Algorithms and Data Structures Coursework

Question 1 Code

```
class BTNode2:
    def __init__(self,d,l,r):
        self.data = d
        self.left = l
        self.right = r
        self.mult = 1

    # prints the node and all its children in a string
    def __str__(self):
        st = "(" + str(self.data) + ", " + str(self.mult) +") -> ["
        if self.left != None:
            st += str(self.left)
        else: st += "None"
        if self.right != None:
            st += ", "+str(self.right)
        else: st += ", None"
        return st + "]"

class BST2:
    def __init__(self):
        self.root = None
        self.size = 0

    def __str__(self):
        return str(self.root)

    # returns True is the tree is empty, otherwise False
    def isEmpty(self):
        if self.root == None or self.size == 0:
            return True
        return False

    # adds the data d in the tree, increases the size by 1
    # and returns None
    def add(self, d):
        self.root = self._addNodeRec(self.root,d)
        return None
```

```
def _addNodeRec(self, ptr, d):
    if ptr == None:
        self.size += 1
        return BTNode2(d,None,None)
    if d == ptr.data:
        ptr.mult += 1
        self.size += 1
        return ptr
    if d < ptr.data:
        ptr.left = self._addNodeRec(ptr.left,d)
    else:
        ptr.right = self._addNodeRec(ptr.right,d)
    return ptr

# returns the number of times that d is stored in the tree
def count(self, d):
    return self._searchNodeRec(self.root, d)

#Searches for a node with a specific value
def _searchNodeRec(self, ptr, d):
    if ptr == None:
        return 0
    if d == ptr.data:
        return ptr.mult
    if d < ptr.data:
        return self._searchNodeRec(ptr.left,d)
    else:
        return self._searchNodeRec(ptr.right,d)

# removes one occurrence of d from the tree and returns None
# if d does not occur in the tree then it returns without changing the tree
# it updates the size of the tree accordingly
def remove(self,d):
    self.root = self._removeNodeRec(self.root,d)
    return None
```

```
def _removeNodeRec(self, ptr, d):
    if ptr == None:
        return None
    if d < ptr.data:
        ptr.left = self._removeNodeRec(ptr.left,d)
        return ptr
    elif d == ptr.data and ptr.mult > 1:
        ptr.mult -= 1
        self.size -= 1
        return ptr
    elif d > ptr.data:
        ptr.right = self._removeNodeRec(ptr.right,d)
        return ptr
    self.size -= 1
    if ptr.left == ptr.right == None:
        return None
    elif ptr.left == None or ptr.right == None:
        if ptr.left != None:
            return ptr.left
        else:
            return ptr.right
    [minNode, minRemoved] = self._removeMinNode(ptr.right)
    minNode.left = ptr.left
    minNode.right = minRemoved
    return minNode

def _removeMinNode(self, ptr):
    if ptr.left == None:
        return [ptr, ptr.right]
    [minNode, minRemoved] = self._removeMinNode(ptr.left)
    ptr.left = minRemoved
    return [minNode,ptr]
```

Question 1 Explanation

isEmpty(self)

- This method checks whether the size of the tree is 0 or the root of the tree is `None`.
- If either is true, then the tree must be empty as there are no elements, thus the method returns `True`.
- Otherwise it returns `False`.

add(self, d)

- This method calls `_addNodeRec` and sets the root of the old tree to refer to the resulting new Tree
- It then returns `None`

_addNodeRec(self, ptr, d)

- This method recursively transitions the tree using binary search until it finds the same value, `d`, as what is being added to the tree.
- Binary search is done by using a pointer `ptr` to move along the tree in the following manner:
    - If `d` is less than `ptr`'s data, then we check the left branch of the current sub-tree for `d`
    - If `d` is more than `ptr`'s data, then we check the right branch of the current sub-tree for `d`
- If such a node exists then it increases that node's multiplicity by 1 and returns `None`
- Otherwise it locates the relevant free position to insert the new `BTNode2`, at the last node's left or right depending on which subtree was selected; upon doing this it increases the size of the tree by 1, and returns the modified tree

count(self, d)

- This method returns the result of `_searchNodeRec`

_searchNodeRec(self, ptr, d)

- This method recursively transitions the tree using binary search with `ptr` until it locates the node containing the desired data, `d`.
- If it is located, it will return that node's multiplicity.
- Otherwise it returns 0 as the data occurs 0 times in the tree.

remove(self, d)

- This method calls `_removeNodeRec` and sets the root of the old tree to refer to the resulting new Tree
- It then returns `None`

_removeNodeRec(self, ptr, d)

- This method recursively transitions the tree using binary search with `ptr` until it finds a node containing the desired data, `d`, and then lowers the size of the tree by 1.
- If it finds such a node, and its multiplicity is greater than 1, it decreases that node's multiplicity by 1.
- If the tree cannot find the node to be removed within the tree then `None` is returned.
- If the node that must be removed is a leaf node and has multiplicity of 1, the node is simply removed
- If the node has one child and a multiplicity of 1, it returns the child and sets the parent to refer to the child instead, bypassing the parent in the tree
- If the node has two children and a multiplicity of 1, then we find the node whose data is equal or the lowest data greater than the node we wish to remove using `_removeMinNode` on the right subtree
- We then make `minNode's` children equal to `ptr`'s children and return the modified tree

_removeMinNode(self, ptr)

- This method recursively transitions the tree from the current position, ptr, to the left until there is no longer a left.
- At this point, we return where `ptr` is currently, as well as its right pointer, which are then set to `minNode` and `minRemoved` respectively
- We then set `ptr`'s current left pointer to `minRemoved` recursively back up the tree to the original call, and then return `minNode` and `ptr`

Question 2 Code

```
class WTNode:
    def __init__(self,d,l,r,n):
        self.data = d
        self.left = l
        self.right = r
        self.next = n
        self.mult = 0

    # prints the node and all its children in a string
    def __str__(self):
        st = "("+str(self.data)+", "+str(self.mult)+") -> ["
        if self.left != None:
            st += str(self.left)
        else: st += "None"
        if self.next != None:
            st += ", "+str(self.next)
        else: st += ", None"
        if self.right != None:
            st += ", "+str(self.right)
        else: st += ", None"
        return st + "]"

class WordTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def __str__(self):
        return str(self.root)

    # returns True is the tree is empty, otherwise False
    def isEmpty(self):
        if self.root == None or self.size == 0:
            return True
        return False
```

```
    # adds the string st in the tree, increases the size by 1
    def add(self,st):
        if st == "":
            return None
        if self.root == None:
            self.root = WTNode(st[0], None, None, None)
            ptr = self.root
            for pos in range(1,len(st)):
                ptr.next = WTNode(st[pos], None, None, None)
                ptr = ptr.next
        ptr = self.root
        pos = 0
        while pos < len(st):
            while True:
                if st[pos] == ptr.data:
                    if pos == len(st)-1:
                        ptr.mult += 1
                    elif ptr.next == None:
                        ptr.next = WTNode(st[pos+1], None, None, None)
                        ptr = ptr.next
                    else:
                        ptr = ptr.next
                    pos += 1
                elif st[pos] < ptr.data:
                    if ptr.left == None:
                        ptr.left = WTNode(st[pos], None, None, None)
                    ptr = ptr.left
                else:
                    if ptr.right == None:
                        ptr.right = WTNode(st[pos], None, None, None)
                    ptr = ptr.right
                break
        self.size += 1
```

```
    # returns the number of times that string st is stored in the tree
    def count(self, st):
        if self.root == None:
            return 0
        if st == "":
            return None
        ptr = self.root
        i = 0
        while ptr != None:
            if st[i] == ptr.data and i == len(st)-1:
                return ptr.mult
            if st[i] == ptr.data:
                i += 1
                ptr = ptr.next
            elif st[i] < ptr.data:
                ptr = ptr.left
            else:
                ptr = ptr.right
        return 0

    # removes one occurrence of string st from the tree
    # by lowering its multiplicity
    def remove(self,st):
        if self.count(st) == 0 or self.count(st) == None:
            return
        ptr = self.root
        i = 0
        while True:
            if st[i] == ptr.data and i == len(st)-1:
                ptr.mult -= 1
                self.size -= 1
                return
            if st[i] == ptr.data:
                i += 1
                ptr = ptr.next
            elif st[i] < ptr.data:
                ptr = ptr.left
            else:
                ptr = ptr.right
```

Question 2 Explanation

isEmpty(self)

- This method checks whether the size of the tree is 0 or the root of the tree is `None`.
- If either is true, then the tree must be empty as there are no elements, thus the method returns `True`.
- Otherwise it returns `False`.

add(self,st)

- This method adds any string that does not already exist in the tree into its proper place
- If we attempt to add the empty string, then we return `None` and the tree is unchanged
- If the root is `None`, then we simply add the entire string with each character within a separate node to the tree
- Otherwise, we iterate through the tree using `ptr` until we encounter as much of the matching string in consecutive nodes as possible, noted by `pos`:
    - If we reach the end of the string and the final character already exists at that node, i.e. if the entire string already exists in the tree, then we simply increment the node's multiplicity by 1
    - If the string we add only partially matches an already existing string up to a particular character, then we simply add the remaining part of the string into a relevant position within the tree to either side of the differing character
    - If the string does not match any already existing string then it is started in a relevant position to the left or right of highest level node
- At the end of adding any new string to the tree, we increase the size of the tree by 1

count(self,st)

- This method counts the number of occurrences of a particular string by returning the multiplicity of the node corresponding to the last character of the desired string.
- If the tree is empty, we return 0
- If the string provided is empty we return `None`
- Otherwise, we iterate through the tree using `ptr` and check each character of the string, `pos`, against existing character nodes. This is done by comparing the character in the string against the node, and checking whether:
    - `pos` is the same, in which case we move to the next of `ptr`
    - `pos` is less than the node's character, we move to the left of `ptr`
    - Otherwise we move to the right of `ptr`.
- If the entire string matches a string already in the tree, then we return the multiplicity of the final node in the string
- If the current character matches the current node's character then we move to look at the next character in the string and move to the next pointer.
- If the entire string is not found in the tree, then we return 0

remove(self,st)

- This method checks whether the given string exists to be removed, and if it does will locate the node and lower its multiplicity by 1.
- If the count method returns 0 or `None`, then the method should return `None`, as this encompasses the case where the string does not exist in the tree, and therefore cannot be removed, the case where the tree is empty, and the case where the string is empty.
- Otherwise, we iterate through the tree, as we did in `count()`
    - Once the entire string has been located, we decrease the multiplicity of the final node in the string by 1, and then decrease the size of the tree by 1. After this, we return, completing the removal.