Name:- Yash Sanjay Kale
Reg No:- 2020BIT047
Practical No:- 7

Write a code with complete simulation of the following

1. AVL Tree

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        self.root = self._insert(self.root, value)

    def _insert(self, node, value):
        if not node:
            return Node(value)
        elif value < node.value:
            node.left = self._insert(node.left, value)
        else:
            node.right = self._insert(node.right, value)

        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))

        balance_factor = self._get_balance_factor(node)

        if balance_factor > 1 and value < node.left.value:
            return self._right_rotate(node)

        if balance_factor > 1 and value > node.left.value:
            node.left = self._left_rotate(node.left)
            return self._right_rotate(node)

        if balance_factor < -1 and value > node.right.value:
            return self._left_rotate(node)

        if balance_factor < -1 and value < node.right.value:
            node.right = self._right_rotate(node.right)
            return self._left_rotate(node)

        return node

    def delete(self, value):
        self.root = self._delete(self.root, value)

    def _delete(self, node, value):
        if not node:
            return node
        elif value < node.value:
```

```
            node.left = self._delete(node.left, value)
        elif value > node.value:
            node.right = self._delete(node.right, value)
        else:
            if node.left is None:
                temp = node.right
                node = None
                return temp
            elif node.right is None:
                temp = node.left
                node = None
                return temp

            temp = self._get_min_value_node(node.right)
            node.value = temp.value
            node.right = self._delete(node.right, temp.value)

        if node is None:
            return node

        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))

        balance_factor = self._get_balance_factor(node)

        if balance_factor > 1 and self._get_balance_factor(node.left) >= 0:
            return self._right_rotate(node)

        if balance_factor > 1 and self._get_balance_factor(node.left) < 0:
            node.left = self._left_rotate(node.left)
            return self._right_rotate(node)

        if balance_factor < -1 and self._get_balance_factor(node.right) <= 0:
            return self._left_rotate(node)

        if balance_factor < -1 and self._get_balance_factor(node.right) > 0:
            node.right = self._right_rotate(node.right)
            return self._left_rotate(node)

        return node

    def _get_height(self, node):
        if not node:
            return 0

        return node.height

    def _get_balance_factor(self, node):
        if not node:
            return 0

        return self._get_height(node.left) - self._get_height(node.right)

    def _left_rotate(self, node):
        new_root = node.right
        node.right = new_root.left
```

```
        new_root.left = node

        node.height = 1 + max(self._get_height(node.left), self._get_height(node.right))
        new_root.height = 1 + max(self)
}

2. Binary Heap
class BinaryHeap:
    def __init__(self):
        self.heap = []

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

    def delete_min(self):
        if len(self.heap) == 0:
            return None

        min_val = self.heap[0]
        last_val = self.heap.pop()

        if len(self.heap) > 0:
            self.heap[0] = last_val
            self._heapify_down(0)

        return min_val

    def _heapify_up(self, index):
        parent_index = (index - 1) // 2

        if index > 0 and self.heap[index] < self.heap[parent_index]:
            self.heap[index], self.heap[parent_index] = self.heap[parent_index], self.heap[index]
            self._heapify_up(parent_index)

    def _heapify_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        smallest_child_index = index

        if left_child_index < len(self.heap) and self.heap[left_child_index] < self.heap[smallest_child_index]:
            smallest_child_index = left_child_index

        if right_child_index < len(self.heap) and self.heap[right_child_index] < self.heap[smallest_child_index
]:
            smallest_child_index = right_child_index

        if smallest_child_index != index:
            self.heap[index], self.heap[smallest_child_index] = self.heap[smallest_child_index], self.heap[inde
x]
            self._heapify_down(smallest_child_index)


3. Max Heap
class MaxHeap:
```

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

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

    def delete_max(self):
        if len(self.heap) == 0:
            return None

        max_val = self.heap[0]
        last_val = self.heap.pop()

        if len(self.heap) > 0:
            self.heap[0] = last_val
            self._heapify_down(0)

        return max_val

    def _heapify_up(self, index):
        parent_index = (index - 1) // 2

        if index > 0 and self.heap[index] > self.heap[parent_index]:
            self.heap[index], self.heap[parent_index] = self.heap[parent_index], self.heap[index]
            self._heapify_up(parent_index)

    def _heapify_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        largest_child_index = index

        if left_child_index < len(self.heap) and self.heap[left_child_index] > self.heap[largest_child_index]:
            largest_child_index = left_child_index

        if right_child_index < len(self.heap) and self.heap[right_child_index] > self.heap[largest_child_index]:

            largest_child_index = right_child_index

        if largest_child_index != index:
            self.heap[index], self.heap[largest_child_index] = self.heap[largest_child_index], self.heap[index]
            self._heapify_down(largest_child_index)
```

4. Min Heap

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

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

    def delete_min(self):
        if len(self.heap) == 0:
```

```
            return None

        min_val = self.heap[0]
        last_val = self.heap.pop()

        if len(self.heap) > 0:
            self.heap[0] = last_val
            self._heapify_down(0)

        return min_val

    def _heapify_up(self, index):
        parent_index = (index - 1) // 2

        if index > 0 and self.heap[index] < self.heap[parent_index]:
            self.heap[index], self.heap[parent_index] = self.heap[parent_index], self.heap[index]
            self._heapify_up(parent_index)

    def _heapify_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        smallest_child_index = index

        if left_child_index < len(self.heap) and self.heap[left_child_index] < self.heap[smallest_child_index]:
            smallest_child_index = left_child_index

        if right_child_index < len(self.heap) and self.heap[right_child_index] < self.heap[smallest_child_index
]:
            smallest_child_index = right_child_index

        if smallest_child_index != index:
            self.heap[index], self.heap[smallest_child_index] = self.heap[smallest_child_index], self.heap[inde
x]
            self._heapify_down(smallest_child_index)


5.Heapfy
def heapify(heap, n, i):
    """
    Heapifies the given heap with n elements, assuming that all elements except the root satisfy the heap p
roperty,
    starting at the given index i.
    """
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and heap[left] > heap[largest]:
        largest = left

    if right < n and heap[right] > heap[largest]:
        largest = right

    if largest != i:
        heap[i], heap[largest] = heap[largest], heap[i]
```

heapify(heap, n, largest)