Question 1

Suppose we want to use a Binary Search Tree to store only keys (without any additional information), and we want to allow duplicate keys. Modify the TreeInsert algorithm to handle duplicate keys. Provide the updated algorithm and explain the changes.

Changes made: Make the first key comparison between that of `x` and `root` to be unstrict.

Listing 1: Modified TreeInsert Algorithm

```
1  def TreeInsert(root, key):
2      if root == None:
3          root = x
4      elif x.key <= root.key:
5          root.left = TreeInsert(root.left, key)
6      elif x.key > root.key:
7          root.right = TreeInsert(root.right, key)
8      else:  # x.key == root.key
9          replace root with x
10     return root  # update x.left, x.right
```

Runtime Analysis:
The worst case runtime occurs when the `elif` branch is always reached in each recursive call.

This causes the insertion behaviour to continually add to the leftmost branch of the tree, with an increasing size each time a duplicate key is inserted.

This results in a worst case runtime of:

$$O(\sum_{i=1}^{n} i) = O(n^2).$$

Question 2

Describe the strategy to ensure duplicate keys are not always inserted on the same side. Use a boolean flag `goLeft` in each node and explain the changes.

Changes made: Return a node with default boolean flag `goLeft` set to `True`. Check if the key is equal to the root's key and toggle the `goLeft` flag. If the flag is `True`, insert the duplicate key on the left side; otherwise, insert it on the right side.

Listing 2: Modified TreeInsert with goLeft Flag

```
def TreeInsert(root, key):
    if root is None:
        return Node(key, goLeft=True)
    if key == root.key:
        if root.goLeft:
            root.left = TreeInsert(root.left, key)
        else:
            root.right = TreeInsert(root.right, key)
        root.goLeft = not root.goLeft
    elif key < root.key:
        root.left = TreeInsert(root.left, key)
    else key > root.key:
        root.right = TreeInsert(root.right, key)
    else:  # x.key == root.key
        replace root with x
    return root
```

Runtime Analysis:
The worst case runtime occurs when the key is always equal to the root's key.

This causes the insertion behaviour to continually swap between the left and right branches of the tree, with an increasing size each time a duplicate key is inserted.

The pattern of insertion is as follows:

- Root, Left, Right, Left, Right, Left, Right, ...

The height of this tree is denoted by $log(n)$, and insertion of each duplicate key will take $O(log(n))$ time.

Alongside, n insertions will be made.

This results in a worst case runtime of:

$$O(n \cdot log(n)).$$

Question 3

Describe the strategy to randomly choose the subtree for duplicate keys. Explain the changes and provide the updated algorithm.

Changes made: A random choice is made for the `goLeft` boolean flag. If the flag is `True`, insert the duplicate key on the left side; otherwise, insert it on the right side.

Listing 3: TreeInsert with Randomized Insertion

```
1  import random
2
3  def TreeInsert(root, key):
4      if root is None:
5          return Node(key)
6      if key == root.key:
7          if random.choice([True, False]):
8              root.left = TreeInsert(root.left, key)
9          else:
10             root.right = TreeInsert(root.right, key)
11     elif key < root.key:
12         root.left = TreeInsert(root.left, key)
13     else:
14         root.right = TreeInsert(root.right, key)
15     return root
```

Worst Case Runtime Analysis:
The worst case runtime occurs when the key is always equal to the root's key.

It is possible that the random choice is always the same, causing the insertion behaviour to continually add to the same branch of the tree, with an increasing size each time a duplicate key is inserted.

This (see Part (a)) results in a worst case runtime of:

$$O(n^2).$$

Question 4

Propose a better strategy for handling duplicate keys and provide a complete algorithm with analysis.

Proposed Strategy: Describe your strategy here.

Listing 4: Proposed TreeInsert Algorithm

```
def TreeInsert(root, key):
    # Your custom logic here
    pass
```