Castillo, Anjelica M.

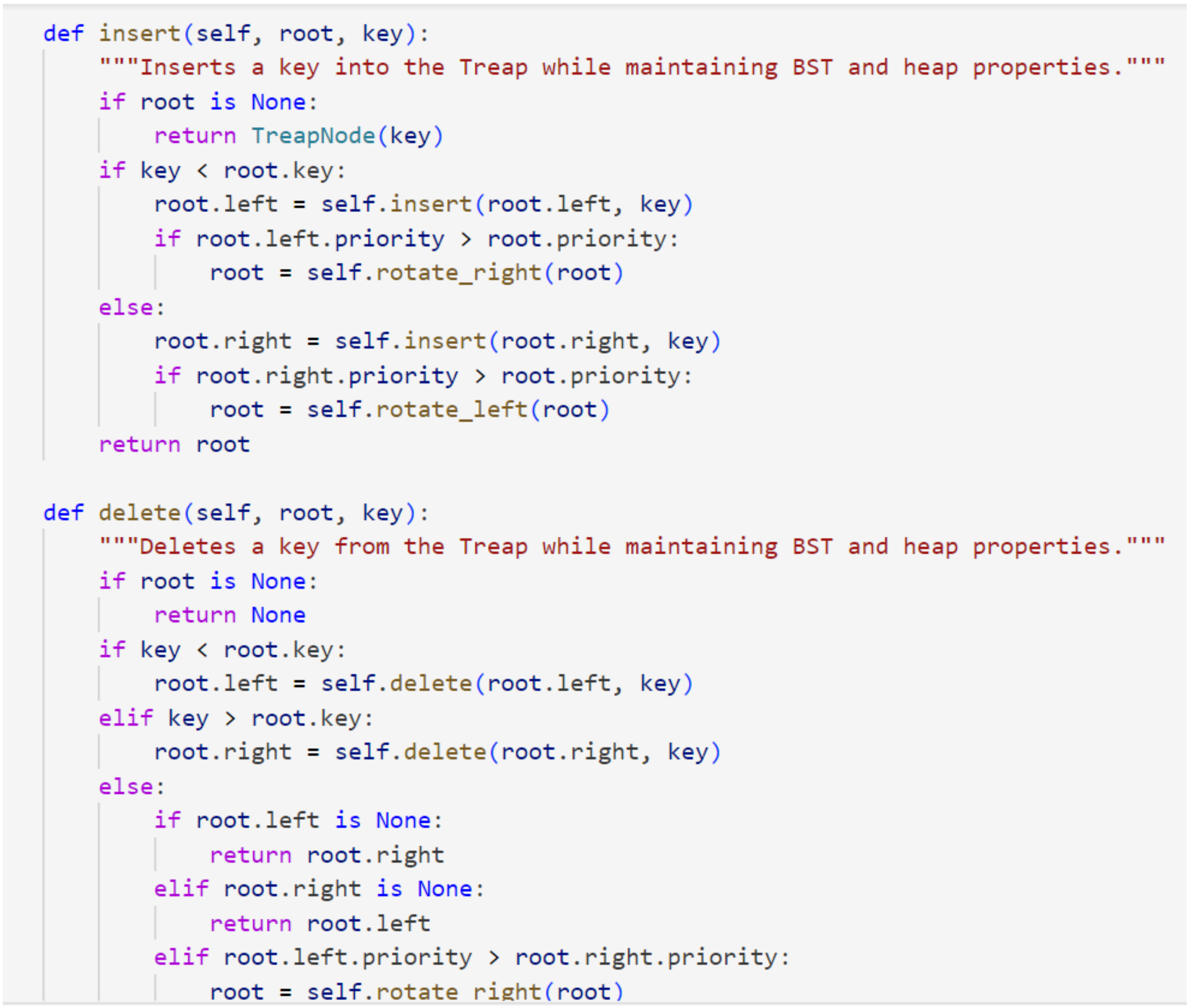
MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms

1. Write an implementation of Treap (using either static or user-input values)

```python
import random
import matplotlib.pyplot as plt
import networkx as nx

class TreapNode:
    """Represents a node in a Treap."""
    def __init__(self, key, priority=None):
        self.key = key
        self.priority = priority if priority is not None else random.randint(1, 100)
        self.left = None
        self.right = None

class Treap:
    """Implements a Treap (Binary Search Tree + Max-Heap)."""

    def rotate_right(self, y):
        """Performs a right rotation."""
        x = y.left
        y.left = x.right
        x.right = y
        return x

    def rotate_left(self, x):
        """Performs a left rotation."""
        y = x.right
        x.right = y.left
        y.left = x
        return y
```

Castillo, Anjelica M.

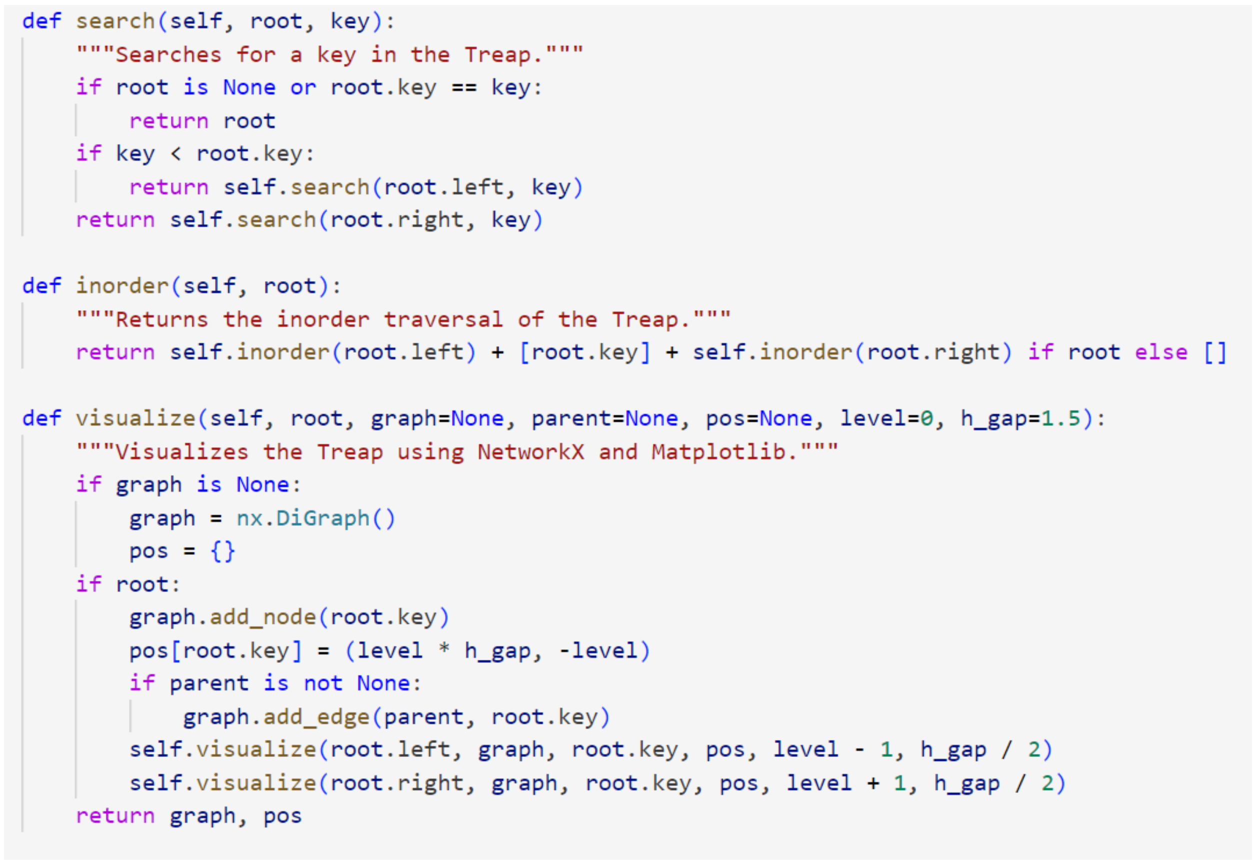
MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms



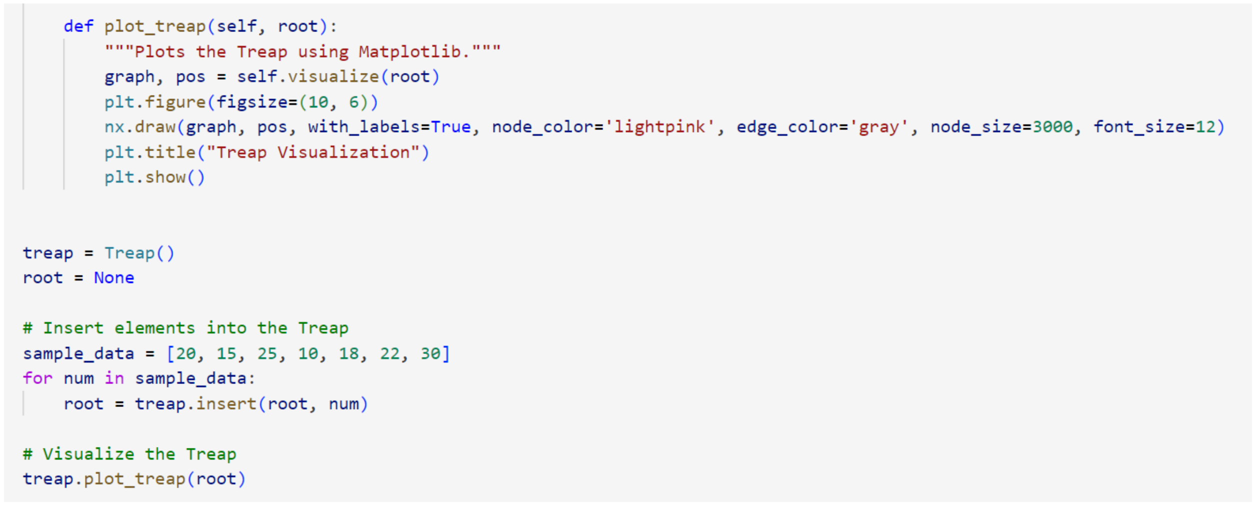
```python
def insert(self, root, key):
    """Inserts a key into the Treap while maintaining BST and heap properties."""
    if root is None:
        return TreapNode(key)
    if key < root.key:
        root.left = self.insert(root.left, key)
        if root.left.priority > root.priority:
            root = self.rotate_right(root)
    else:
        root.right = self.insert(root.right, key)
        if root.right.priority > root.priority:
            root = self.rotate_left(root)
    return root

def delete(self, root, key):
    """Deletes a key from the Treap while maintaining BST and heap properties."""
    if root is None:
        return None
    if key < root.key:
        root.left = self.delete(root.left, key)
    elif key > root.key:
        root.right = self.delete(root.right, key)
    else:
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left
        elif root.left.priority > root.right.priority:
            root = self.rotate_right(root)
```

Castillo, Anjelica M.

MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms



```python
def search(self, root, key):
    """Searches for a key in the Treap."""
    if root is None or root.key == key:
        return root
    if key < root.key:
        return self.search(root.left, key)
    return self.search(root.right, key)

def inorder(self, root):
    """Returns the inorder traversal of the Treap."""
    return self.inorder(root.left) + [root.key] + self.inorder(root.right) if root else []

def visualize(self, root, graph=None, parent=None, pos=None, level=0, h_gap=1.5):
    """Visualizes the Treap using NetworkX and Matplotlib."""
    if graph is None:
        graph = nx.DiGraph()
        pos = {}
    if root:
        graph.add_node(root.key)
        pos[root.key] = (level * h_gap, -level)
        if parent is not None:
            graph.add_edge(parent, root.key)
        self.visualize(root.left, graph, root.key, pos, level - 1, h_gap / 2)
        self.visualize(root.right, graph, root.key, pos, level + 1, h_gap / 2)
    return graph, pos
```

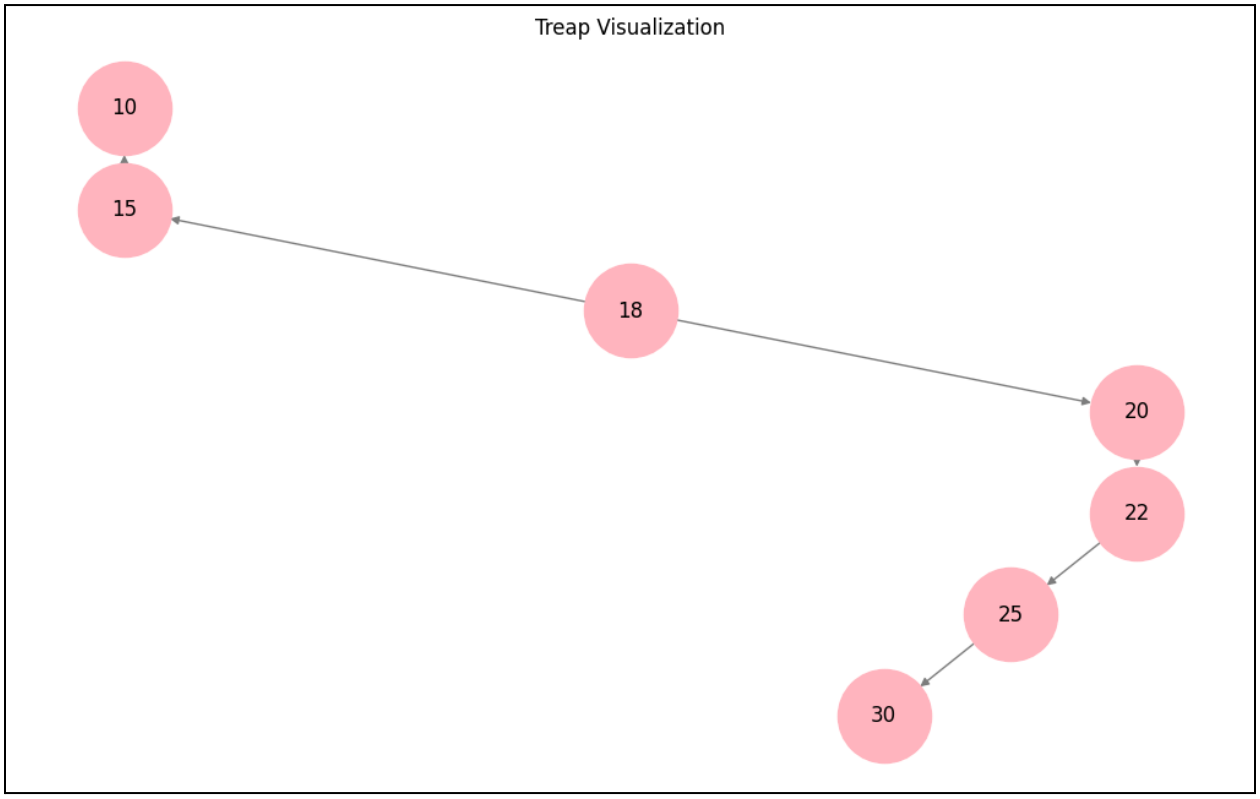


```python
    def plot_treap(self, root):
        """Plots the Treap using Matplotlib."""
        graph, pos = self.visualize(root)
        plt.figure(figsize=(10, 6))
        nx.draw(graph, pos, with_labels=True, node_color='lightpink', edge_color='gray', node_size=3000, font_size=12)
        plt.title("Treap Visualization")
        plt.show()


treap = Treap()
root = None

# Insert elements into the Treap
sample_data = [20, 15, 25, 10, 18, 22, 30]
for num in sample_data:
    root = treap.insert(root, num)

# Visualize the Treap
treap.plot_treap(root)
```

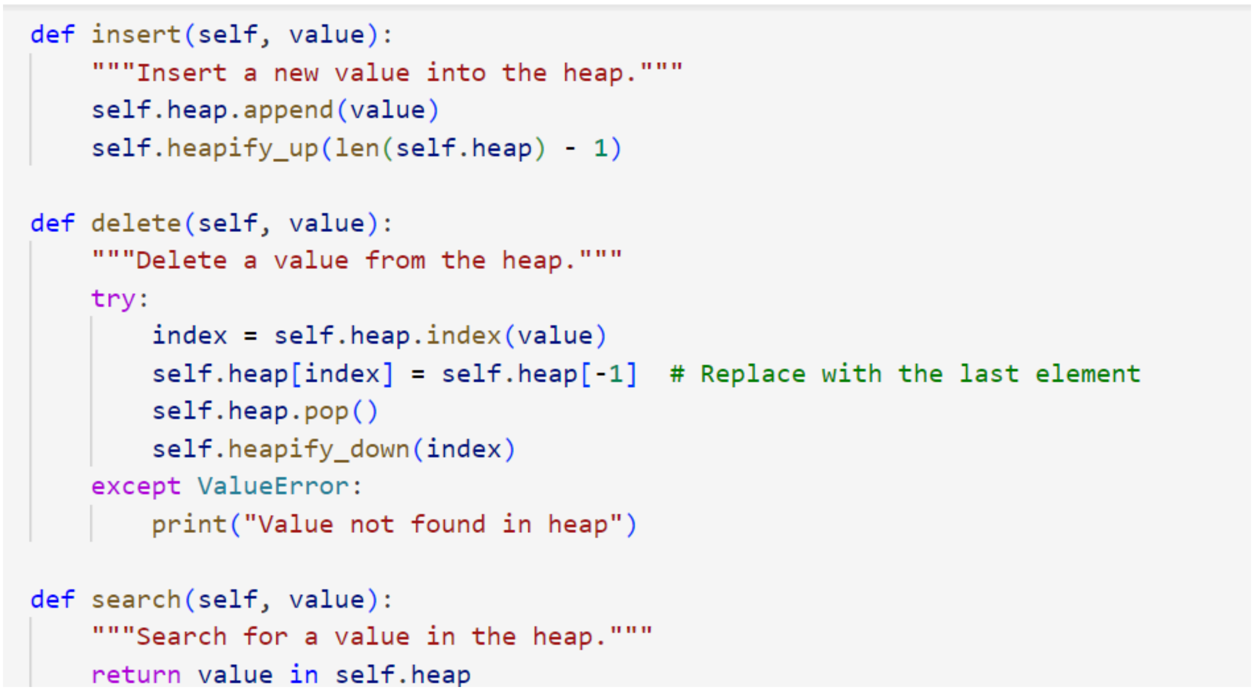
Output:
This is an implementation of simple max-heap is a binary tree where each parent node is greater than or equal to its children, ensuring the largest element is always at the root.

Castillo, Anjelica M.

MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms



2. Create a function for insert, remove and search.

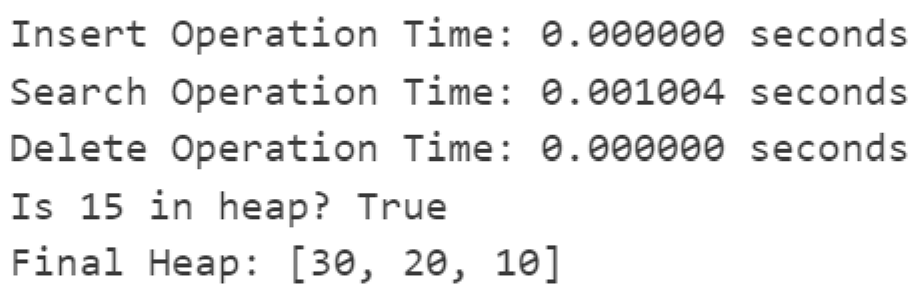


```
def insert(self, value):
    """Insert a new value into the heap."""
    self.heap.append(value)
    self.heapify_up(len(self.heap) - 1)

def delete(self, value):
    """Delete a value from the heap."""
    try:
        index = self.heap.index(value)
        self.heap[index] = self.heap[-1]  # Replace with the last element
        self.heap.pop()
        self.heapify_down(index)
    except ValueError:
        print("Value not found in heap")

def search(self, value):
    """Search for a value in the heap."""
    return value in self.heap
```

Castillo, Anjelica M.
MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms

Time complexity comparison:

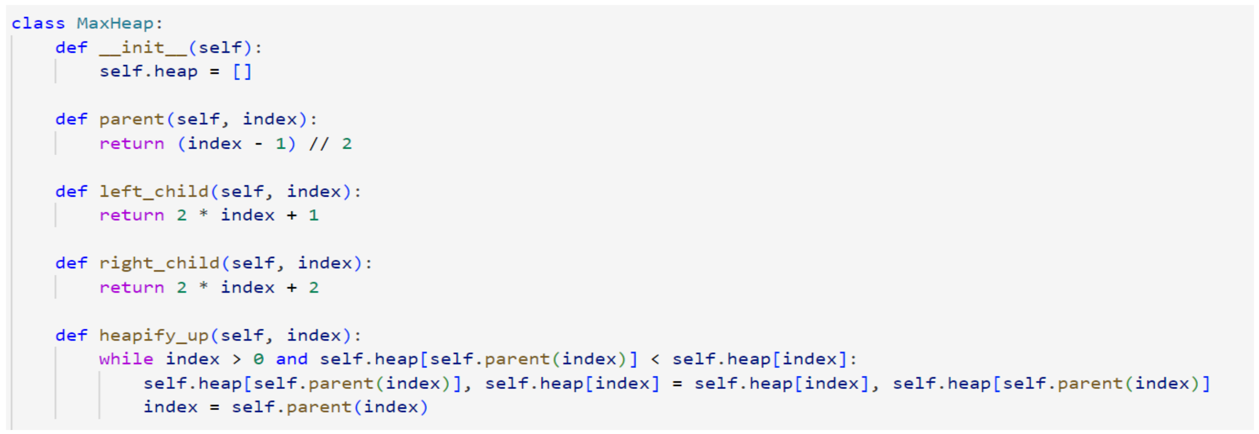
- **Insert**: O(log n) - The time it takes is proportional to the height of the heap.
- **Search**: O(n) - The time it takes is proportional to the number of elements in the heap.
- **Delete**: O(log n) - The time it takes is proportional to the height of the heap, as we need to restore the heap property after deletion.



```
Insert Operation Time: 0.000000 seconds
Search Operation Time: 0.001004 seconds
Delete Operation Time: 0.000000 seconds
Is 15 in heap? True
Final Heap: [30, 20, 10]
```

3. Simulate the results (you may show the AVL rotations) on how the values were inserted and removed.

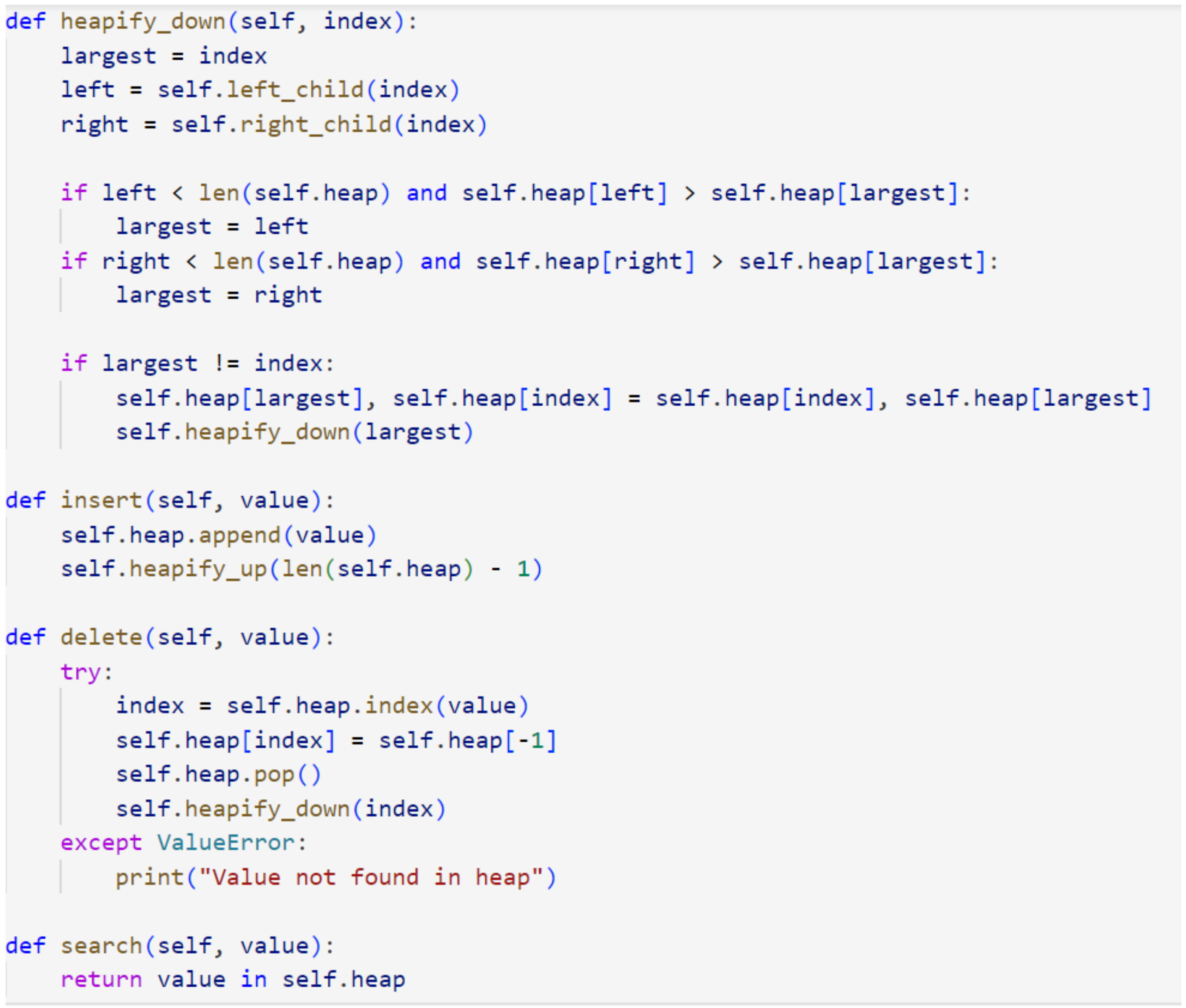
In a Max-Heap, we insert values [10, 20, 15, 30] one by one while ensuring the heap property is maintained, meaning each parent node is greater than or equal to its children. The insertions occur as follows: 10 becomes the root, 20 is inserted and swaps with 10 since it's larger, 15 is added without needing swaps, and 30 is added and undergoes multiple swaps to maintain the heap property. Finally, when deleting 15, it's replaced with the last element (10), and a heapify-down operation restores the heap property by swapping 10 with 20.



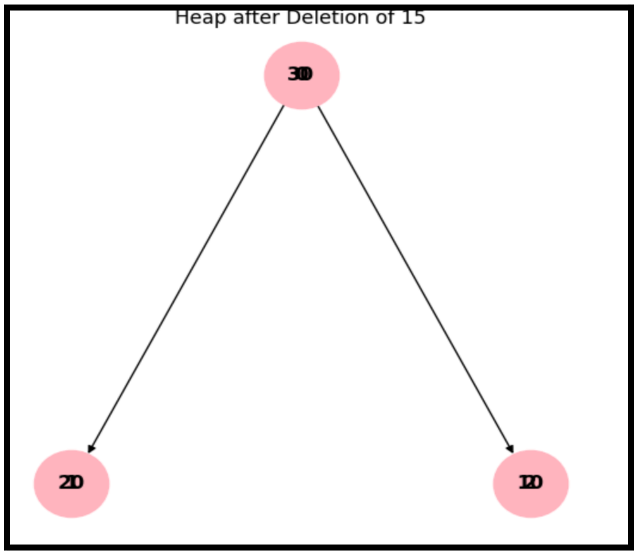
```python
class MaxHeap:
    def __init__(self):
        self.heap = []

    def parent(self, index):
        return (index - 1) // 2

    def left_child(self, index):
        return 2 * index + 1

    def right_child(self, index):
        return 2 * index + 2

    def heapify_up(self, index):
        while index > 0 and self.heap[self.parent(index)] < self.heap[index]:
            self.heap[self.parent(index)], self.heap[index] = self.heap[index], self.heap[self.parent(index)]
            index = self.parent(index)
```

Castillo, Anjelica M.

MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms



```python
def heapify_down(self, index):
    largest = index
    left = self.left_child(index)
    right = self.right_child(index)

    if left < len(self.heap) and self.heap[left] > self.heap[largest]:
        largest = left
    if right < len(self.heap) and self.heap[right] > self.heap[largest]:
        largest = right

    if largest != index:
        self.heap[largest], self.heap[index] = self.heap[index], self.heap[largest]
        self.heapify_down(largest)

def insert(self, value):
    self.heap.append(value)
    self.heapify_up(len(self.heap) - 1)

def delete(self, value):
    try:
        index = self.heap.index(value)
        self.heap[index] = self.heap[-1]
        self.heap.pop()
        self.heapify_down(index)
    except ValueError:
        print("Value not found in heap")

def search(self, value):
    return value in self.heap
```



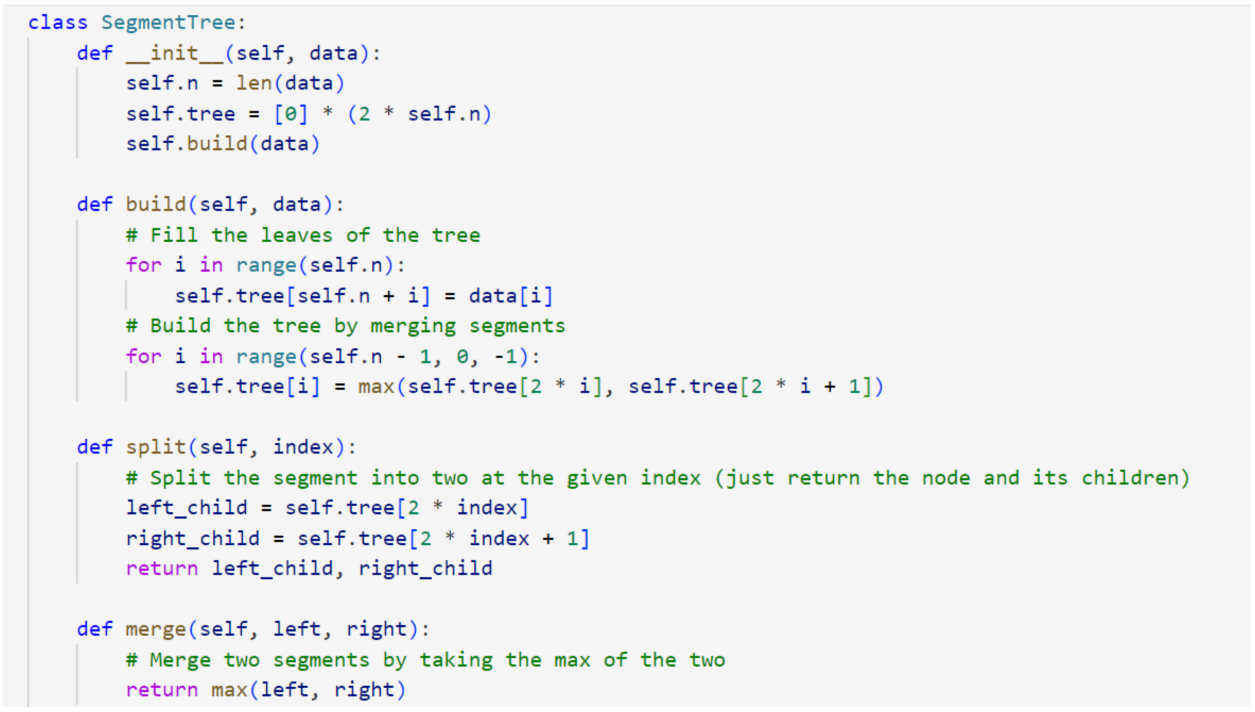
Castillo, Anjelica M.

MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms

1. Compare it using a Tree and a Heap.

Tree	Heap
A tree is a hierarchical data structure consisting of nodes connected by edges, where there is no specific order between parent and child nodes. Example: An organizational chart or file system, where nodes represent employees or files, and their positions are not based on value but structure.	A heap is a complete binary tree that satisfies specific ordering rules: in a max-heap, each parent is greater than or equal to its children. Example: In a max-heap, after inserting 10, 20, and 30, the root will be 30, ensuring the largest element is always at the top.

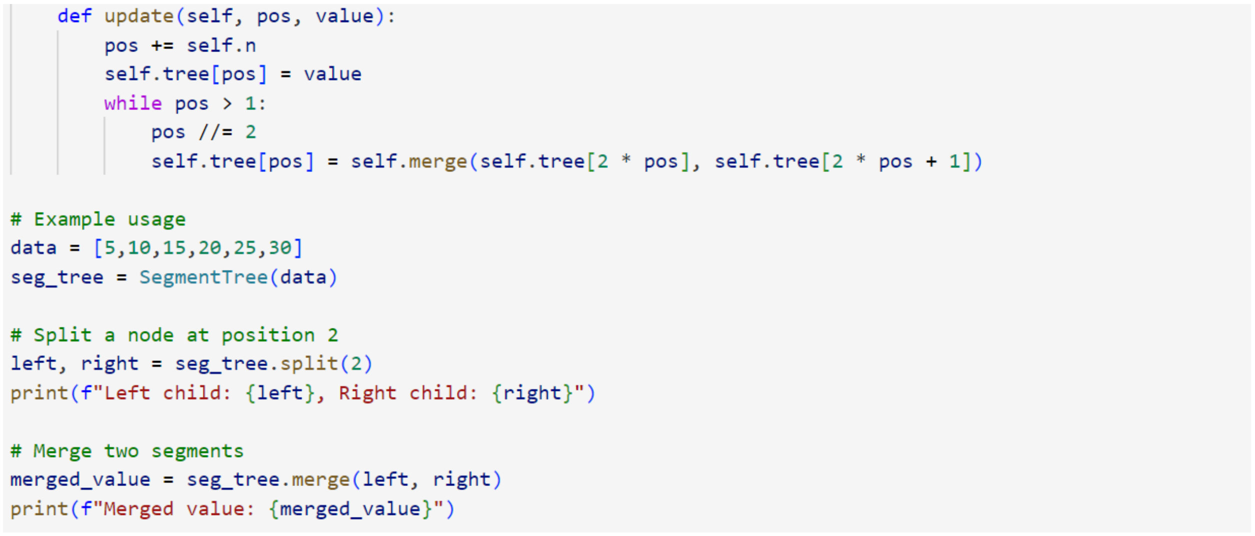
2. Create another function for split and merge. (segmented trees)



```python
class SegmentTree:
    def __init__(self, data):
        self.n = len(data)
        self.tree = [0] * (2 * self.n)
        self.build(data)

    def build(self, data):
        # Fill the leaves of the tree
        for i in range(self.n):
            self.tree[self.n + i] = data[i]
        # Build the tree by merging segments
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = max(self.tree[2 * i], self.tree[2 * i + 1])

    def split(self, index):
        # Split the segment into two at the given index (just return the node and its children)
        left_child = self.tree[2 * index]
        right_child = self.tree[2 * index + 1]
        return left_child, right_child

    def merge(self, left, right):
        # Merge two segments by taking the max of the two
        return max(left, right)
```

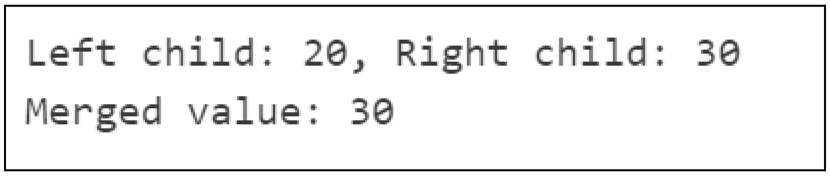
Castillo, Anjelica M.

MCSCC 123-MSCS12S1 - Advanced Data Structures and Algorithms



```python
    def update(self, pos, value):
        pos += self.n
        self.tree[pos] = value
        while pos > 1:
            pos //= 2
            self.tree[pos] = self.merge(self.tree[2 * pos], self.tree[2 * pos + 1])

# Example usage
data = [5,10,15,20,25,30]
seg_tree = SegmentTree(data)

# Split a node at position 2
left, right = seg_tree.split(2)
print(f"Left child: {left}, Right child: {right}")

# Merge two segments
merged_value = seg_tree.merge(left, right)
print(f"Merged value: {merged_value}")
```

The left child is 20 and the right child is 30, and when merged, the larger value, 30, is chosen. This ensures that the heap property is maintained, where the parent node is always greater than or equal to its children. Result:



```
Left child: 20, Right child: 30
Merged value: 30
```