NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

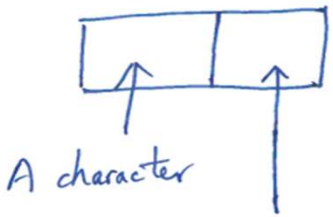
Tutorial 6 Trie

Retrieval

↓

Trie ≡ Prefix tree ≡ Digital tree

26 pointers



A boolean value to represent whether this character represents the end of a word.

Trie Node Declaration:

```
class TrieNode {
    boolean isEndOfWord;
    TrieNode children[];

    public TrieNode(){
        isEndOfWord = false;
        children = new TrieNode[26];
    }
}
```

Insert in Trie:

When we insert a character (part of a key) into a Trie, we start from the root node and then search for a reference, which corresponds to the first key character of the string whose character we are trying to insert in the Trie. Two scenarios are possible:

- A reference exists, so then we traverse down the tree following the reference to the next children level.
- A reference does not exist, then we create a new node and refer it with parents reference matching the current key character. We repeat this step until we get to the last character of the key, then we mark the current node as an end node and the algorithm finishes.

```java
public void insert(String word) {
    TrieNode node = root;

    for (char c : word.toCharArray()) {
        if (node.children[c-'a'] == null) {
            node.children[c-'a'] = new TrieNode();
        }
        node = node.children[c-'a'];
    }
    node.isEndOfWord = true;
}
```

Search in Trie:

A key in a Trie is stored as a path that starts from the root node, and it might go all the way to the leaf node or some intermediate node. If we want to search for a key in a Trie, we start with the root node and then traverses downwards if we get a reference match for the next character of the key we are searching, then there are two cases:

A reference for the next character exists; hence we move downwards following this link, and proceed to search for the next key character.

A reference does not exist for the next character. If there are no more characters of the key present and this character is marked as isEndOfWord = true, then we return **true**, implying that we have found the key. Otherwise, two more cases are possible, and in each of them we return **false**. These are:

There are key characters left in the key, but we cannot traverse down as the path is terminated, hence the key doesn't exist.

No characters in the key are left, but the last character is not marked as isEndOfWord = false. Therefore, the search key is just the prefix of the key we are trying to search in the Trie.

```java
public boolean search(String word) {
    return isMatch(word, root, 0, true);
}

public boolean startsWith(String prefix) {
    return isMatch(prefix, root, 0, false);
}

public boolean isMatch( String s, TrieNode node, int index, boolean isFullMat
    if (node == null)
        return false;

    if (index == s.length())
        return !isFullMatch || node.isEndOfWord;

    return isMatch(s, node.children[s.charAt(index) - 'a'], index + 1, isFull
}
```

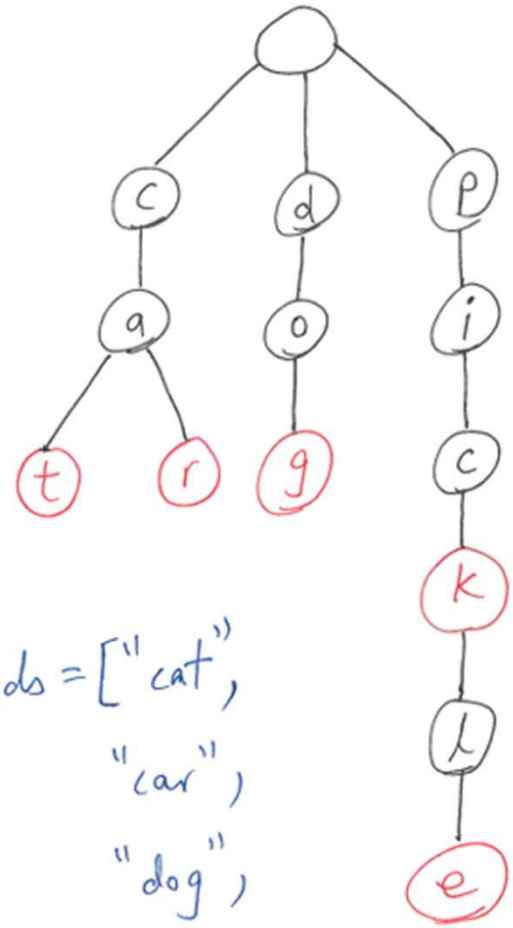
Trie are more efficient than hash tables

→ won't be any collisions

→ no need for hash functions

→ look up time for a string in trie
is $O(k)$ where k = length of the word

→ can take even less than $O(k)$ time
when the word is not there in the trie.




```
words = ["cat",
         "car",
         "dog",
         "pick",
         "pickle"]
```

Question 1

- You are given a Trie that stores multiple words. Write a function count_words() to count how many words are stored in the Trie. The function prototype is given as follow:

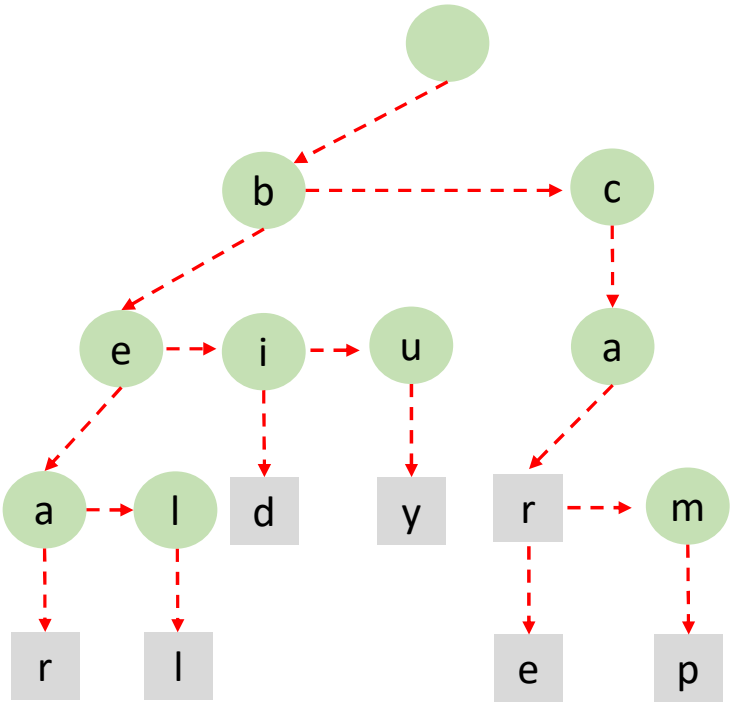
```
def count_words(self, node):
```

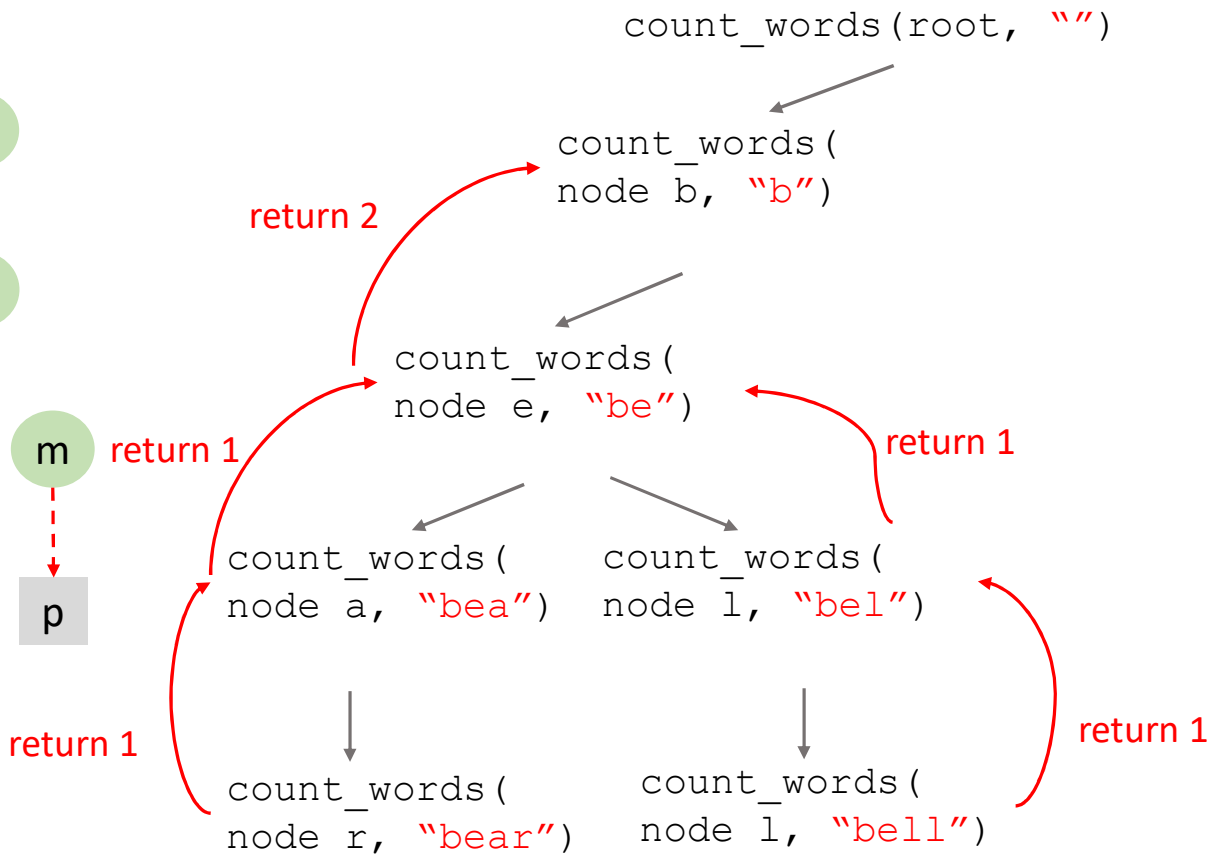
Question 1

```
def print_all_words_dfs(self, node,
prefix):
    if node.is_end_of_word:
        print(prefix)
    child = node.first_child
    while child:
        self. print_all_words(child,
                        prefix+child.char)
        child = child.next_sibling
```

```
def count_words(self, node):
    if node.is_end_of_word:
        count = 1
    else: count = 0
    child = node.first_child
    while child:
        count =
count+self.count_words(child,
                        prefix+child.char)
        child = child.next_sibling
    return count
```

Question 1


b
c
e
i
u
a
a
l
d
y
r
m
r
l
e
p

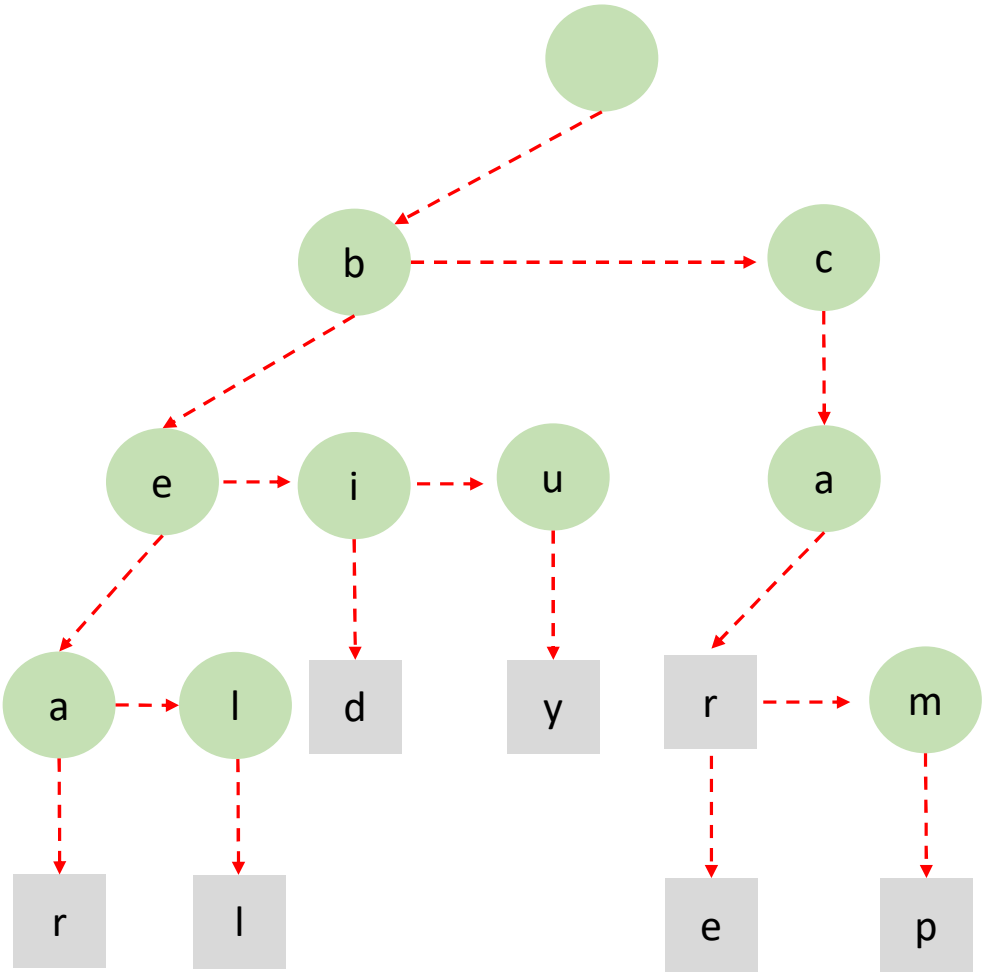

count_words(root, "")
count_words(
node b, "b")
return 2
count_words(
node e, "be")
return 1
return 1
count_words(
node a, "bea")
count_words(
node l, "bel")
return 1
count_words(
node r, "bear")
count_words(
node l, "bell")

Question 2

- Given a Trie that stores multiple words, implement a function find_words_with_prefix() that returns all words that start with a given prefix. The function prototype is given as follow:

```
def find_words_with_prefix(self,node,prefix):
```

Application Example: Autocomplete



- Traverse the Trie to the node matching the prefix, e.g., “ca”
- Perform dfs/bfs to collect all complete words
- Return the words based on some rules

Question 2

```
def collect_all_words_dfs(self, node, prefix,
results):
    if node.is_end_of_word:
        results.append(prefix)
    child = node.first_child
    while child:
        self. collect_all_words_dfs(child,
                  prefix+child.char, results)
        child = child.next_sibling
def _find_child(self, node, char) :
    current = node.child
    while current:
        if current.char == char:
            return current
        current = current.next
    return None
```

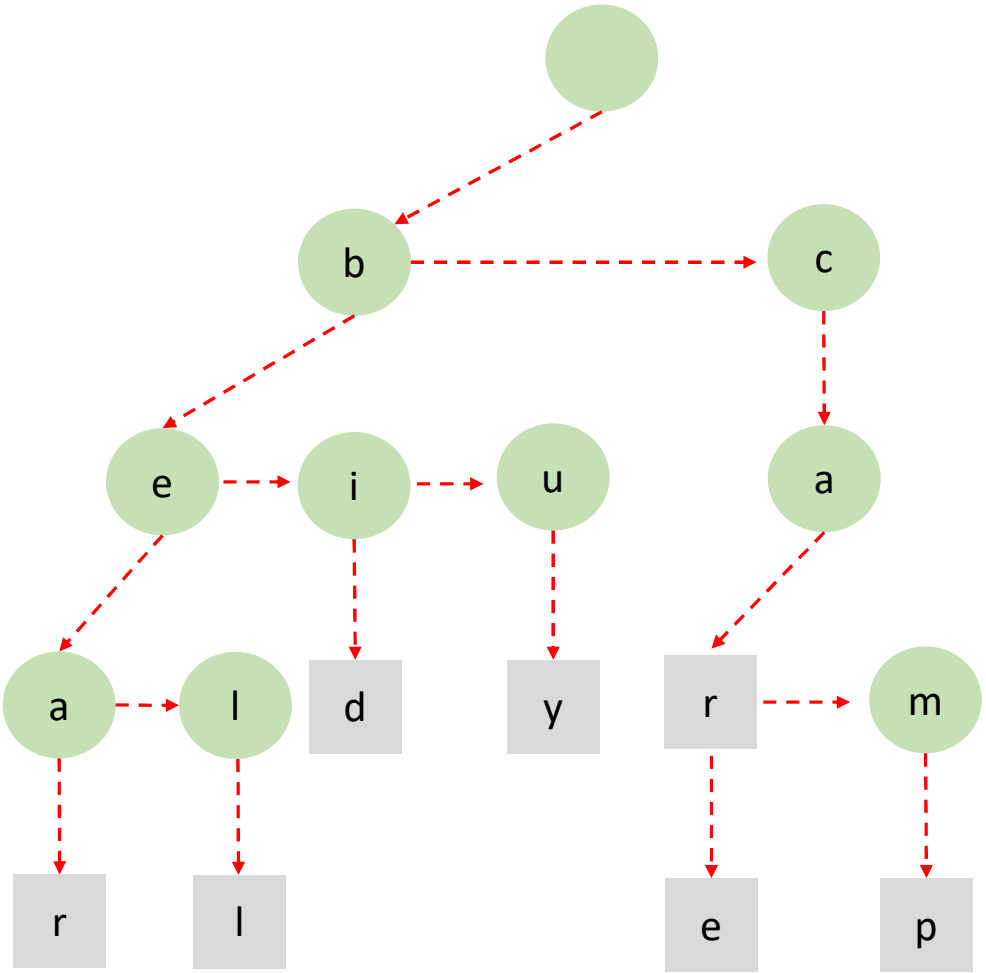
```
def find_words_with_prefix(self,node,prefix):
#Use a list to collect the words
    results = []
    node = self.root
#Step 1: Traverse the Trie to the node
matching the prefix
    for char in prefix:
        node = self._find_child(node, char)
        if not node:
            return []
#Step2: Perform dfs to collect all complete
words
    self.collect_all_words_dfs(node, prefix,
                                        results)
    return results
```

Question 3

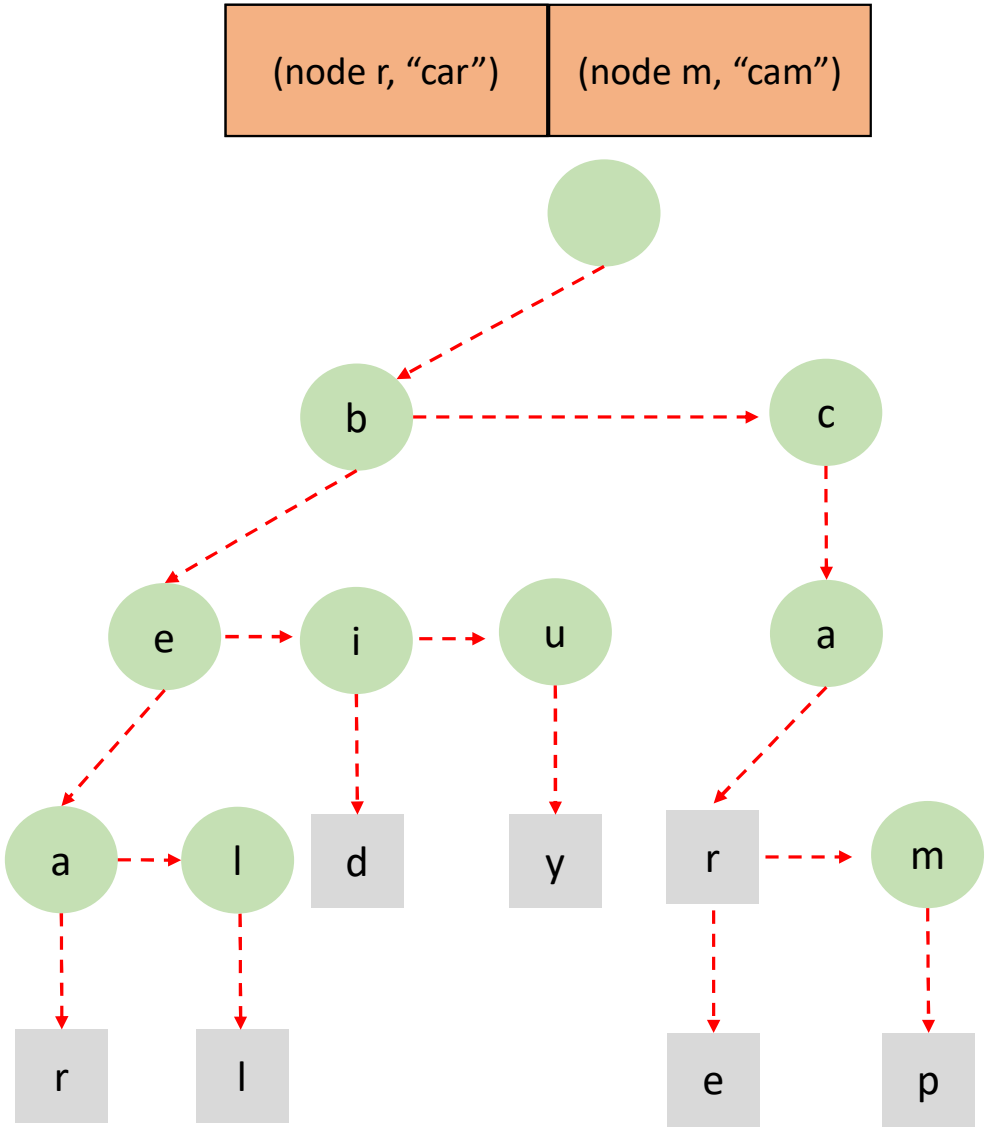
- Given a Trie storing multiple words, write a function find_shortest_word_with_prefix() that returns the shortest word that starts with a given prefix. If no word starts with the prefix, return None.

```
def find_shortest_word_with_prefix(self,node,prefix):
```

Question 3



- Traverse the Trie to the node matching the prefix, e.g., “ca”
- Perform bfs from the ending node of the prefix
- The first complete word will be the shortest one.




```
def find_shortest_word_with_prefix(self, prefix)
#Step 1: Traverse to the end of the prefix
    node = self.root
    for char in prefix:
    node = self._find_child(node, char)
    if not node:
        return None
# Step 2: BFS
     queue = Queue()
     queue.enqueue((node, prefix))
     while not queue.is_empty():
         current_node, path = queue.dequeue()
         if current_node.is_end_of_word:
             return path
         child = current_node.first_child
         while child:
             queue.enqueue((child,path+child.char))
             child = child.next_sibling
         return None
```