Tries Data Structure Planning Notes

*Before writing any code I used this document to get my thoughts together and try to create a plan for attacking the project. This is not a complete document but reflects my planning process.*

Description
A trie, short for "retrieval tree" or "prefix tree," is a hierarchical tree-based data structure used for efficient storage and retrieval of strings or keys. It provides fast search, insertion, deletion, and prefix matching operations.

Abstract and Concrete definitions. User story perspective.

Invariants
- Each node in the trie has at most 26 children corresponding to the letters of the alphabet.
- Every path from root node to word ending node represents a unique word in the trie, ensuring unique representation and retrievability.
- The height of the trie is equal to the longest word inserted.
- The root node has no parent and no associated character.
- Each node maintains a numerical count indicating how many words end at that node.
- Words that share prefixes share nodes and separate at the end of their common prefix.
- The trie maintains a hierarchical tree structure with parent-child relationships and no cycles.

Supporting Data Structures:
- Hash map for storing pointers
- Sets for the alphabet
- Vector

Helper Functions
- Generate an alphabet set

Complexity Analysis? Runtime and Space

Trie Node
- Is Root
- Is Word End
- Word Count
- Next Map/Dictionary of pointers to nodes
- Character(s)

Init Trie Node (string)

- Creates a new trie node with default values
- Stores string in character(s) property

Trie Class

- Properties:
  - Root node
    - For accessing the trie
  - Number of words inserted.
  - Number of nodes/characters
  - Alphabet or list of valid characters
- Methods
  - Constructor (alphabet)
  - New Trie Node()
  - Insert (word)
  - Delete(word)
  - Word Count (word)
    - Returns the number of times that word has been inserted
    - Can be modified as a search, 0 would return false and 1 or more would return true.
  - Autocomplete(prefix)
    - Returns a list of words that match that prefix
    - Extend this method for the longest word matching the prefix, the longest word in the list.
    - Extend this method with the number of items in the list for number of words with that prefix.
  - Size()
    - Return the number of words inserted
  - Characters()
    - Return the number of characters stored in the tree
  - Nodes()
    - Return the number of nodes in the tree (could be interesting for the radix tree where nodes and character counts would not be the same).
  - Clear()
    - Removes all words and nodes from the trie
- Helper Functions for Compression Trees (Optional)
  - Compress nodes/characters
  - Decompress nodes/characters

Tests

- Test Initialize Node:
  - Normal Behavior

        - Check that a new node has all the correct default values
    - Edge cases
        - Invalid input, should fail
- Test initialize trie
    - Normal Behavior
        - Check that new trie has correct default values
    - Edge Cases
        - Invalid input should fail
- Test Insert
    - Create a new trie
    - Normal Behavior
    - Insert the word “automate”
        - Check that 8 characters were inserted
        - Check that word count is 1 on the last node
    - Insert the word “automobile”
        - Check that 5 characters were inserted
        - Check that word count is 1 one on the last node
        - Check that both “automate” and “automobile” share the prefix “autom-“
    - Insert the word “inspect”
        - Check that a new branch was formed and that it has 7 characters
        - Check that the last word has a one in the word count
    - Insert the word “introvert”
        - Check that 7 characters were inserted
        - Check that the word count is 1 on the last node
        - Check that both “inspect” and “introvert” share the same prefix
    - Insert the word “in”
        - Check that there is a word ending node after the edge n
    - Insert the word “automate”
        - Check that the word count went from 1 to 2.
    - Edge Cases
        - Insert non alphabet characters
        - Insert empty string
        - Insert the wrong data type
    -
- Test Delete
    - Normal Behavior
    -
- Test Word Count
- Test Autocomplete
- Test Size
- Test Characters
- Test Nodes

Pseudocode

```
Init node(c)
        New trie node
        Is root = false
        Is word end = false
        Word count = 0
        Characters = ‘c’
        Next = new hash table {}

Insert(word)
        Cursor = root node
        For c in word:
                If c not in cursor.next (it’s not in the pointers to next nodes)
                        Cursor.next[c] = new trienode(c)
                        Trie nodes += 1
                        Trie nodes characters += 1
                Cursor = cursor.next[c]
        Cursor is word ending node = true
        Cursor Word count += 1
        Return

Word count (word)
        Cursor = root node
        For c in word:
                If c not in cursor.next:
                        Return 0
                Cursor = cursor.next[c]
        Return cursor word count

Autocomplete(prefix)
        Word list = []
        Cursor = root
        For c in prefix:
                If c not in cursor.next:
                        Return word list
                Cursor = cursor.next[c]
        Autocomplete Recurse (cursor, prefix, word list)
        Return word list

Autocomplete Recurse (subtree, prefix, word list)
```

```
If subtree is word End:
        Insert prefix, subtree word count in to word list
For key in subtree next:
        Autocomplete Recurse(subtree.next[key], prefix + key, word list)
Return
```