priority queue.

2. Delete: Deletion involves updating the head of the linked list. This operation has a time complexity of O(1) in the worst case.
3. Change: Changing the priority involves searching for the element in the linked list and rearranging it if its priority has increased. In the worst case, this operation is O(N) because you may need to traverse the entire list to find the item.

These time complexities reflect the worst-case scenario and assume no additional data structures or optimizations. Linked list implementations are less efficient than heap-based implementations for large priority queues, but they are straightforward to implement and suitable for small-sized priority queues or when dynamic resizing is not a concern.

Exercise 3

Write down the algorithm and implement a priority queue (both min and max) using a heap tree-based data structure (both min and max). Determine the runtime for each of the following:

1. In the worst case, describe the runtime to insert an item into the priority queue.
2. In the worst case, describe the runtime to remove the element with highest priority.
3. In the worst case, describe the runtime to change the priority of an element.

Show an example for each.

Runtime complexities are:

1. Insert: In a binary heap-based implementation, insertion takes O(log N) time in the worst case, where N is the number of elements in the heap. This is because you may need to perform "up-heap" operations to maintain the heap property.
2. Delete: Deletion, which includes finding and removing the element with the highest priority, also takes O(log N) time in the worst case. This is because you need to perform "down-heap" operations to maintain the heap property after removing the root element.
3. Change: Changing the priority of an element involves finding the item in the heap and then possibly performing heap operations. Finding the element is O(N) in the worst case, and then the subsequent heapify operation is O(log N).

Heap-based priority queues are efficient for many practical use cases because they maintain the highest or lowest priority element at the root, allowing quick access to it. The worst-case time complexities described here are based on a balanced heap.

Exercise 4

Tabulate to compare the time complexity of insert, remove and change priority operations for array/linked list/ heap priority queues.

Operation	Array	Linked-List	Min-Heap	Max-Heap
Insertion	O(1) or O(N)	O(N)	O(log N)	O(log N)
Deletion	O(N) or O(1)	O(1)	O(log N)	O(log N)
Changing	O(N)	O(N)	O(N + log N)	O(N + log N)

Exercise 5

Write a paragraph about what have you learnt from this lab exercises?

In summary, the choice of data structure depends on the specific use case and the relative importance of different operations. Arrays and linked lists are straightforward to implement but may not be the most efficient choice for large priority queues. Heap-based structures provide better performance for insert and remove operations, but changing the priority remains relatively expensive. The choice between min-heap and max-heap depends on whether you need the highest or lowest priority element to be easily accessible.

Deliverables

- Codes
- Report with algorithms, screenshots of results for each exercise and conclusion
- Attach the codes as appendix in your Report
- Recorded zoom video (~5 minutes) explaining the difference in implementation of priority queues using different data structures and also demonstrate the working of the program.

Code

Exercise 1

```
class PriorityQueue:
    def __init__(self, is_max_heap=False):
        self.elements = []
        self.is_max_heap = is_max_heap

    def insert(self, item, priorityValue):
        self.elements.append((item, priorityValue))
        self._heapify_up(len(self.elements) - 1)

    def _heapify_up(self, idx):
        while idx > 0:
            parent_idx = (idx - 1) // 2
            if (self.elements[idx][1] > self.elements[parent_idx][1]) if
self.is_max_heap else (self.elements[idx][1] < self.elements[parent_idx][1]):
                self.elements[idx], self.elements[parent_idx] =
self.elements[parent_idx], self.elements[idx]
                idx = parent_idx
            else:
                break

    def peek(self):
        if not self.elements:
            return None
        return self.elements[0][0]

    def delete(self):
        if not self.elements:
            return None
        if len(self.elements) == 1:
            return self.elements.pop()[0]
        root = self.elements[0][0]
        self.elements[0] = self.elements.pop()
```

```
        self._heapify_down(0)
        return root

    def _heapify_down(self, idx):
        while True:
            left_child_idx = 2 * idx + 1
            right_child_idx = 2 * idx + 2
            largest = idx

            if left_child_idx < len(self.elements) and
((self.elements[left_child_idx][1] > self.elements[largest][1]) if self.is_max_heap
else (self.elements[left_child_idx][1] < self.elements[largest][1])):
                largest = left_child_idx

            if right_child_idx < len(self.elements) and
((self.elements[right_child_idx][1] > self.elements[largest][1]) if self.is_max_heap
else (self.elements[right_child_idx][1] < self.elements[largest][1])):
                largest = right_child_idx

            if largest != idx:
                self.elements[idx], self.elements[largest] = self.elements[largest],
self.elements[idx]
                idx = largest
            else:
                break

    def changePriority(self, item, newPriority):
        for i in range(len(self.elements)):
            if self.elements[i][0] == item:
                old_priority = self.elements[i][1]
                self.elements[i] = (item, newPriority)
                if (newPriority > old_priority) if self.is_max_heap else
(newPriority < old_priority):
                    self._heapify_down(i)
                else:
                    self._heapify_up(i)

# Example usage
min_priority_queue = PriorityQueue()
max_priority_queue = PriorityQueue(is_max_heap=True)

# Insertion in both queues
min_priority_queue.insert('A', 5)
min_priority_queue.insert('B', 3)
min_priority_queue.insert('C', 7)
max_priority_queue.insert('X', 10)
max_priority_queue.insert('Y', 8)
max_priority_queue.insert('Z', 12)

# Peek and delete operations
print("Min Priority Queue:")
print(min_priority_queue.peek())  # Should print 'B' (highest priority)
```

```
print(min_priority_queue.delete())  # Should print 'B' (highest priority)
print(min_priority_queue.delete())  # Should print 'A'
print(min_priority_queue.peek())  # Should print 'C' (highest priority)

print("\nMax Priority Queue:")
print(max_priority_queue.peek())  # Should print 'Z' (highest priority)
print(max_priority_queue.delete())  # Should print 'Z' (highest priority)
print(max_priority_queue.delete())  # Should print 'X'
print(max_priority_queue.peek())  # Should print 'Y' (highest priority)

# Changing priority
min_priority_queue.changePriority('C', 1)
print("Changed Priority in Min Priority Queue:")
print(min_priority_queue.peek())  # Should print 'C' (highest priority)

max_priority_queue.changePriority('Y', 15)
print("\nChanged Priority in Max Priority Queue:")
print(max_priority_queue.peek())  # Should print 'Y' (highest priority)
```

Exercise 2

```
class Node:
    def __init__(self, item, priorityValue):
        self.item = item
        self.priorityValue = priorityValue
        self.next = None

class PriorityQueue:
    def __init__(self, is_max_heap=False):
        self.head = None
        self.is_max_heap = is_max_heap

    def insert(self, item, priorityValue):
        new_node = Node(item, priorityValue)
        if not self.head:
            self.head = new_node
        else:
            if (priorityValue > self.head.priorityValue) if self.is_max_heap else
(priorityValue < self.head.priorityValue):
                new_node.next = self.head
                self.head = new_node
            else:
                current = self.head
                while current.next and ((priorityValue <=
current.next.priorityValue) if self.is_max_heap else (priorityValue >=
current.next.priorityValue)):
                    current = current.next
                new_node.next = current.next
                current.next = new_node
```

```
    def peek(self):
        if not self.head:
            return None
        return self.head.item

    def delete(self):
        if not self.head:
            return None
        item = self.head.item
        self.head = self.head.next
        return item

    def changePriority(self, item, newPriority):
        if not self.head:
            return
        if (newPriority > self.head.priorityValue) if self.is_max_heap else
(newPriority < self.head.priorityValue):
            return
        if self.head.item == item:
            self.head.priorityValue = newPriority
            return
        current = self.head
        while current.next and current.next.item != item:
            current = current.next
        if current.next:
            current.next.priorityValue = newPriority
            current = self.head
            while current.next and ((newPriority <= current.next.priorityValue) if
self.is_max_heap else (newPriority >= current.next.priorityValue)):
                current = current.next
            if current.next != current:
                temp = current.next
                current.next = temp.next
                temp.next = self.head
                self.head = temp

# Example usage
min_priority_queue = PriorityQueue()
max_priority_queue = PriorityQueue(is_max_heap=True)

# Insertion in both queues
min_priority_queue.insert('A', 5)
min_priority_queue.insert('B', 3)
min_priority_queue.insert('C', 7)
max_priority_queue.insert('X', 10)
max_priority_queue.insert('Y', 8)
max_priority_queue.insert('Z', 12)

# Peek and delete operations
print("Min Priority Queue:")
print(min_priority_queue.peek())  # Should print 'B' (highest priority)
print(min_priority_queue.delete())  # Should print 'B' (highest priority)
```

```
print(min_priority_queue.delete())  # Should print 'A'
print(min_priority_queue.peek())  # Should print 'C' (highest priority)

print("\nMax Priority Queue:")
print(max_priority_queue.peek())  # Should print 'Z' (highest priority)
print(max_priority_queue.delete())  # Should print 'Z' (highest priority)
print(max_priority_queue.delete())  # Should print 'X'
print(max_priority_queue.peek())  # Should print 'Y' (highest priority)

# Changing priority
min_priority_queue.changePriority('C', 1)
print("Changed Priority in Min Priority Queue:")
print(min_priority_queue.peek())  # Should print 'C' (highest priority)

max_priority_queue.changePriority('Y', 15)
print("\nChanged Priority in Max Priority Queue:")
print(max_priority_queue.peek())  # Should print 'Y' (highest priority)
```

Exercise 3

```
import heapq

class PriorityQueue:
    def __init__(self, is_max_heap=False):
        self.heap = []
        self.is_max_heap = is_max_heap

    def insert(self, item, priorityValue):
        if self.is_max_heap:
            priorityValue = -priorityValue
        heapq.heappush(self.heap, (priorityValue, item))

    def peek(self):
        if not self.heap:
            return None
        if self.is_max_heap:
            return self.heap[0][1]
        return self.heap[0][1]

    def delete(self):
        if not self.heap:
            return None
        _, item = heapq.heappop(self.heap)
        return item

    def changePriority(self, item, newPriority):
        if self.is_max_heap:
            newPriority = -newPriority

        # Find the item in the heap
```

```
        for i, (priority, existing_item) in enumerate(self.heap):
            if existing_item == item:
                self.heap[i] = (newPriority, item)
                heapq.heapify(self.heap)  # Rebuild the heap

# Example usage
min_priority_queue = PriorityQueue()
max_priority_queue = PriorityQueue(is_max_heap=True)

# Insertion in both queues
min_priority_queue.insert('A', 5)
min_priority_queue.insert('B', 3)
min_priority_queue.insert('C', 7)
max_priority_queue.insert('X', 10)
max_priority_queue.insert('Y', 8)
max_priority_queue.insert('Z', 12)

# Peek and delete operations
print("Min Priority Queue:")
print(min_priority_queue.peek())  # Should print 'B' (highest priority)
print(min_priority_queue.delete())  # Should print 'B' (highest priority)
print(min_priority_queue.delete())  # Should print 'A'
print(min_priority_queue.peek())  # Should print 'C' (highest priority)

print("\nMax Priority Queue:")
print(max_priority_queue.peek())  # Should print 'Z' (highest priority)
print(max_priority_queue.delete())  # Should print 'Z' (highest priority)
print(max_priority_queue.delete())  # Should print 'X'
print(max_priority_queue.peek())  # Should print 'Y' (highest priority)

# Changing priority
min_priority_queue.changePriority('C', 1)
print("Changed Priority in Min Priority Queue:")
print(min_priority_queue.peek())  # Should print 'C' (highest priority)

max_priority_queue.changePriority('Y', 15)
print("\nChanged Priority in Max Priority Queue:")
print(max_priority_queue.peek())  # Should print 'Y' (highest priority)
```