```java
public class IntNode {
	private int content;
	private IntNode left;
	private IntNode right;

	// Konstruktor
	public IntNode (int content) {
		this.content = content;
		left = null;
		right = null;
	}

	// Getter und Setter
	public int getContent() {
		return content;
	}

	public void setContent(int content) {
		this.content = content;
	}

	public IntNode getLeft() {
		return left;
	}

	public void setLeft(IntNode left) {
		this.left = left;
	}

	public IntNode getRight() {
		return right;
	}

	public void setRight(IntNode right) {
		this.right = right;
	}
}
```

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

public class IntBinTree {
	private IntNode root;

	public IntBinTree () { }

	public IntBinTree (int content) {
		this.root = new IntNode (content);
	}

	public IntBinTree(IntBinTree left, Integer content, IntBinTree right) {
		root = new IntNode(content);
		if (left != null) {
			root.setLeft(left.root);
		}
		if (root != null) {
			root.setRight(right.root);
		}
	}

	private IntBinTree(IntNode root) {
		this.root = root;
	}

	public boolean isEmpty() {
		return root == null;
	}

	public Integer getValue() {
		if (isEmpty ()) {
			return null; // error
		}
		return root.getContent();
	}

	public IntBinTree getLeft() {
		if (isEmpty()) {
			return null; // error
		}
		return new IntBinTree(root.getLeft());
	}

	public IntBinTree getRight() {
		if (this.isEmpty()) {
			return null; // error
		}
		return new IntBinTree(root.getRight());
	}

	public void setLeft(IntBinTree tree) {
```

```
            root.setLeft(tree.root);
    }

    public void setRight(IntBinTree tree) {
            root.setRight(tree.root);
    }


    // Aufgabe 1b
    public Integer[] inorder() {
    List<Integer> result = new ArrayList<>();
    inorder(root, result);
    return result.toArray(new Integer[0]);
}

private void inorder(IntNode node, List<Integer> result) {
    if (node == null) {
        return;
    }
    inorder(node.getLeft(), result);
    result.add(node.getContent());
    inorder(node.getRight(), result);
}

// Aufgabe 1c
public static IntBinTree createTree(Integer[] values) {
    if (values.length == 0) {
        return new IntBinTree()
    }

    LinkedList<IntBinTree> queue = new LinkedList<>();
    IntBinTree root = new IntBinTree(values[0]);
    queue.add(root);

    for (int i = 1; i < values.length; i++) {
        IntBinTree current = queue.removeFirst();
        if (current.getLeft().isEmpty()) {
            current.setLeft(new IntBinTree(values[i]));
            queue.add(current.getLeft());
        } else if (current.getRight().isEmpty()) {
            current.setRight(new IntBinTree(values[i]));
            queue.add(current.getRight());
        }
    }
    return root;
}

// Aufgabe 1d
public int countNodes() {
    return countNodes(root);
}

private int countNodes(IntNode node) {
    if (node == null) {
```

```
            return 0;
        }
        return 1 + countNodes(node.getLeft()) + countNodes(node.getRight());
    }

    public int countInnerNodes() {
        return countInnerNodes(root);
    }

    private int countInnerNodes(IntNode node) {
        if (node == null) {
            return 0;
        }
        if (node.getLeft() == null && node.getRight() == null) {
            return 0;
        }
        return 1 + countInnerNodes(node.getLeft()) +
countInnerNodes(node.getRight());
    }

    public int countLeaves() {
        return countLeaves(root);
    }

    private int countLeaves(IntNode node) {
        if (node == null) {
            return 0;
        }
        if (node.getLeft() == null && node.getRight() == null) {
            return 1;
        }
        return countLeaves(node.getLeft()) + countLeaves(node.getRight());
    }

    public int getHeight() {
        return getHeight(root);
    }

    private int getHeight(IntNode node) {
        if (node == null) {
            return -1;
        }
        return 1 + Math.max(getHeight(node.getLeft()),
getHeight(node.getRight()));
    }

    // Aufgabe 1e
    public boolean isFull() {
        return isFull(root);
    }

    private boolean isFull(IntNode node) {
        if (node == null) {
            return true;
```

```java
        }
        if (node.getLeft() == null || node.getRight() == null) {
            return false;
        }
        return isFull(node.getLeft()) && isFull(node.getRight());
    }

    public boolean isComplete() {
        if (root == null) {
            return true;
        }
        Queue<IntNode> queue = new LinkedList<>();
        queue.offer(root);
        boolean flag = false;
        while (!queue.isEmpty()) {
            IntNode current = queue.poll();
            if (current.getLeft() == null && current.getRight() != null) {
                return false;
            }
            if (flag && (current.getLeft() != null || current.getRight() !=
null)) {
                return false;
            }
            if (current.getLeft() == null || current.getRight() == null) {
                flag = true;
            }
            if (current.getLeft() != null) {
                queue.offer(current.getLeft());
            }
            if (current.getRight() != null) {
                queue.offer(current.getRight());
            }
        }
        return true;
    }

    public boolean isPerfect() {
        return isPerfect(root, getHeight(root));
    }

    private boolean isPerfect(IntNode node, int height) {
        if (node == null) {
            return true;
        }
        if (getHeight(node.getLeft()) != height - 1 ||
getHeight(node.getRight()) != height - 1) {
            return false;
        }
        return isPerfect(node.getLeft(), height - 1) &&
isPerfect(node.getRight(), height - 1);
    }
}
```

```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class IntBinTreeTest {

	@Test
	void testIsFull() {
		Integer[] values1 = { 1, 2, 3 };
		IntBinTree tree1 = IntBinTree.createTree(values1);
		assertTrue(tree1.isFull());

		Integer[] values2 = { 1, 2 };
		IntBinTree tree2 = IntBinTree.createTree(values2);
		assertFalse(tree2.isFull());
	}

	@Test
	void testIsComplete() {
		Integer[] values1 = { 1, 2, 3, 4, 5, 6, 7 };
		IntBinTree tree1 = IntBinTree.createTree(values1);
		assertTrue(tree1.isComplete());

		Integer[] values2 = { 1, 2, 3, 4, 5, null, 7 };
		IntBinTree tree2 = IntBinTree.createTree(values2);
		assertFalse(tree2.isComplete());
	}

	@Test
	void testIsPerfect() {
		Integer[] values1 = { 1, 2, 3, 4, 5, 6, 7 };
		IntBinTree tree1 = IntBinTree.createTree(values1);
		assertTrue(tree1.isPerfect());

		Integer[] values2 = { 1, 2, 3, 4, 5, null, 7 };
		IntBinTree tree2 = IntBinTree.createTree(values2);
		assertFalse(tree2.isPerfect());

		Integer[] values3 = { 1 };
		IntBinTree tree3 = IntBinTree.createTree(values3);
		assertTrue(tree3.isPerfect());
	}

	@Test
	void testCountNodes() {
		Integer[] values = { 1, 2, 3, 4, 5, 6, 7 };
		IntBinTree tree = IntBinTree.createTree(values);
		assertEquals(7, tree.countNodes());
	}

	@Test
	void testCountInnerNodes() {
		Integer[] values = { 1, 2, 3, 4, 5, 6, 7 };
		IntBinTree tree = IntBinTree.createTree(values);
```

```
		assertEquals(3, tree.countInnerNodes());
	}

	@Test
	void testCountLeaves() {
		Integer[] values = { 1, 2, 3, 4, 5, 6, 7 };
		IntBinTree tree = IntBinTree.createTree(values);
		assertEquals(4, tree.countLeaves());
	}

	@Test
	void testGetHeight() {
		Integer[] values = { 1, 2, 3, 4, 5, 6, 7 };
		IntBinTree tree = IntBinTree.createTree(values);
		assertEquals(3, tree.getHeight());
	}

}
```

```
public class TrieNode {

    private char letter;
    private int value;
    private TrieNode[] children;
    private static final int ALPHABET_SIZE = 26; // assuming we're using only
lowercase english alphabet

    public TrieNode() {
        children = new TrieNode[ALPHABET_SIZE];
        value = -1;
    }

    public TrieNode(char letter) {
        this();
        this.letter = letter;
    }

    public int getValue() {
        return value;
        }

    public void setValue(int value) {
        this.value = value;
        }

    /**
     * Adds a new child node with the given key and value to this node.
     * @param letter The character of this node
     * @param value The value associated with this node, or -1 if no value is
associated
     * @return The new child if it could be added or if a node with the given
key is already
     *         there, null if there are already three children
     */
    public TrieNode addChild(char letter, int value) {
        int index = letter - 'a';
        if (children[index] == null) {
            children[index] = new TrieNode(letter);
            children[index].setValue(value);
            return children[index];
        }
        return null;
        }

    /**
     * Searches this node's direct children for a node with the given key.
     * @param letter The character to look for
     * @return Returns the TrieNode with the given key if it was found, or null
otherwise
     */
    public TrieNode find(char letter) {
        int index = letter - 'a';
        if (children[index] != null) {
```

```
            return children[index];
        }
        return null;
        }

}
```

```
public class Trie {

    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    /**
     * Adds a new value with the given key to the trie, creating new TrieNodes
as required.
     * @param key The character sequence associated with the new value
     * @param value The new value
     * @return True if the value could be added to the trie, false otherwise
     */
    public boolean addValue(char[] key, int value) {
        TrieNode current = root;
        for (char letter : key) {
            TrieNode child = current.find(letter);
            if (child == null) {
                child = current.addChild(letter, -1);
                if (child == null) {
                    return false;
                }
            }
            current = child;
        }
        current.setValue(value);
        return true;
        }

    /**
     * Returns the value associated with a given key, or -1 if the key could not
be found.
     * @param key The given key
     * @return The associated value, or -1 if the key is not represented in this
trie
     */
    public int findValue(char[] key) {
        TrieNode current = root;
        for (char letter : key) {
            TrieNode child = current.find(letter);
            if (child == null) {
                return -1;
            }
            current = child;
        }
        return current.getValue();
    }
}
```

```
public class TestTrie {

    private static final String[] KEYS = new String[] {
            "green", "greed", "great", "grey",
            "grass", "blue", "blues", "bla",
            "blank", "blame", "black"
    };

    public static void main(String[] args) {
        Trie trie = new Trie();
        for (int i=0;i<KEYS.length;i++) {
            trie.addValue(KEYS[i].toCharArray(), i + 1);
        }
        for (int i=0;i<KEYS.length;i++) {
            System.out.println(trie.findValue(KEYS[i].toCharArray()));
        }
        // Expected output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
    }

}
```