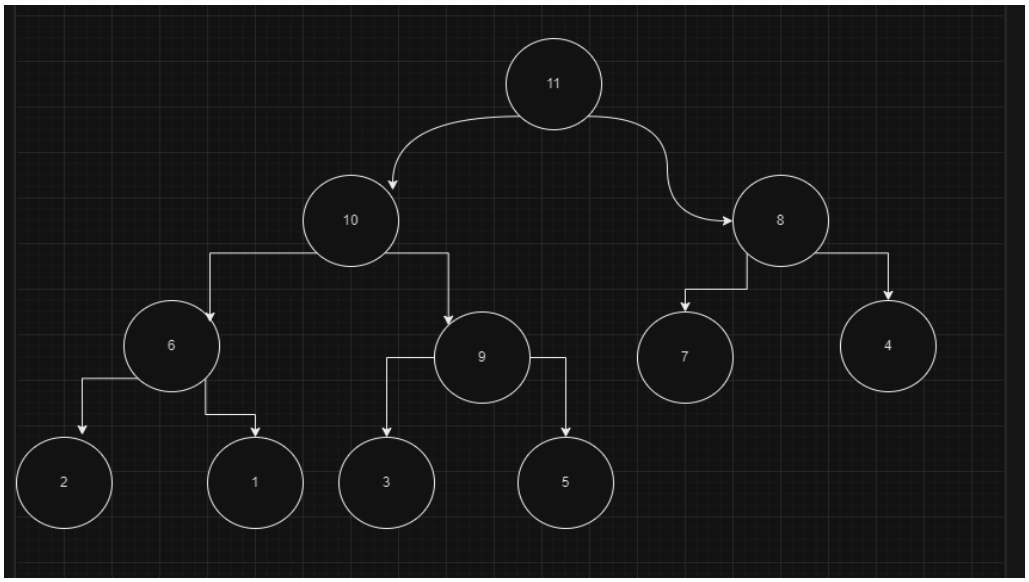
Estrutura de dados e algoritimos avançados

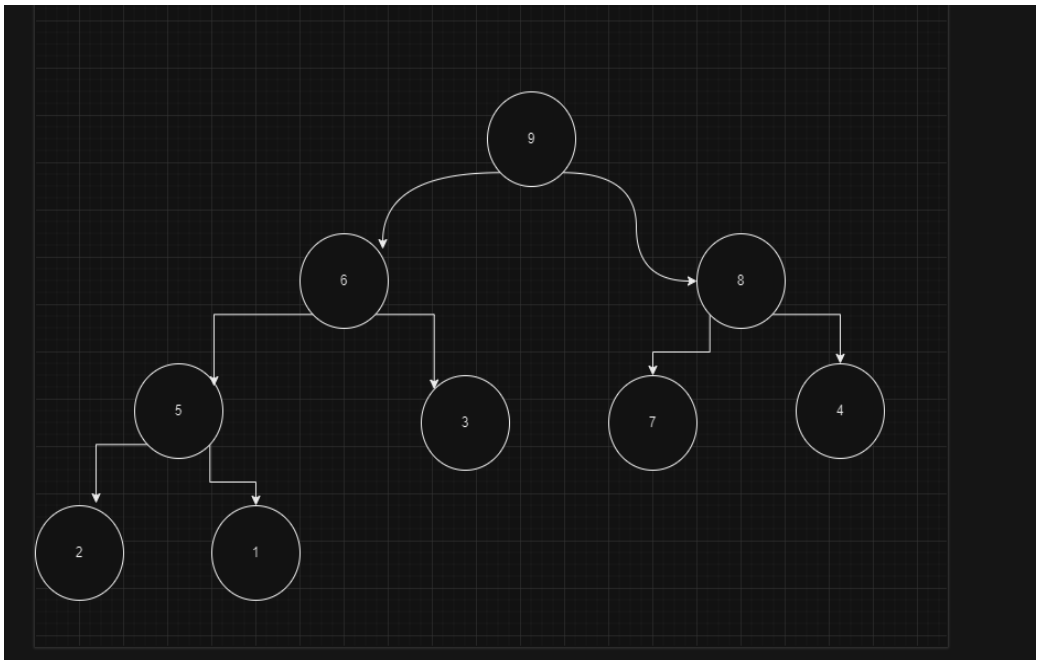
Nome: Ian costa dos Santos

Turma: 24E1_2

1. **Desenhe uma representação do heap abaixo após a inserção do valor 11:**



2. **Desenhe uma representação do heap abaixo após a remoção do nó raiz:**



3. **Imagine que você construiu um heap, inserindo nele os seguintes números nesta ordem: 55, 22, 34, 10, 2, 99, 68. Se você removesse os números do heap, um de cada vez, e os inserisse em um array (sempre no fim), em que ordem os números ficariam?**

Se inserirmos os números 55, 22, 34, 10, 2, 99, 68 em um heap e depois os removermos um por um e os inserirmos em um array no final, a ordem dos números no array será:

2, 10, 22, 34, 55, 68, 99
Isso ocorre porque um heap binário é uma estrutura de dados que mantém a propriedade de heap, onde o pai é sempre maior ou menor que os filhos, dependendo se é um heap máximo ou mínimo.

4. **Com base no resultado da questão anterior, descreva um algoritmo de ordenação baseado em um heap. Analise a complexidade desse algoritmo e o compare (em relação a este aspecto) com os algoritmos de ordenação que discutimos no trimestre passado.**

Um algoritmo de ordenação baseado em heap, como o HeapSort, funciona da seguinte maneira:

Construir um heap máximo (ou mínimo) a partir do array desordenado.
Trocar o primeiro elemento (o maior no caso de um heap máximo) com o último elemento do heap não ordenado.
Reduzir o tamanho do heap não ordenado em 1.
Restaurar a propriedade de heap (max-heapify ou min-heapify, dependendo do tipo de heap) no topo do heap não ordenado.
Repetir os passos 2-4 até que o heap não ordenado tenha tamanho 1.

5. **Um heap vazio tem 0 níveis. Um heap com 1 item possui 1 nível. De um modo geral, o número de níveis de um heap é 1 a mais que o nível do nó folha mais profundo. Implemente um método “niveis” na classe Heap apresentada em aula, que retorna o nº de níveis no heap. Seu método deve ter complexidade O(log N).**

```python
class Heap:
    def __init__(self):
        self.heap_list = []

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

    def remove(self):
        if len(self.heap_list) == 0:
            raise IndexError("Heap is empty")
        last_element = self.heap_list.pop()
        if len(self.heap_list) > 0:
            removed_item = self.heap_list[0]
            self.heap_list[0] = last_element
            self._heapify_down(0)
            return removed_item
        return last_element

    def _heapify_up(self, index):
        parent_index = (index - 1) // 2
        while index > 0 and self.heap_list[index] > self.heap_list[parent_index]:
            self.heap_list[index], self.heap_list[parent_index] = (
                self.heap_list[parent_index],
                self.heap_list[index],
            )
            index = parent_index
            parent_index = (index - 1) // 2
```

```
    def _heapify_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        largest = index
        if (
            left_child_index < len(self.heap_list)
            and self.heap_list[left_child_index] > self.heap_list[largest]
        ):
            largest = left_child_index
        if (
            right_child_index < len(self.heap_list)
            and self.heap_list[right_child_index] > self.heap_list[largest]
        ):
            largest = right_child_index
        if largest != index:
            self.heap_list[index], self.heap_list[largest] = (
                self.heap_list[largest],
                self.heap_list[index],
            )
            self._heapify_down(largest)

    def niveis(self):
        n = len(self.heap_list)
        return n.bit_length() if n > 0 else 0

heap = Heap()
heap.insert(10)
heap.insert(5)
heap.insert(15)
heap.insert(3)
heap.insert(8)
print("Número de níveis:", heap.niveis())
```

Saida:

```
AzureAD+IanSantos@BRALFSUSABIT03 MINGW64 ~/Documents/faculdade/python
$ C:/Users/IanSantos/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/IanSantos/Documents/faculdade/python/dados-e-algoritimos/tp1/q5.py
Número de níveis: 3
```

6. **Considere que uma aplicação irá processar registros de pacientes que dão entrada na emergência de um hospital. Cada registro conterá os seguintes dados do paciente correspondente: Nº sequencial de entrada na fila (int), Nome (str), Prioridade (int). Na aplicação, os registros serão representados como dicionários, e armazenados em uma lista conforme o exemplo apresentado abaixo.**
   Implemente uma fila de prioridade baseada em heap para controlar a ordem de atendimento dos pacientes na emergência, priorizando os pacientes com maior gravidade. Entre pacientes com a mesma gravidade, deve ser adotada uma política FIFO, isto é, quem tiver dado entrada antes deve ser atendido primeiro.

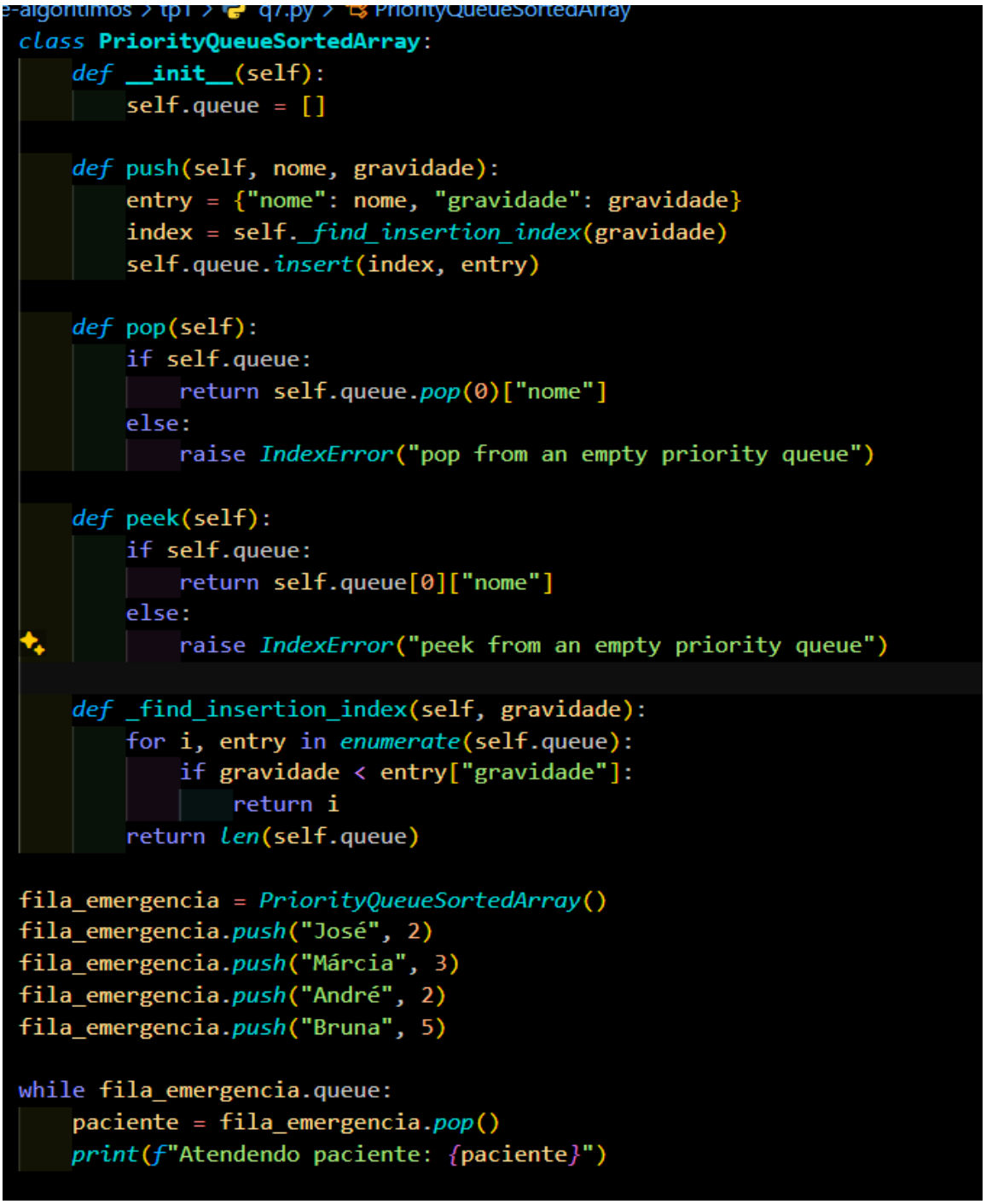
```python
class PriorityQueue:
    def __init__(self):
        self.heap = []
        self.seq_count = 0

    def push(self, nome, gravidade):
        entry = (-gravidade, self.seq_count, nome)
        self.seq_count += 1
        self.heap.append(entry)
        self._sift_up(len(self.heap) - 1)

    def pop(self):
        if len(self.heap) > 0:
            top_priority = self.heap[0]
            last_entry = self.heap.pop()
            if len(self.heap) > 0:
                self.heap[0] = last_entry
                self._sift_down(0)
            return top_priority[-1]
        else:
            raise IndexError("pop from an empty priority queue")

    def peek(self):
        if len(self.heap) > 0:
            return self.heap[0][-1]
        else:
            raise IndexError("peek from an empty priority queue")

    def _sift_up(self, index):
        while index > 0:
            parent_index = (index - 1) // 2
            if self.heap[parent_index] > self.heap[index]:
                self.heap[parent_index], self.heap[index] = self.heap[index], self.heap[parent_index]
                index = parent_index
            else:
                break
```

```python
    def _sift_down(self, index):
        max_index = len(self.heap) - 1
        while True:
            left_child_index = 2 * index + 1
            right_child_index = 2 * index + 2
            min_index = index

            if left_child_index <= max_index and self.heap[left_child_index] < self.heap[min_index]:
                min_index = left_child_index
            if right_child_index <= max_index and self.heap[right_child_index] < self.heap[min_index]:
                min_index = right_child_index

            if min_index != index:
                self.heap[min_index], self.heap[index] = self.heap[index], self.heap[min_index]
                index = min_index
            else:
                break

# Exemplo de uso
fila_emergencia = PriorityQueue()
fila_emergencia.push("José", 2)
fila_emergencia.push("Márcia", 3)
fila_emergencia.push("André", 2)
fila_emergencia.push("Bruna", 5)

while len(fila_emergencia.heap) > 0:
    paciente = fila_emergencia.pop()
    print(f"Atendendo paciente: {paciente}")
```

Saida:

```
AzureAD+IanSantos@BRALFSUSABIT03 MINGW64 ~/Documents/faculdade/python
$ C:/Users/IanSantos/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/IanSantos/Documents/faculdade/python/dados-e-algoritimos/tp1/q6.py
Atendendo paciente: Bruna
Atendendo paciente: Márcia
Atendendo paciente: José
Atendendo paciente: André
```

7. **Ainda no contexto da questão anterior, reimplemente a fila de prioridade especificada, desta vez baseada em um array ordenado.**



```
e-algoritmos > tp1 > q7.py > PriorityQueueSortedArray
class PriorityQueueSortedArray:
    def __init__(self):
        self.queue = []

    def push(self, nome, gravidade):
        entry = {"nome": nome, "gravidade": gravidade}
        index = self._find_insertion_index(gravidade)
        self.queue.insert(index, entry)

    def pop(self):
        if self.queue:
            return self.queue.pop(0)["nome"]
        else:
            raise IndexError("pop from an empty priority queue")

    def peek(self):
        if self.queue:
            return self.queue[0]["nome"]
        else:
            raise IndexError("peek from an empty priority queue")

    def _find_insertion_index(self, gravidade):
        for i, entry in enumerate(self.queue):
            if gravidade < entry["gravidade"]:
                return i
        return len(self.queue)

fila_emergencia = PriorityQueueSortedArray()
fila_emergencia.push("José", 2)
fila_emergencia.push("Márcia", 3)
fila_emergencia.push("André", 2)
fila_emergencia.push("Bruna", 5)

while fila_emergencia.queue:
    paciente = fila_emergencia.pop()
    print(f"Atendendo paciente: {paciente}")
```

Saida:

```
AzureAD+IanSantos@BRALFSUSABIT03 MINGW64 ~/Documents/faculdade/python
$ C:/Users/IanSantos/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/IanSantos/Documents/faculdade/python/dados-e-algoritimos/tp1/q7.py
Atendendo paciente: José
Atendendo paciente: André
Atendendo paciente: Márcia
Atendendo paciente: Bruna
```

8. **Analise e compare a complexidade das operações de inserção e remoção nas implementações de fila de prioridade desenvolvidas nas questões 6 e 7. Com base nessa comparação, qual você adotaria em uma aplicação real? Justifique sua resposta.**

Fila de Prioridade com Heap (Utilizando a biblioteca heapq):

Inserção (push): A complexidade de inserção em um heap é O(log n), onde n é o número de elementos na fila.

Remoção (pop): A complexidade de remoção em um heap é O(log n), onde n é o número de elementos na fila.

Fila de Prioridade com Array Ordenado:

Inserção (push): A complexidade de inserção em um array ordenado é O(n), onde n é o número de elementos na fila, devido à necessidade de deslocamento de elementos para manter a ordem.

Remoção (pop): A complexidade de remoção em um array ordenado é O(1), pois a remoção do primeiro elemento é uma operação constante.

Conclusao:

Com base nessas análises, podemos ver que a implementação com heap apresenta melhor desempenho tanto na inserção quanto na remoção em comparação com a implementação com array ordenado. O heap mantém a propriedade de heap, o que permite inserções e remoções eficientes em O(log n), enquanto o array ordenado requer deslocamentos de elementos, o que aumenta a complexidade da inserção para O(n).

Portanto, em uma aplicação real onde a eficiência de inserção e remoção é crucial, a implementação com heap seria mais adequada devido à sua melhor complexidade de tempo em relação à inserção e remoção.