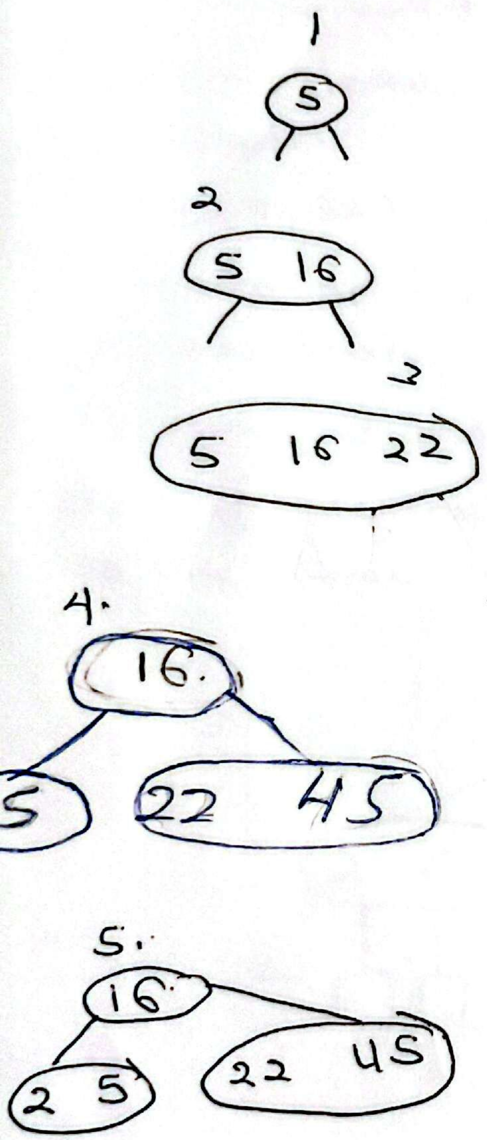
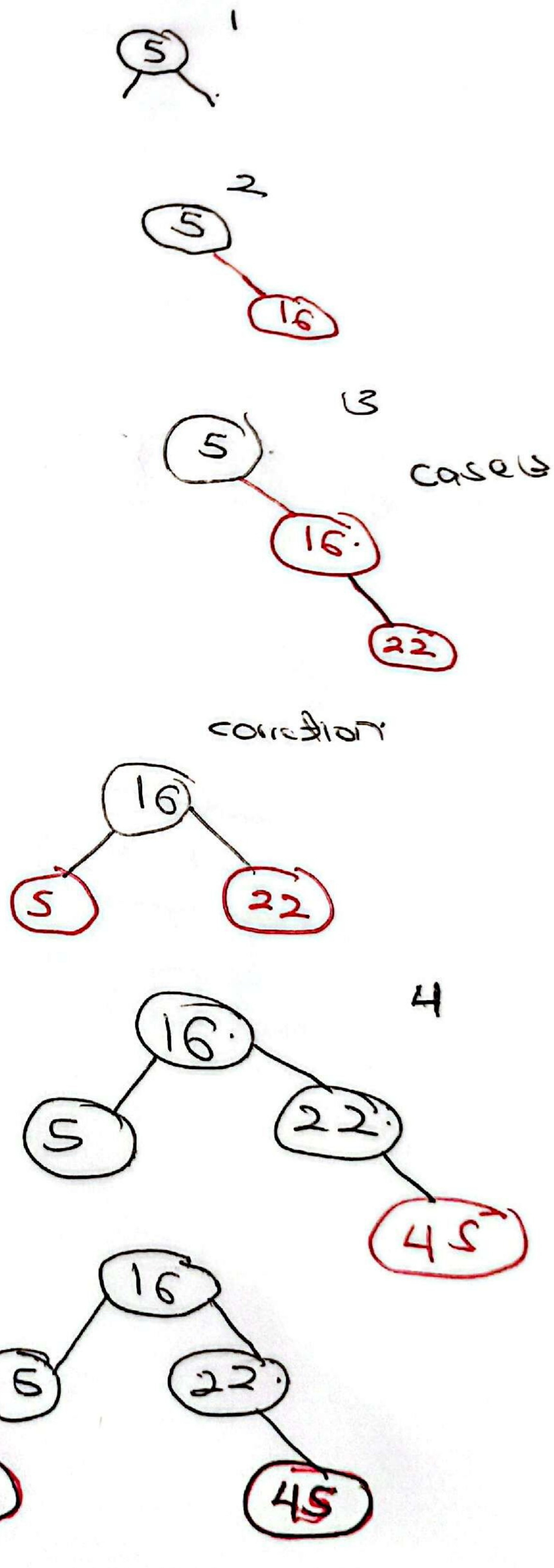
R-3.11. Assignment 9.

5, 16, 22, 45 2, 10, 18, 30, 50, 12, 13, 33

(2,4) tree

red black tree





Assignment 9

R-3.14 (Red-Black Trees: True or False Statements)

a.) a subtree of a red-black tree is itself a red-black tree

True.

Justification:
A red-black tree is a binary search tree (BST) that satisfies the following properties:

1. Every node is either red or black.

2. The root is black.

3. All external (null) nodes are black.

4. If a node is red, then both its children are black (no two consecutive reds).

5. Every path from a node to its descendant external nodes contains the same number of black nodes (black-height property).

If you take any subtree rooted at a node v, it still satisfies properties (1), (4), and (5) because they hold recursively for all nodes. However, property (2) might fail (the subtree root might not be black). But if we consider the subtree as a separate red-black tree without enforcing that the root must be black, it still satisfies red-black properties. Many textbooks relax this property for subtrees.
Conclusion: True, if we allow the root of the subtree to be red.

b. The sibling of an external node is either external or it is red.
True.

Justification:
An external node in red-black trees refers to a null leaf (black). Its parent must have two children (one being the external null).

If the sibling were black and internal, the black-height property would be violated, because the path through the external child would have fewer black nodes than the path through the sibling's subtree.

Therefore, if the sibling is not external, it must be red, ensuring the same black-height.

c. Given a red-black tree T, there is a unique (2,4) tree T′ associated with T.

True.

Justification:
There is a well-known one-to-one correspondence between red-black trees and (2,4) trees (also called 2-3-4 trees):

Black nodes correspond to nodes in the 2-3-4 tree.

A red child of a black node is merged with its parent to form a 3-node or 4-node.

Since the merging process is deterministic, the (2,4) tree is unique.

d.) Given a (2,4) tree T, there is a unique red-black tree T′ associated with T.

False.

Counterexample:
A 4-node in a (2,4) tree corresponds to one black node with two red children in a red-black tree.
But the arrangement of red children (left or right) can vary, producing
different red-black trees for the same 2-4 tree.
For example:

```
  Black(10)
  /     \
Red(5)   Red(15)
```

or

```
  Black(10)
  /     \
Red(15)  Red(5)
```

represent the same 4-node in the (2,4) tree.

C-3.10: FindAllInRange(k1, k2) in O(log n + s)

```
FindAllInRange(k1, k2):
    result ← empty list
    def InOrderRange(node):
        if node = null:
            return
        if node.key > k1:
            InOrderRange(node.left)
        if k1 < node.key < k2:
            result.add(node)
        if node.key < k2:
            InOrderRange(node.right)

    root ← D.root
    InOrderRange(root)
    return iterator(result)
```

Running Time Justification:

Searching to the first element in range takes O(log n) because we traverse down the tree.

Once we find the range, we do an in-order traversal only on s nodes that are in the range, costing O(s).

Total: O(log n + s).

A. isPermutation(A, B) using a Dictionary

```
isPermutation(A, B):
    if len(A) ≠ len(B):
        return False

    dict ← empty dictionary

    // Count occurrences in A
    for x in A:
        if x in dict:
            dict[x] += 1
        else:
            dict[x] = 1

    // Subtract occurrences using B
    for y in B:
        if y not in dict or dict[y] = 0:
            return False
        dict[y] -= 1

    return True
```

Worst-case Time Complexity:

Building the dictionary: $O(n)$

Checking B: $O(n)$

Total: $O(n)$ (assuming dictionary operations are $O(1)$ on average)

B. isBalanced(T)

```
isBalanced(T):
    def checkHeight(node):
        if node = null:
            return 0
        left = checkHeight(node.left)
        if left = -1:
            return -1
        right = checkHeight(node.right)
        if right = -1:
            return -1
        if abs(left - right) > 1:
            return -1
        return 1 + max(left, right)

    return (checkHeight(T.root) ≠ -1)
```

Time Complexity:

Each node is visited once, and height computation is done in constant time per node.

O(n) for n nodes.