Camila Torres, Jeimy Yaya

Control de lectura – Montones y colas

1. Ilustre el paso a paso de *heapsort* sobre el arreglo:
$$A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$$

A = [5, 13, 2, 25, 7, 17, 20, 8, 4]

Max-heap [25, 20, 17, 8, 7, 2, 13, 5, 4]

[4, 20, 17, 8, 7, 2, 13, 5, 25]

Max-heap [20, 8, 17, 5, 7, 2, 13, 4]

[4, 8, 17, 5, 7, 2, 13, 20, 25]

Max-heap [17, 8, 13, 5, 7, 2, 4]

[4, 8, 13, 5, 7, 2, 17, 20, 25]

Max-heap [13, 8, 4, 5, 7, 2]

[2, 8, 4, 5, 7, 13, 17, 20, 25]

Max-heap [8, 7, 4, 5, 2]

[2, 7, 4, 5, 8, 13, 17, 20, 25]

Max-heap [7, 5, 4, 2]

[2, 4, 5, 7, 8, 13, 17, 20, 25]

Max-heap [5, 4, 2]

[2, 4, 5, 7, 8, 13, 17, 20, 25]

Max-heap [4, 2]

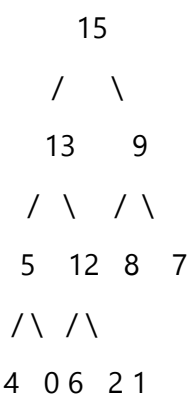
[2, 4, 5, 7, 8, 13, 17, 20, 25]

Max-heap [2]

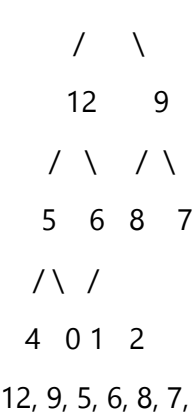
[2, 4, 5, 7, 8, 13, 17, 20, 25]

2. Ilustre el paso a paso de *heap_extract_max* sobre el heap:
$$A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$$

A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]



Se extrae el elemento máximo que es 15



[13, 12, 9, 5, 6, 8, 7, 4, 0, 1, 2]

Max-heapify

[13, 12, 9, 5, 6, 8, 7, 0, 1]

Camila Torres, Jeimy Yaya

3. Ilustre el paso a paso de *max_heap_insert(10)* sobre el heap:

$$A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$$

[15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]

[15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1, 10]

[15, 13, 9, 5, 12, 10, 7, 4, 0, 6, 2, 1, 8]

[15, 13, 10, 5, 12, 9, 7, 4, 0, 6, 2, 8]

4. Implemente el código para las siguientes operaciones sobre un min-heap:
   - heap_mínimum
   - heap_extract_min
   - heap_decrease_key
   - min_heap_insert

```python
class MinHeap:
    def __init__(self, array=[]):
        self.heap = array
        self.build_heap()

    # Función para obtener el elemento mínimo del heap
    def heap_minimum(self):
        if len(self.heap) > 0:
            return self.heap[0]
        else:
            return None

    # Función para extraer el elemento mínimo del heap
    def heap_extract_min(self):
        if len(self.heap) > 0:
            min_val = self.heap[0]
            last_val = self.heap.pop()
            if len(self.heap) > 0:
                self.heap[0] = last_val
                self.min_heapify(0)
            return min_val
        else:
            return None

    # Función para disminuir el valor de una clave en el heap
    def heap_decrease_key(self, index, value):
        if index < len(self.heap):
            self.heap[index] = value
            parent = (index - 1) // 2
            if index > 0 and self.heap[index] < self.heap[parent]:
                self._bubble_up(index)
            else:
                self.min_heapify(index)
        else:
            return None
```

```
    # Función para insertar un nuevo elemento en el heap
    def min_heap_insert(self, value):
        self.heap.append(value)
        self._bubble_up(len(self.heap) - 1)

    # Función auxiliar para transformar un array en un min-heap
    def build_heap(self):
        for i in range(len(self.heap) // 2, -1, -1):
            self.min_heapify(i)

    # Función auxiliar para mantener la propiedad heap hacia arriba
    def _bubble_up(self, index):
        parent = (index - 1) // 2

        while index > 0 and self.heap[index] < self.heap[parent]:
            self.heap[index], self.heap[parent] = self.heap[parent],
self.heap[index]
            index = parent
            parent = (index - 1) // 2

    # Función auxiliar para mantener la propiedad heap hacia abajo
    def min_heapify(self, index):
        left_child = index * 2 + 1
        right_child = index * 2 + 2
        smallest = index

        if left_child < len(self.heap) and self.heap[left_child] <
self.heap[smallest]:
            smallest = left_child

        if right_child < len(self.heap) and self.heap[right_child] <
self.heap[smallest]:
            smallest = right_child

        if smallest != index:
            self.heap[index], self.heap[smallest] = self.heap[smallest],
self.heap[index]
            self.min_heapify(smallest)
```

5. Desarrolle un algoritmo para determinar si un árbol binario es un max-heap:
    - **Entrada:** Arbol binario

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class BinaryTree:
    def __init__(self, root=None):
        self.root = root

def is_max_heap(node):
    # Si el nodo es una hoja, entonces cumple la propiedad de max-heap
    if node.left is None and node.right is None:
```

```
        return True

    # Verificar si el valor del nodo es mayor o igual que el valor de sus
hijos
    if (node.left is not None and node.value < node.left.value) or \
       (node.right is not None and node.value < node.right.value):
        return False

    # Verificar recursivamente si los hijos también cumplen la propiedad
de max-heap
    return is_max_heap(node.left) and is_max_heap(node.right)

def main():
    leaf1 = TreeNode(7)
    leaf2 = TreeNode(6)
    leaf3 = TreeNode(5)
    leaf4 = TreeNode(4)

    node1 = TreeNode(8, leaf1, leaf2)
    node2 = TreeNode(9, leaf3, leaf4)

    root = TreeNode(10, node1, node2)

    tree = BinaryTree(root)
    print(is_max_heap(tree.root))
main()
```