```
# part a

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end = False

# Standard Trie
class StandardTrie:
    def __init__(self):
        self.root = TrieNode()

    # Insert word into trie
    def insert(self, word):
        curr = self.root
        for ch in word:
            if ch not in curr.children:
                curr.children[ch] = TrieNode()
            curr = curr.children[ch]
        curr.is_end = True

    # Search prefix and return all words that start with that prefix
    def autocomplete(self, prefix):
        results = []
        curr = self.root

        # First go to the end of the prefix
        for ch in prefix:
            if ch not in curr.children:
                return results  # empty list
            curr = curr.children[ch]

        # Now do DFS from this point
        self._dfs(curr, prefix, results)
        return results

    def _dfs(self, node, path, results):
        if node.is_end:
            results.append(path)
        for ch in node.children:
            self._dfs(node.children[ch], path + ch, results)

# Test
print("Standard Trie Example")
words = ["Sample", "Samplers", "Same", "Sampling", "Summer", "Sad"]
std_trie = StandardTrie()
for w in words:
    std_trie.insert(w)

print("Suggestions for 'Sam':", std_trie.autocomplete("Sam"))

#part b

# Ternary Trie Node
class TernaryNode:
    def __init__(self, char):
        self.char = char
        self.left = None
        self.middle = None
        self.right = None
        self.is_end = False

# Ternary Trie
class TernaryTrie:
    def __init__(self):
        self.root = None

    def insert(self, word):
        if word:
            self.root = self._insert(self.root, word, 0)

    def _insert(self, node, word, index):
        ch = word[index]

        if node is None:
            node = TernaryNode(ch)

        if ch < node.char:
            node.left = self._insert(node.left, word, index)
        elif ch > node.char:
            node.right = self._insert(node.right, word, index)
        else:
            if index + 1 < len(word):
                node.middle = self._insert(node.middle, word, index + 1)
            else:
                node.is_end = True
        return node

    def autocomplete(self, prefix):
        results = []
        node = self._search(self.root, prefix, 0)
        if node:
            if node.is_end:
                results.append(prefix)
            self._dfs(node.middle, prefix, results)
        return results

    def _search(self, node, prefix, index):
        if node is None or index >= len(prefix):
            return None

        ch = prefix[index]
        if ch < node.char:
            return self._search(node.left, prefix, index)
        elif ch > node.char:
            return self._search(node.right, prefix, index)
        else:
            if index == len(prefix) - 1:
                return node
            return self._search(node.middle, prefix, index + 1)

    def _dfs(self, node, path, results):
        if node is None:
            return
        self._dfs(node.left, path, results)
        if node.is_end:
            results.append(path + node.char)
        self._dfs(node.middle, path + node.char, results)
        self._dfs(node.right, path, results)

# Test
print("\nTernary Trie Example")
words = ["Sample", "Samplers", "Same", "Sampling", "Summer", "Sad"]
ternary_trie = TernaryTrie()
for w in words:
    ternary_trie.insert(w)

print("Suggestions for 'Sam':", ternary_trie.autocomplete("Sam"))
```