2185. Counting Words With a Given Prefix

You are given an array of strings `words` and a string `pref`.

Return *the number of strings in* `words` *that contain* `pref` *as a* ***prefix***.

A **prefix** of a string `s` is any leading contiguous substring of `s`.

Example 1:

```
Input: words = ["pay","attention","practice","attend"], pref = "at"
Output: 2
Explanation: The 2 strings that contain "at" as a prefix are: "attention" and
"attend".
```

Example 2:

```
Input: words = ["leetcode","win","loops","success"], pref = "code"
Output: 0
Explanation: There are no strings that contain "code" as a prefix.
```

Constraints:

- `1 <= words.length <= 100`
- `1 <= words[i].length, pref.length <= 100`
- `words[i]` and `pref` consist of lowercase English letters.

2185. Counting Words With a Given Prefix

```
/*
   STL find
   Time complexity: O(n.m.k),  WC: k=m => O(nm^2)
   Space complexity: O(1)
   n: length of the given list of words
   m: length of the longest word in the given list words
   k: length of the given prefix
*/
class Solution {
   public:
      int prefixCount(std::vector<std::string>& words, std::string pref) {
         int ans=0;
         for(auto& word: words){
            if(word.find(pref)==0) ans++;
         }
         return ans;
      }
};
```

2185. Counting Words With a Given Prefix

```
/*
   STL starts_with
   Time complexity: O(n.k), WC: k=m => O(nm)
   Space complexity: O(1)
   n: length of the given list of words
   m: length of the longest word in the given list words
   k: length of the given prefix
*/
class Solution {
   public:
      int prefixCount(std::vector<std::string>& words, std::string pref) {
         int ans=0;
         for(auto& word: words){
            if(word.starts_with(pref)) ans++;
         }
         return ans;
      }
};
```

2185. Counting Words With a Given Prefix

Prefix tree (Trie)

The process of matching characters sequentially from the beginning aligns perfectly with the concept of a Trie. A Trie is a specialized tree-like data structure designed to handle strings efficiently, especially for operations like ***prefix searches***, ***word insertions***, and ***word lookups***.

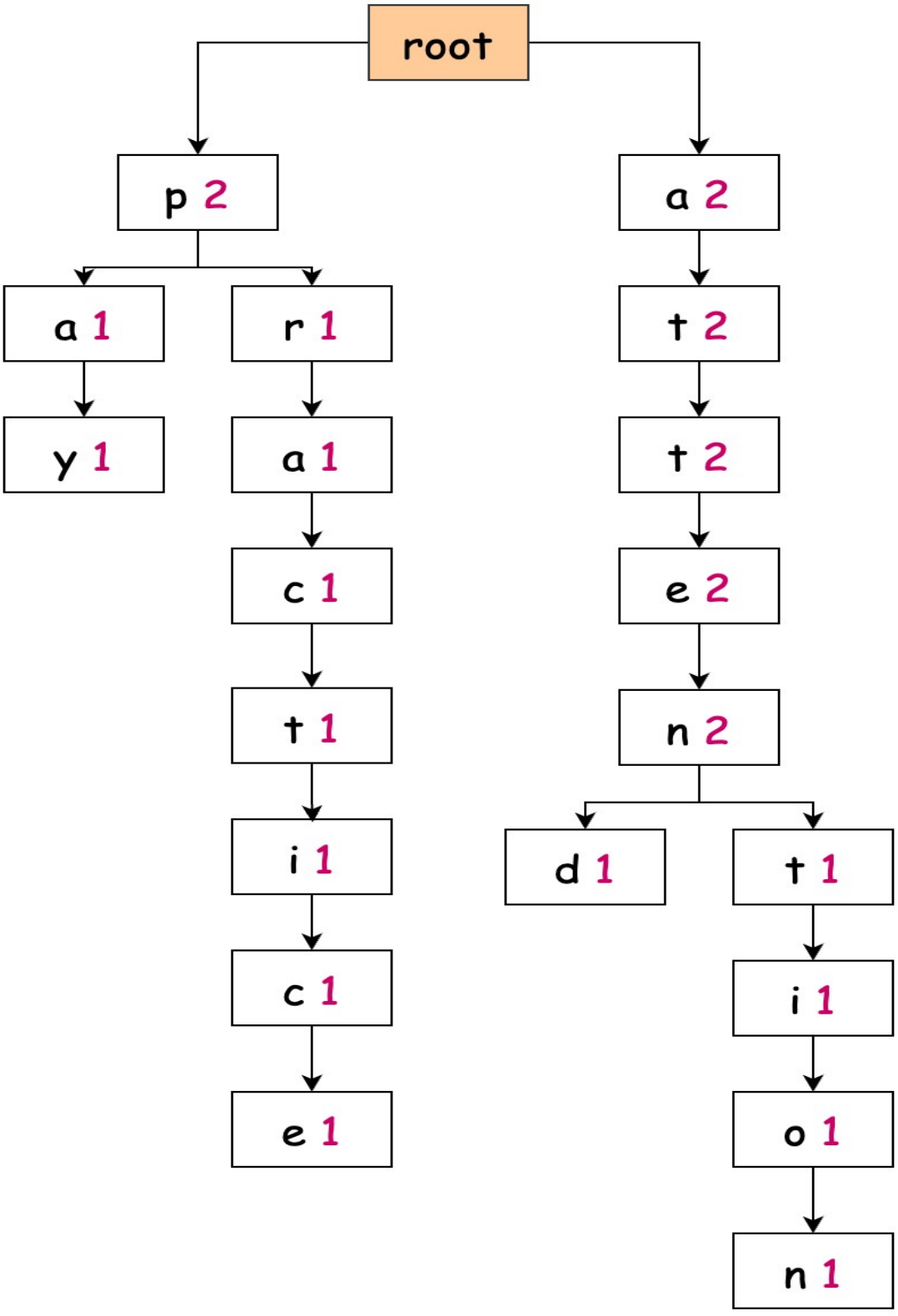
Each node in a Trie represents a single character, and the path from the root node to any other node forms a prefix or a word.

Usually a Trie is used in two below ways:

- **Insertion**: When inserting a word into the Trie, we start at the root and traverse down the tree, creating new nodes for each character of the word if they don't already exist. At the end of the word, we mark the final node to signify the completion of the word.
- **Search for a Prefix**: To check if a word starts with a given prefix, we simply traverse the Trie following the nodes corresponding to each character of the prefix. If we can traverse all characters successfully, the prefix exists in the Trie.

For example, if we built a Trie using the words array in *Example #1* of the problem description:

`words = ["pay","attention","practice","attend"], pref = "at"`
this is how it would look like:


root
p 2
a 2
a 1
r 1
t 2
y 1
a 1
t 2
c 1
e 2
t 1
n 2
i 1
d 1
t 1
c 1
i 1
e 1
o 1
n 1

What makes Tries particularly powerful is their ability to efficiently handle prefix-based operations. *Think about how you use autocomplete on your phone - as you type each character, it quickly suggests words that start with those letters*. **Tries are thus a natural choice for problems involving prefixes or auto-completion**. By structuring the characters hierarchically, they allow for fast and intuitive access to any subset of stored strings.

Our version of the Trie has a unique feature - instead of just marking where words end, we keep count of how many words share each prefix. For example, if *three words begin with "cat"*, then after we reach 't' in the Trie, that node would show a count of 3.

To build this solution, we start by designing our Trie structure. Each node needs two essential pieces: links to its children (representing the next possible characters) and the count variable. **Since we're working with lowercase English letters, we can use an array of size 26 for the links**, where **each index represents a character *(a = 0, b = 1, etc.)***. This array approach gives us constant-time access to child nodes. The definition if a trie node will be:

```
class TrieNode{
        public:
           TrieNode* children[26]={nullptr};
           int count=0;
};
```

The definition of the whole trie:

```
class Trie{
    public:
        class TrieNode{
            public:
                TrieNode* children[26]={nullptr};
                int count=0;
        };
        // Defintion of the root node
        TrieNode* root;
    public:
        // Constructor that creates the root node
        Trie(){
            root=new TrieNode();
        }
         // Destructor that deletes the trie to avoid memory leaks
        ~Trie(){};
    // Insert a word into the trie
    void insert(std::string& s){}
    // Compute how many word have s as a prefix
    int compute(std::string& s){}
};
```

2185. Counting Words With a Given Prefix

```
/*
   Trie
   Time complexity: O(n.m+k),  WC: k=m => O(nm)
   Space complexity: O(nm)
   n: length of the given list of words
   m: length of the longest word in the given list words
   k: length of the given prefix
*/
class Trie{
   public:
      class TrieNode{
         public:
            TrieNode* children[26]={nullptr};
            int count=0;
      };
      TrieNode* root;
   public:
      Trie(){
         root=new TrieNode();
      }


      ~Trie(){delete_trie(root);}

      // Delete the try to avoid memory leaks
      void delete_trie(TrieNode* root){
         if(!root) return;
         for(int i=0;i<26;++i){
            TrieNode* node=root->children[i];
            delete_trie(node);
         }
         delete root;
      }
```

```
    void insert(std::string& s){
        TrieNode* cur=root;
        for(auto& c: s){
            int i=c-'a';
            TrieNode* node=cur->children[i];
            if(!node){
                node=new TrieNode();
                cur->children[i]=node;
            }
            cur->children[i]->count++;
            cur=node;
        }
    }

    int compute(std::string& s){
        TrieNode* cur=root;
        for(auto& c: s){
            int i=c-'a';
            TrieNode* node=cur->children[i];
            if(!node) return 0;
            cur=node;
        }
        return cur->count;
    }
};

class Solution {
public:
    int prefixCount(std::vector<std::string>& words, std::string pref){
        int k=pref.size();
        Trie trie=Trie();
        for(auto& word: words) trie.insert(word);

        int ans=trie.compute(pref);
        return ans;
    }
};
```