Custom Classes

```
"""
Comparater()
    takes in the ordering scheme of the priority queue and compares
elements    based on if the invoking priority queue is a max or min
queue.
"""

class Comparator():
    # returns a function to be used in lambda based on the ordering
scheme of the priority queue
    def __init__(self, is_min=True): # by default sorts by min
priority first
        self.is_min = is_min

    def compare(self, a,b):
        if self.is_min:
            return a < b
        return a > b

class PriorityQueueElement():
    def __init__(self, priority, value):
        self.p = priority
        self.v = value
    def __repr__(self):
        return f"[{self.p} , {self.v}]"
```

- comparator(): is used in each of the PriorityQueImplementations to quickly compare objects. If we have a min priority queue, we will pass is_min = True to the Comparator() instance within a given Priority Queue class and it will select the appropriate comparison operator to use for 2 elements

- PriorityQueueElement(): the basic object used to represent an element in the priority queue. It is read (priority, element).

Demonstration

```
import random

from base_classes import PriorityQueueElement, Comparator

from priority_queue_array import PriorityQueueArray
from priority_queue_linked_list import PriorityQueueLinkedList
from priority_queue_heap_array import PriorityQueueHeap
from heap_sort import heap_sort

    # Demonstration

    # min heap
alpha = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
'y', 'z']

# test elements
elements = [(random.randint(0,100), alpha[i]) for i in range(5)]
print(f"Unordered Array of Elements: { elements }\n")

    # Exercise 1 - priority queue array
print("\nArray Implementation")
priority_queue_array = PriorityQueueArray(is_min=True)

# inserting elements
for element in elements:
    priority_queue_array.insert(element[0], element[1])

# Elements now in heap
print(priority_queue_array.pq)
```

```
# removing largest element
print(priority_queue_array.delete())

# changing an elements priority
target_v = priority_queue_array.pq[0].v
target_p = priority_queue_array.pq[0].p

print(f"Changing Priority of { target_p, target_v }")
priority_queue_array.change_priority(new_priority=100,
val=target_v)
print(priority_queue_array.pq)


    # Exercise 2 - linked list
print("\nLinked List Implementation")
priority_queue_linked_list = PriorityQueueLinkedList()

# inserting elements
for element in elements:
    priority_queue_linked_list.insert(element[0], element[1])

# removing largest element
print(priority_queue_linked_list.delete())

# changing an elements priority
target_v = priority_queue_linked_list.head.element.v
target_p = priority_queue_linked_list.head.element.p

print(f"Changing Priority of { target_p, target_v }")
priority_queue_linked_list.change_priority(new_priority=100,
val=target_v)

# checking the final element of the linked list
cur = priority_queue_linked_list.head

while cur.next:
```

```
    cur = cur.next

print(f"Now moved to the end of the priority Queue: {cur.element}")


    # Exercise 3 - Heap Tree (Array Based)
print("\nHeap Implementation")
priority_queue_heap = PriorityQueueHeap(is_min=True)

# inserting elements
for element in elements:
    priority_queue_heap.insert(element[0], element[1])

# printing elements in heap
print(priority_queue_heap.pq)

# changing elements priority
# changing an elements priority
target = priority_queue_heap.pq[0]
target_p = priority_queue_heap.pq[0].p

print(f"Changing Priority of { target.p, target.v }")
priority_queue_heap.change_priority(new_priority=100,
element=target)
priority_queue_heap.bubble_down()
print(f"Last Element is now: {priority_queue_heap.pq}")

    # Exercise 4 - Heap Sort
print("\nHeap Sort")

print(f"unordered elements: {elements}")
print(f"sorted elements using heap-sort (normal order): {
heap_sort(elements)}")
print(f"sorted elements using heap-sort (Reverse order): {
heap_sort(elements, reverse=True)}")
```

```
    # Max Heap

# Exercise 1 - priority queue array
print("\nArray Implementation")
priority_queue_array = PriorityQueueArray(is_min=False)

# inserting elements
for element in elements:
    priority_queue_array.insert(element[0], element[1])

# Elements now in heap
print(priority_queue_array.pq)

# removing largest element
print(priority_queue_array.delete())

# changing an elements priority
target_v = priority_queue_array.pq[0].v
target_p = priority_queue_array.pq[0].p

print(f"Changing Priority of { target_p, target_v }")
priority_queue_array.change_priority(new_priority=100,
val=target_v)
print(priority_queue_array.pq)


    # Exercise 2 - linked list
print("\nLinked List Implementation")
priority_queue_linked_list = PriorityQueueLinkedList(is_min=False)

# inserting elements
for element in elements:
    priority_queue_linked_list.insert(element[0], element[1])

# removing largest element
print(priority_queue_linked_list.delete())
```

```python
# changing an elements priority
target_v = priority_queue_linked_list.head.element.v
target_p = priority_queue_linked_list.head.element.p

print(f"Changing Priority of { target_p, target_v }")
priority_queue_linked_list.change_priority(new_priority=100,
val=target_v)

# checking the final element of the linked list
cur = priority_queue_linked_list.head

while cur.next:
    cur = cur.next

print(f"Now moved to the end of the priority Queue: {cur.element}")


    # Exercise 3 - Heap Tree (Array Based)
print("\nHeap Implementation")
priority_queue_heap = PriorityQueueHeap(is_min=False)

# inserting elements
for element in elements:
    priority_queue_heap.insert(element[0], element[1])

# printing elements in heap
print(priority_queue_heap.pq)

# changing elements priority
# changing an elements priority
target = priority_queue_heap.pq[0]
target_p = priority_queue_heap.pq[0].p

print(f"Changing Priority of { target.p, target.v }")
priority_queue_heap.change_priority(new_priority=100,
element=target)
```

```
priority_queue_heap.bubble_down()
print(f"Last Element is now: {priority_queue_heap.pq}")

    # Exercise 4 - Heap Sort
print("\nHeap Sort")

print(f"unordered elements: {elements}")
print(f"sorted elements using heap-sort (normal order): {
heap_sort(elements)}")
print(f"sorted elements using heap-sort (Reverse order): {
heap_sort(elements, reverse=True)}")
```

Output

```
C:\Users\aaron\AppData\Local\Programs\Python\Python39\python.exe
C:\Users\aaron\Projects\classWork\CMPSC413\lab4\main.py
Unordered Array of Elements: [(39, 'a'), (61, 'b'), (37, 'c'), (25,
'd'), (2, 'e')]



Array Implementation
[[2 , e], [25 , d], [37 , c], [39 , a], [61 , b]]
[2 , e]
Changing Priority of (25, 'd')
[[25 , d], [37 , c], [39 , a], [61 , b], [100 , d]]

Linked List Implementation
None
Changing Priority of (25, 'd')
Now moved to the end of the priority Queue: [100 , d]

Heap Implementation
[[2 , e], [25 , d], [39 , a], [61 , b], [37 , c]]
```

```
Changing Priority of (2, 'e')
Last Element is now: [[25 , d], [37 , c], [39 , a], [61 , b], [100
, e]]

Heap Sort
unordered elements: [(39, 'a'), (61, 'b'), (37, 'c'), (25, 'd'),
(2, 'e')]
sorted elements using heap-sort (normal order): [[2 , e], [25 , d],
[37 , c], [39 , a], [61 , b]]
sorted elements using heap-sort (Reverse order): [[61 , b], [39 ,
a], [37 , c], [25 , d], [2 , e]]

Array Implementation
[[61 , b], [39 , a], [37 , c], [25 , d], [2 , e]]
[61 , b]
Changing Priority of (39, 'a')
[[100 , a], [39 , a], [37 , c], [25 , d], [2 , e]]

Linked List Implementation
None
Changing Priority of (39, 'a')
Now moved to the end of the priority Queue: [2 , e]

Heap Implementation
[[61 , b], [39 , a], [37 , c], [25 , d], [2 , e]]
Changing Priority of (61, 'b')
Last Element is now: [[100 , b], [39 , a], [37 , c], [25 , d], [2 ,
e]]

Heap Sort
unordered elements: [(39, 'a'), (61, 'b'), (37, 'c'), (25, 'd'),
(2, 'e')]
```

```
sorted elements using heap-sort (normal order): [[2 , e], [25 , d],
[37 , c], [39 , a], [61 , b]]
sorted elements using heap-sort (Reverse order): [[61 , b], [39 ,
a], [37 , c], [25 , d], [2 , e]]

Process finished with exit code 0
```

Exercise 1: Array

Code

```
from base_classes import Comparator, PriorityQueueElement


class PriorityQueueArray():
    def __init__(self):
        self.pq = []
        self.is_min = True

        self.cp = Comparator(self.is_min)

    def is_empty(self):
        return True if self.pq is None else False

    def insert(self, priority, value):
        new_element = PriorityQueueElement(priority, value)

        if self.is_empty():
            self.pq.append(new_element)

        else:
            idx_found = False
```

```
            for i in range(len(self.pq)):   # iterate over elements
                if self.cp.compare(priority, self.pq[i].p): # 
compare using chose priority order of queue
                    self.pq.insert(i, new_element)
                    idx_found = True
                    break
            if idx_found == False:
                self.pq.append(new_element)

    def peek(self):
        if not self.is_empty():
            return self.pq[0]

    def delete(self):
        if not self.is_empty():
            del self.pq[0]

    def change_priority(self, new_priority, val):
        # finds target, deletes it and creates a new one with the 
updated priority
        element_idx = None

        for i in range(len(self.pq)):
            if self.pq[i].val == val:
                element = self.pq[i]
                break

        if element_idx:
            del self.pq[element_idx]  # drop target from the list

        self.insert(new_priority, val) # add it back as a new 
target with different priority
```

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.

in the worst case scenario, we can expect an $O(n)$ time complexity in my implementation. This is due to us having to traverse the entire array to find the correct spot for the new element, specifically in the case for which the new element has the lowest priority

2. In the worst case, describe the runtime to remove the element with highest priority.

in my implementation, the run time for any element is $O(1)$, this is because my insertion function handles the tasks of maintaining the heap property. This allows delete() (or pop()) to pop the root which will always be the max priority element.

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).

in the worst case, we do a linear search across the entire array taking $O(n)$ time.

Exercise 2: Linked-List

Code

```
from base_classes import Comparator, PriorityQueueElement

# takes in elements of type PriorityQueueElement
class Node():
    def __init__(self, element, next=None):
        self.element = element
        self.next = next
```

```
class PriorityQueueLinkedList():
    def __init__(self, head=None, is_min = True):
        self.head = head
        self.is_min = is_min
        self.cp = Comparator(self.is_min)

    def is_empty(self):
        return self.head == None

    def insert(self, priority, val):
        new_node = Node(PriorityQueueElement(priority, val))

        # if ll is empty or new node will replace the current
head...
        if self.is_empty() or self.cp.compare(priority,
self.head.element.p):
            new_node.next = self.head
            self.head = new_node

        # otherwise, follow normal insertion procedure
        else:
            cur = self.head # set cur to loop over ll starting from
head
            while cur.next and not self.cp.compare(priority,
cur.next.element.p): # find correct position
                cur = cur.next

            # perform insertion
            new_node.next = cur.next
            cur.next = new_node


    def peek(self):
        # gets the first target in the linked list
        if self.head:
            return self.head.element
```

```python
    def delete(self):
        # removes the target with the highest priority (the current
head)
        if self.head:
            self.head = self.head.next


    def change_priority(self, new_priority, val):
        # remove the old node first
        cur = self.head
        prev = None

        while cur and cur.element.v != val:
            prev = cur
            cur = cur.next

        if cur:
            if prev:
                prev.next = cur.next
            else:
                self.head = cur.next

            #  insert the new node with updated priority
            self.insert(new_priority, val)
```

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.

this will take $O(n)$ Time in the worst case - an insertion at the end of the list

2. In the worst case, describe the runtime to remove the element with highest priority.

this will take $O(1)$ time as the root element will always be at the head position of the linked list.

3. In the worst case, describe the runtime to change the priority of an element.

this will take $O(n)$ time at worst since this involves finding the element within the linked list first and then performing a priority change by reinserting the element. At worst, we may take the last element and change its priority to be equal to what it was before, resulting in an $O(2n) = O(n)$ operation.

Exercise-3: Heap (Using Array Implementation)

Code

```
from base_classes import PriorityQueueElement, Comparator

class PriorityQueueHeap():
    def __init__(self, is_min=True):
        self.is_min = is_min # priority ordering set to min by
default
        self.pq = [] # list based implementation
        self.cp = Comparator(is_min).compare

    def is_empty(self):
        return len(self.pq) == 0

    def get_r_child_idx(self, parent_idx):
        r_child_idx = (2 * parent_idx) + 2
```

```
        if r_child_idx < len(self.pq):
            return r_child_idx
        return None

    def get_l_child_idx(self, parent_idx):
        l_child_idx = (2 * parent_idx) + 1

        if l_child_idx < len(self.pq):
            return l_child_idx
        return None

    def get_parent_idx(self, element_idx): # using O(logn) time to
find element (best case)
        return (element_idx - 1) // 2

    # finds the index of a target
    def get_element_idx(self, target):
        for i, element in enumerate(self.pq): # unpacks the element
object
            if element == target:
                return i
        return None  # if not found

    def insert(self, priority, value):
        element = PriorityQueueElement(priority=priority,
value=value) # make a new element object
        self.pq.append(element) # throw it in the array
        self.bubble_up(len(self.pq) - 1) # bubble it up to the
correct position

    def bubble_up(self, start_idx=0):
        # start bubbling up from the specified index
        cur_idx = start_idx

        while cur_idx > 0:  # stop if node is at the root
            parent_idx = self.get_parent_idx(cur_idx)
```

```
            # check if the heap property is violated and swap if it
is
            if self.cp(self.pq[cur_idx].p, self.pq[parent_idx].p):
                self.pq[cur_idx], self.pq[parent_idx] =
self.pq[parent_idx], self.pq[cur_idx]
                cur_idx = parent_idx  # move up to the parent index
            else:
                break  # stop if the heap property is not violated

    def bubble_down(self, cur_idx=0):
        # keep within bounds of array
        while cur_idx < len(self.pq):
            l_idx = self.get_l_child_idx(cur_idx)
            r_idx = self.get_r_child_idx(cur_idx)
            swap_idx = None

            # find which child to swap with
            if l_idx is not None and (r_idx is None or
self.cp(self.pq[l_idx].p, self.pq[r_idx].p)): # max priority left
                swap_idx = l_idx
            elif r_idx is not None: # max priority right
                swap_idx = r_idx

            # do the swap if we need to
            if swap_idx is not None and not
self.cp(self.pq[cur_idx].p, self.pq[swap_idx].p):
                self.pq[cur_idx], self.pq[swap_idx] =
self.pq[swap_idx], self.pq[cur_idx]
                cur_idx = swap_idx
            else:
                break # don't need to swap so we're done

    def get_root(self):
        return self.pq[0]

    def peek(self):
        # will return the element at the root position
```

```
        if not self.is_empty():
            return self.pq[0]

    def delete(self): # functions like pop() on the heap
        if self.is_empty():
            return None

        root = self.get_root()
        last_element = self.pq.pop()  # remove the last element

        if not self.is_empty():  # check if the heap is not empty
after removing the last element
            self.pq[0] = last_element  # place the last element at
the root
            self.bubble_down(0)  # heapify the heap

        return root

    def change_priority(self, element, new_priority):
        element_idx = self.get_element_idx(element) # finding the
element in the pq
        self.pq[element_idx].p = new_priority # updating the
priority in place

        if element_idx: # if element exists
            # see if we need to bubble up or down            #
bubble up - new priority is higher than old            # true when
new_priority > cur_priority and we have max or new_priority <
cur_priority and we have min            if self.is_min and
new_priority < element.p:
                self.bubble_up(start_idx=element_idx)
            elif not self.is_min and new_priority > element.p:
                self.bubble_up(start_idx=element_idx)
            else:
                self.bubble_down()
```

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.

the worst case insertion runtime would be $O(log(n))$. This is due to the fact that my heap implementation functions essentially as a binary tree. The worst case would be a leaf element needing to be bubbled up to the root, requiring at most $log(n)$ swaps.

2. In the worst case, describe the runtime to remove the element with highest priority.

In all cases, removing the element with the highest priority will take $O(1)$ time. This is due to my insert() function guaranteeing that the heap property is always maintained over the array.

3. In the worst case, describe the runtime to change the priority of an element.

In the worst case, Changing the priority would mean replacing a leaf element (lowest priority element) with one of the same priority. We'd then do at most $O(2log(n))$ operations which is still $O(log(n))$.

Exercise 4: Heap Sort

code

```
def heap_sort(arr=None, reverse=False):
    is_min = True
```

```
    # if we want to sort in descending order (reversed, we can set
our heap to be a maxheap)
    if reverse:
        is_min = False

    heap = PriorityQueueHeap(is_min=is_min)
    sorted_array = []

    # add elements to the heap
    for element in arr:
        heap.insert(element[0], element[1])

    # root contains highest priority element -> pop into list
    while not heap.is_empty():
        root_element = heap.delete()

        if root_element is not None:
            sorted_array.append(root_element)

    return sorted_array
```

Write down the algorithm and implement a heap sort (both ascending and descending) using heap data structure. Determine the runtime for each of the following.

1. In the worst case, describe the runtime to sort in ascending order.

In the worst case, we'd have a time complexity of $O(nlogn)$. This is because we can build the heap using $O(logn)$ time. Then we must go through the process of continuously popping n-elements from the heap resulting in the $O(nlogn)$ time complexity.

2. In the worst case, describe the runtime to sort in descending order.

This is the same as for ascending order since my implementation uses the Comparator() object.

Exercise 5: Time Complexities

Operation	Array-Based Priority Queue	Linked List-Based Priority Queue	Heap-Based Priority Queue (Binary Tree)
Insert	O(n)	O(n)	O(log n)
Remove (e.g., remove max/min)	O(n)	O(n)	O(log n)
Change Priority	O(n)	O(n)	O(log n) (worst case)

Exercise 6: Conclusion

I have thoroughly learned how to implement a heap using various different data structures. It is clear that the Binary Tree implementation is superior since we can exploit branch pruning to reduce the time complexity across the board compared to the simple array and linked list implementations. This is due to the fact that we make simple linear traversals over the list based implementations vs. binary search.