QUESTION-1

```
class Node:
  def __init__(self, val):
    self.val = val  # Set the value of the node
    self.left = None  # Initialize the left child to None
    self.right = None  # Initialize the right child to None


class BST:
  def __init__(self):
    self.root = None  # Initialize the root of the tree to None

  def insert(self, val):
    if self.root is None:
      self.root = Node(val)  # If tree is empty, create root node with given value
    else:
      self._insert(val, self.root)  # Otherwise, insert the value recursively starting from the root

  def _insert(self, val, current_node):
    if val < current_node.val:  # If the value to be inserted is less than current node's value
      if current_node.left is None:  # If the current node doesn't have a left child
        current_node.left = Node(val)  # Create a new node with given value and set it as left child of current
node
      else:
        self._insert(val, current_node.left)  # Otherwise, recursively insert the value in the left subtree
    else:  # If the value to be inserted is greater than or equal to current node's value
      if current_node.right is None:  # If the current node doesn't have a right child
        current_node.right = Node(val)  # Create a new node with given value and set it as right child of
current node
      else:
        self._insert(val, current_node.right)  # Otherwise, recursively insert the value in the right subtree

  def delete(self, val):
    if self.root is not None:  # If the tree is not empty
      self._delete(val, self.root)  # Call the recursive delete function starting from the root

  def _delete(self, val, current_node):
    if val == current_node.val:  # If the value to be deleted matches the current node's value
      if current_node.left is None and current_node.right is None:  # If the current node is a leaf node
        if current_node == self.root:  # If the current node is the root of the tree
          self.root = None  # Set the root to None
        else:
          parent_node = self._find_parent_node(val, self.root)  # Find the parent node of the current node
          if current_node == parent_node.left:  # If the current node is the left child of the parent node
            parent_node.left = None  # Set the left child of the parent node to None
          else:  # If the current node is the right child of the parent node
            parent_node.right = None  # Set the right child of the parent node to None
      elif current_node.left is None:  # If the current node has only a right child
        if current_node == self.root:  # If the current node is the root of the tree
```

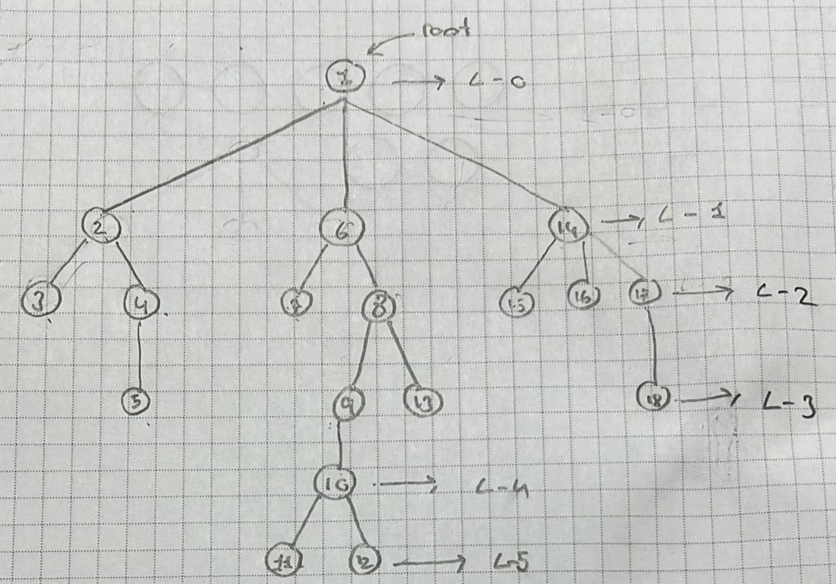
```
            self.root = current_node.right  # Set the right child of the current node as the new root
          else:
            parent_node = self._find_parent_node(val, self.root)  # Find the parent node of the current node
            if current_node == parent_node.left:  # If the current node is the left child of the parent node
              parent_node.left = current_node.right  # Set the right child of the current node as the new left
child of the parent node
            else:  # If the current node is the right child of the parent node
              parent_node.right = current_node.right  # Set the right child of the current node as the new right
child of the parent node
    else:  # If the current node has both left and right children
      successor_node = self._find_successor_node(current_node.right)  # Find the inorder successor node of
the current node
      current_node.val = successor_node.val  # Copy the value of the successor node to the current node
      self._delete(successor_node.val, current_node.right)  # Recursively delete the successor node from the
right subtree
    elif val < current_node.val:  # If the value to be deleted is less than the current node's value
      if current_node.left is not None:  # If the current node has a left child
      self._delete(val, current_node.left)  # Recursively delete the value from the left subtree
    elif val > current_node.val:  # If the value to be deleted is greater than the current node's value
      if current_node.right is not None:  # If the current node has a right child
      self._delete(val, current_node.right)  # Recursively delete the value from the right subtree

def _find_parent_node(self, val, current_node):
  if (current_node.left is not None and current_node.left.val == val) or \
      (current_node.right is not None and current_node.right.val == val):  # If the value is a child of the
current node
    return current_node  # Return the current node
  elif val < current_node.val and current_node.left is not None:  # If the value is less than the current node's
value and the current node has a left child
    return self._find_parent_node(val, current_node.left)  # Recursively find the parent node in the left
subtree
  elif val > current_node.val and current_node.right is not None:  # If the value is greater than the current
node's value and the current node has a right child
    return self._find_parent_node(val, current_node.right)  # Recursively find the parent node in the right
subtree

def _find_successor_node(self, current_node):
  if current_node.left is None:  # If the current node doesn't have a left child
    return current_node  # Return the current node
  else:  # If the current node has a left child
    return self._find_successor_node(current_node.left)  # Recursively find the inorder successor node in the
left subtree
```

The time complexity of the insert and delete functions in a binary search tree depends on the height of the tree. If the tree is balanced, i.e., the height is logarithmic in the number of nodes, then the time complexity of both insert and delete functions is O(log n), where n is the number of nodes in the tree. However, in the worst

case, the tree can be unbalanced, i.e., the height can be linear in the number of nodes, which results in a time complexity of O(n) for both insert and delete functions

QUESTION-2

...../...../.....



Breadth - list

= 1-2-6-14-3-4-7-8-15-16-17-5-9-13-18-10-11-12

1→2→6→14→3→4→7→8→15→16→17→5→9→13→18→10→11-

Depth - list

<root> <left> <right> → Preorder

1→2→3→4→5→6→7→8→9→10→11→12→13→14→15→16→17→1

Time complexity = O(n)

QUESTION-3

```
class Node:
    def __init__(self, value=None):
        self.value = value
        self.next = None


class Queue:
    def __init__(self):
        self.head = None  # initialize head pointer to None
        self.tail = None  # initialize tail pointer to None

    def queue(self, value):
        node = Node(value)  # create a new node with the given value
        if self.tail is not None:
            self.tail.next = node  # if queue is not empty, add new node at the end
        else:
            self.head = node  # if queue is empty, new node becomes head
        self.tail = node  # update tail to the new node

    def dequeue(self):
        if self.head is None:  # if queue is empty
            return None
        value = self.head.value  # get the value of the head node
        self.head = self.head.next  # move the head pointer to the next node
        if self.head is None:  # if there are no more nodes left
            self.tail = None  # update tail to None as well
        return value  # return the value of the dequeued node

    def front(self):
        if self.head is None:  # if queue is empty
            return None
        return self.head.value  # return the value of the head node

    def empty(self):
        return self.head is None  # return True if head is None, i.e., queue is empty
```

QUESTION-4

...../...../.....

Size() => Use breadth-first travel to count the number of nodes
time complexity O(n), space complexity, where w is
the maximum width of tree

height() => Use breadth-first travel to keep track of the
current level and increment a counter until the last level
time complexity O(n), space complexity O(n), where
w is maximum width of the tree. Alternatively, use
a stack or queue to do a depth-first traversal
iteratively without recursion with time complexity = O(n)

```
class TreeNode:
    def __init__(self, value, children=[]):
        self.value = value
        self.children = children

    def size(self):
        # Base case: leaf node with no children
        if not self.children:
            return 1

        # Recursive case: node with children
        total_size = 1  # include self in count
        for child in self.children:
            total_size += child.size()  # recursively count each child's subtree size
        return total_size

    def height(self):
        # Base case: leaf node with no children
        if not self.children:
            return 0
```

```
        # Recursive case: node with children
        max_child_height = 0
        for child in self.children:
            max_child_height = max(max_child_height, child.height())  # recursively find the height of each child's subtree
        return max_child_height + 1  # add 1 for the current level
```

QUESTION-5

WİTH NO recursive

```
class Node {
   int data;
   Node left, right;

   public Node(int item) {
      data = item;
      left = right = null;
   }
}

class BinaryTree {
   Node root;

   public Node findMax() {
      if (root == null)
         return null;

      Node current = root;
      while (current.right != null) {
         current = current.right;
      }

      return current;
   }

   public Node findMin() {
      if (root == null)
```

```
            return null;


        Node current = root;
        while (current.left != null) {
            current = current.left;
        }


        return current;
    }
}


public static void main(String[] args) {
        // Create a binary search tree
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(6);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);
        tree.root.right.left = new Node(5);
        tree.root.right.right = new Node(7);


        // Test findMax()
        Node maxNode = tree.findMax();
        if (maxNode != null) {
            System.out.println("Maximum value in the binary search tree: " + maxNode.data);
        } else {
```

```
            System.out.println("Binary search tree is empty.");
        }

        // Test findMin()
        Node minNode = tree.findMin();
        if (minNode != null) {
            System.out.println("Minimum value in the binary search tree: " + minNode.data);
        } else {
            System.out.println("Binary search tree is empty.");
        }
    }
```

WİTH   recursive

```
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinaryTree {
    Node root;

    public Node findMax(Node node) {
```

```
        if (node.right == null)
            return node;

        return findMax(node.right);
    }

    public Node findMin(Node node) {
        if (node.left == null)
            return node;

        return findMin(node.left);
    }
}
public static void main(String[] args) {
        // Create a binary search tree
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(4);
        tree.root.left = new Node(2);
        tree.root.right = new Node(6);
        tree.root.left.left = new Node(1);
        tree.root.left.right = new Node(3);
        tree.root.right.left = new Node(5);
        tree.root.right.right = new Node(7);

        // Test findMax() recursively
        Node maxNode = tree.findMax(tree.root);
        if (maxNode != null) {
```

```java
        System.out.println("Maximum value in the binary search tree (recursive): " +
maxNode.data);
      } else {
        System.out.println("Binary search tree is empty.");
      }

      // Test findMin() recursively
      Node minNode = tree.findMin(tree.root);
      if (minNode != null) {
        System.out.println("Minimum value in the binary search tree (recursive): " +
minNode.data);
      } else {
        System.out.println("Binary search tree is empty.");
      }
    }
```