

SC1007

Trie

Dr Liu Siyuan

Email: syliu@ntu.edu.sg

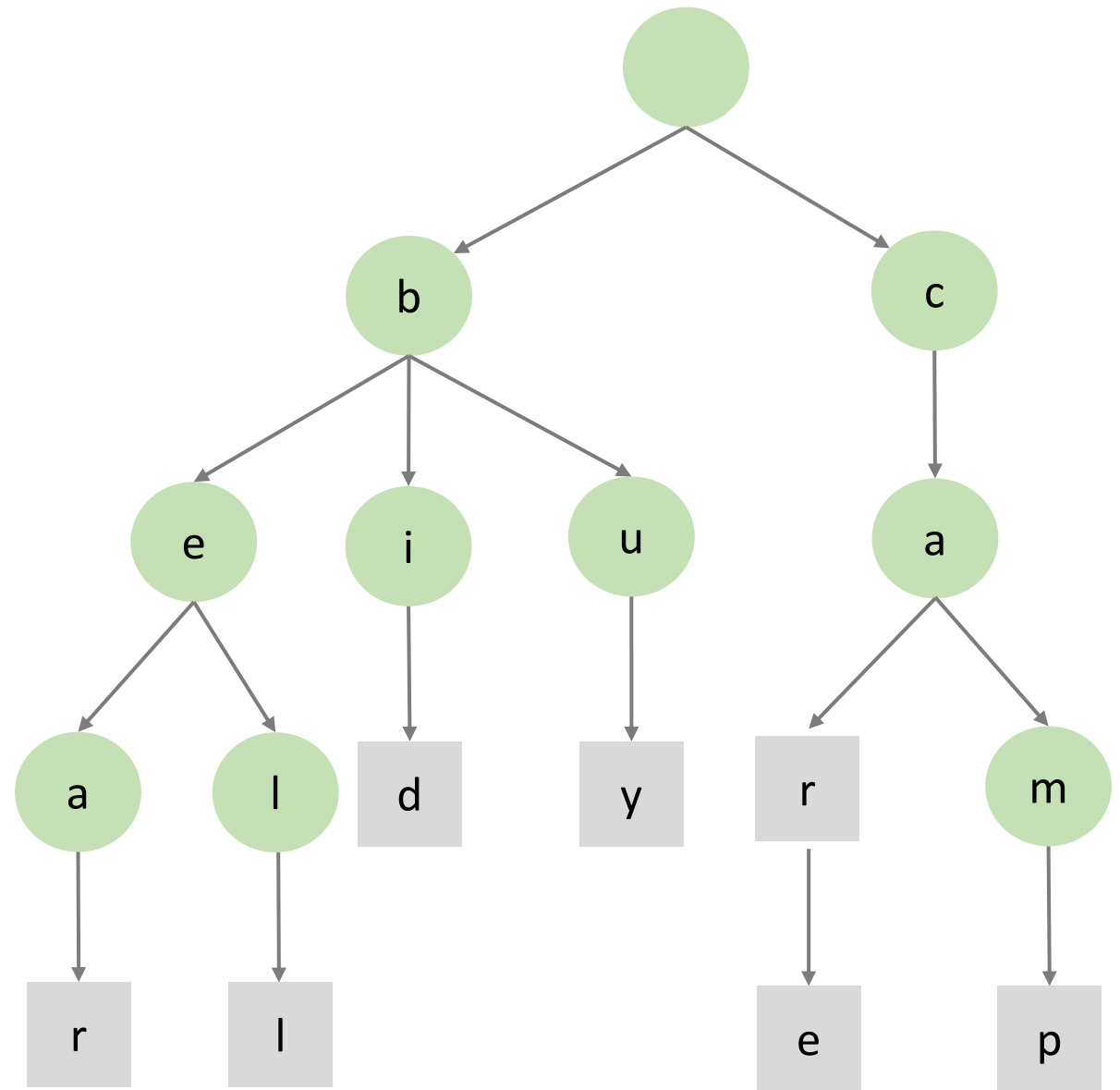
Office: N4-02C-72a

Overview

- What is a trie
- Implementations with linked list
 - Insert a word
 - Search a word
 - Traversal of a trie
- Application examples
 - Print all words
 - Autocomplete
 - Spell checking

What Is a Trie

- A tree-based data structure used for efficient string operations. Also called prefix tree or digital tree.
- It is a specialized search tree data structure used to store and retrieve strings from a dictionary or set.



The trie structure for strings: bear, bell, bid, buy, car, care, camp

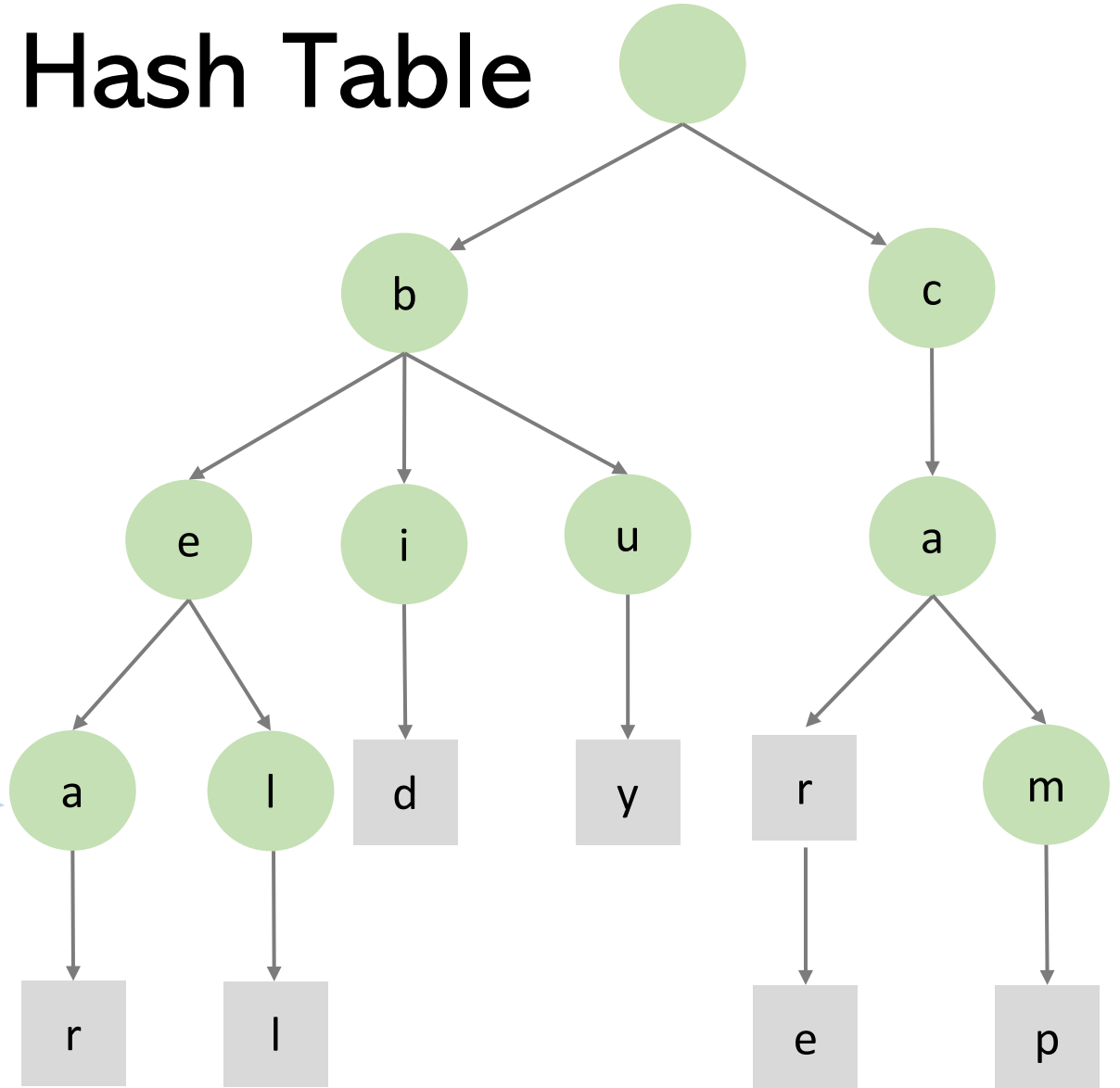
Applications of Trie

- Autocomplete
 - Google Search and mobile keyboards use tries to suggest completions as you type. This allows fast lookups of possible word completions.
- Spell Checking
 - Tries enable efficient verification of word existence in dictionaries. They also help generate word suggestions for misspelled words.
- Technical Applications
 - IP routing tables use tries for address lookup. Compression algorithms leverage tries for pattern recognition.

Implementations with Hash Table

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

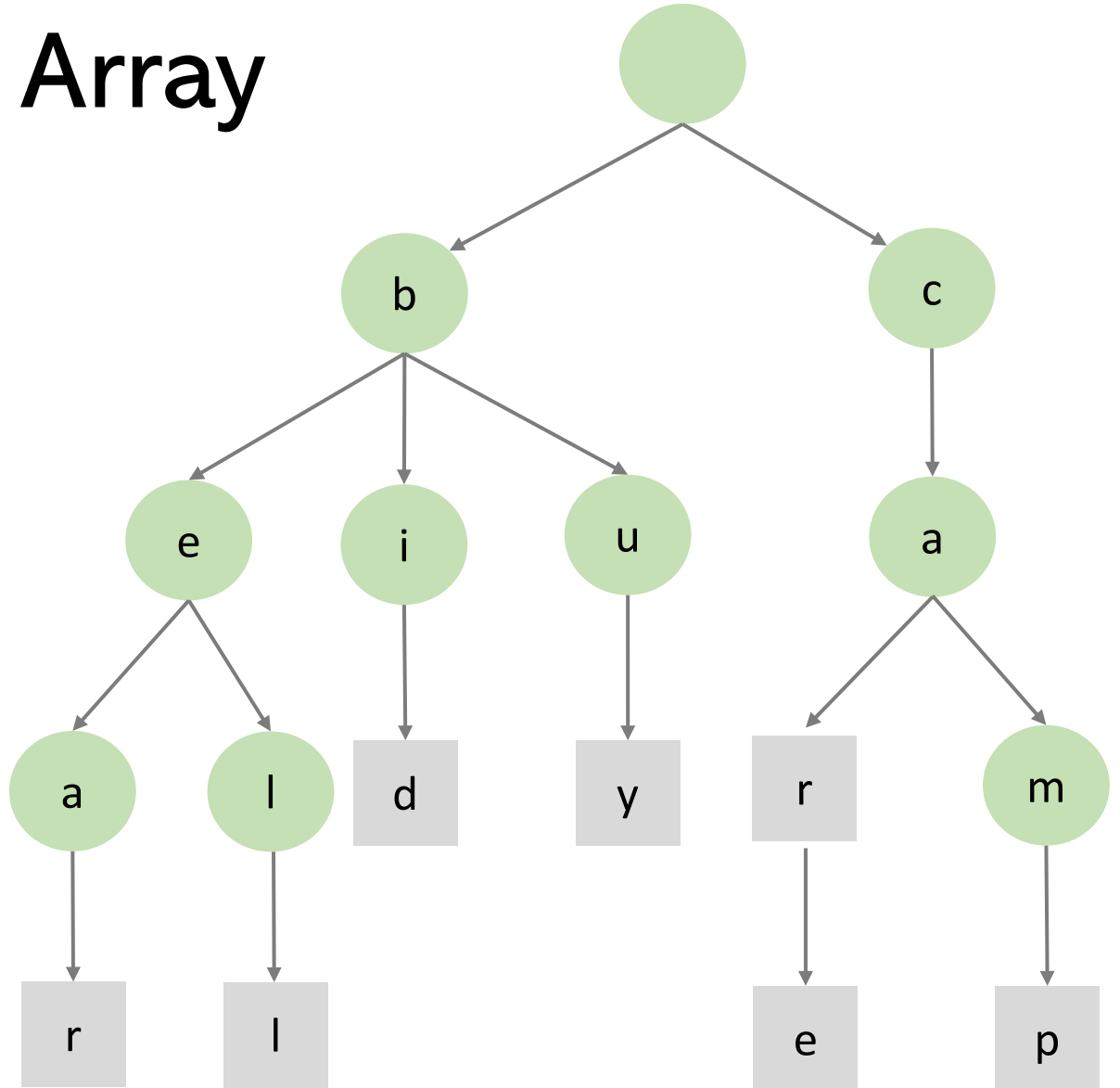
```
children = {'r':TrieNode(children={},
                           is_end_of_word=True)}
is_end_of_word = False
```



Implementations with Array

```
class TrieNode:
    def __init__(self):
        self.children = [None] * 26
        self.is_end_of_word = False
```

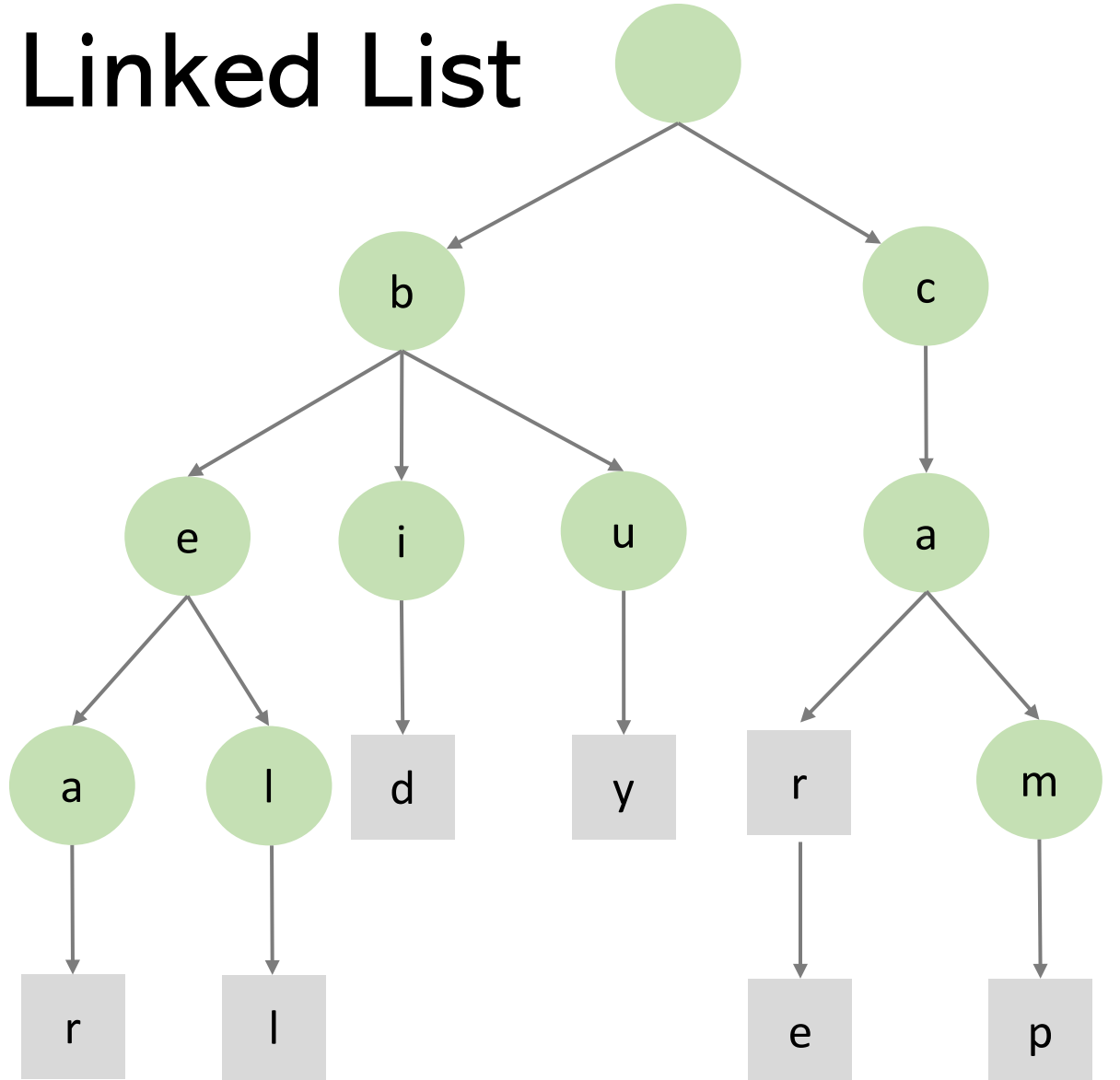
```
child[17] = TrieNode(
    children = [None]*26,
    is_end_of_word = True)
end_of_word = False
```



Implementations with Linked List

```
class TrieNode:
    def __init__(self, char):
        self.char = char
        self.is_end_of_word = False
        self.child = None
        self.next = None
```

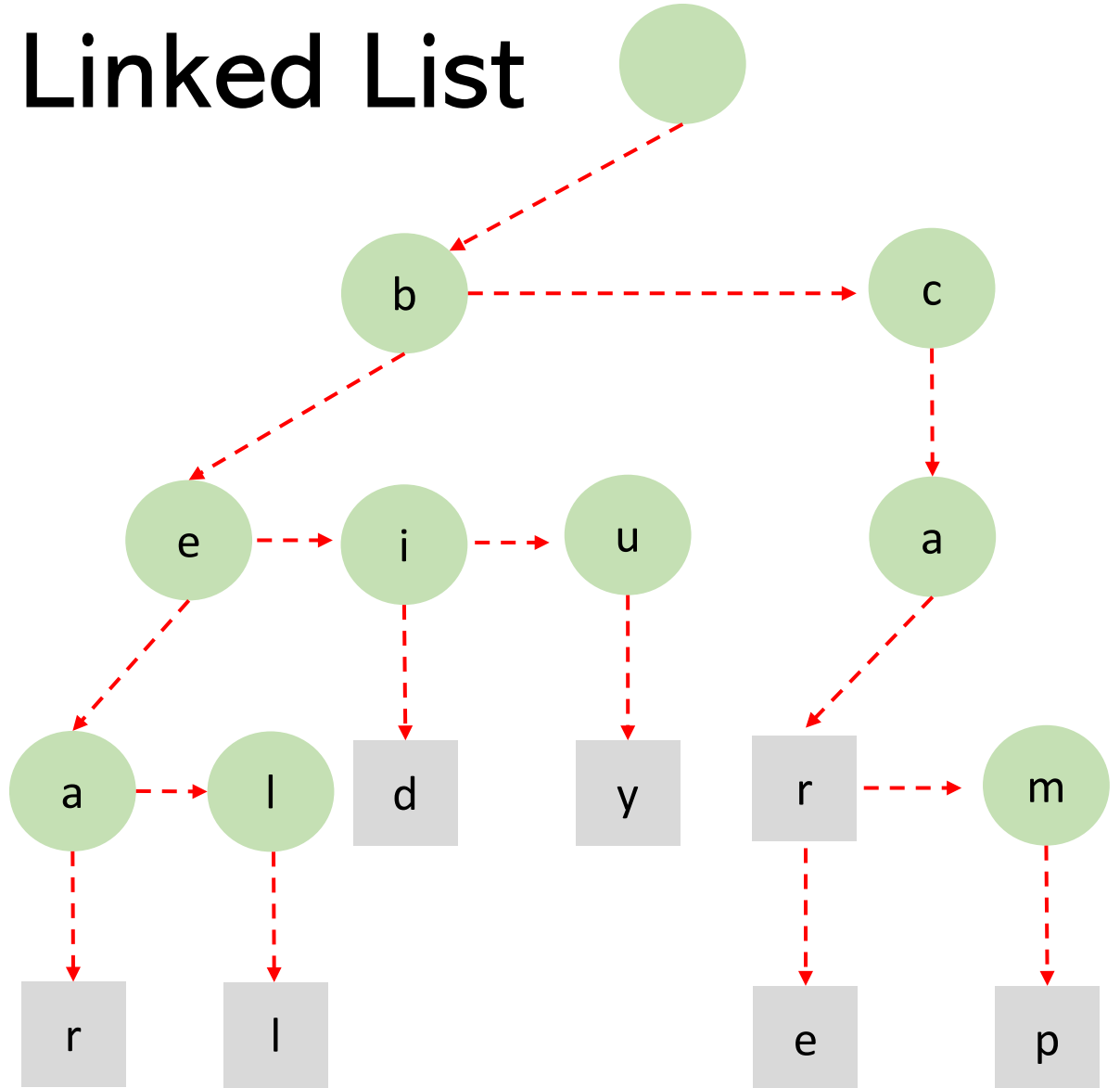
char = 'a'
end_of_word = False
child = TrieNode('r')
next = TrieNode('l')



Implementations with Linked List

```
class TrieNode:
    def __init__(self, char):
        self.char = char
        self.is_end_of_word = False
        self.child = None
        self.next = None
```

char = 'a'
end_of_word = False
child = TrieNode('r')
next = TrieNode('l')



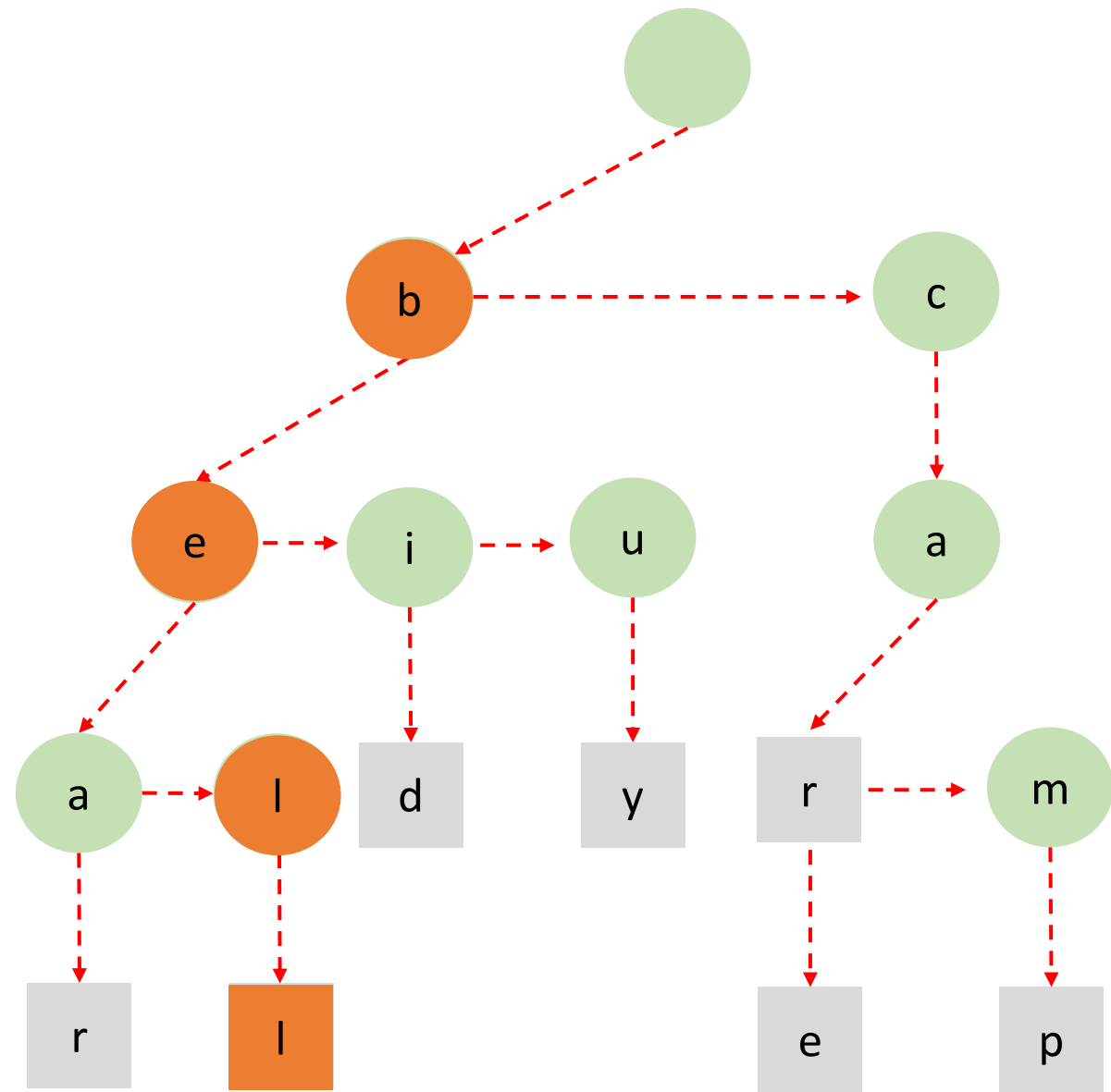
Implementations with Linked List

- The core operations for a trie:
 - Search a word
 - Insert a word
 - Traversal
- Usually we will not delete a word from a trie
 - Dictionaries don't usually change
 - Deleting from a trie is much more complex than inserting
- The binary tree traversal algorithms can be applied in trie
 - Preorder (dfs)
 - Level-by-level (bfs)

```
class Trie:
    def __init__(self):
        self.root = TrieNode("")
    def search(self, word):
    def insert(self, word):
    def dfs(self, node):
    def bfs(self, node):
```

Search a Word

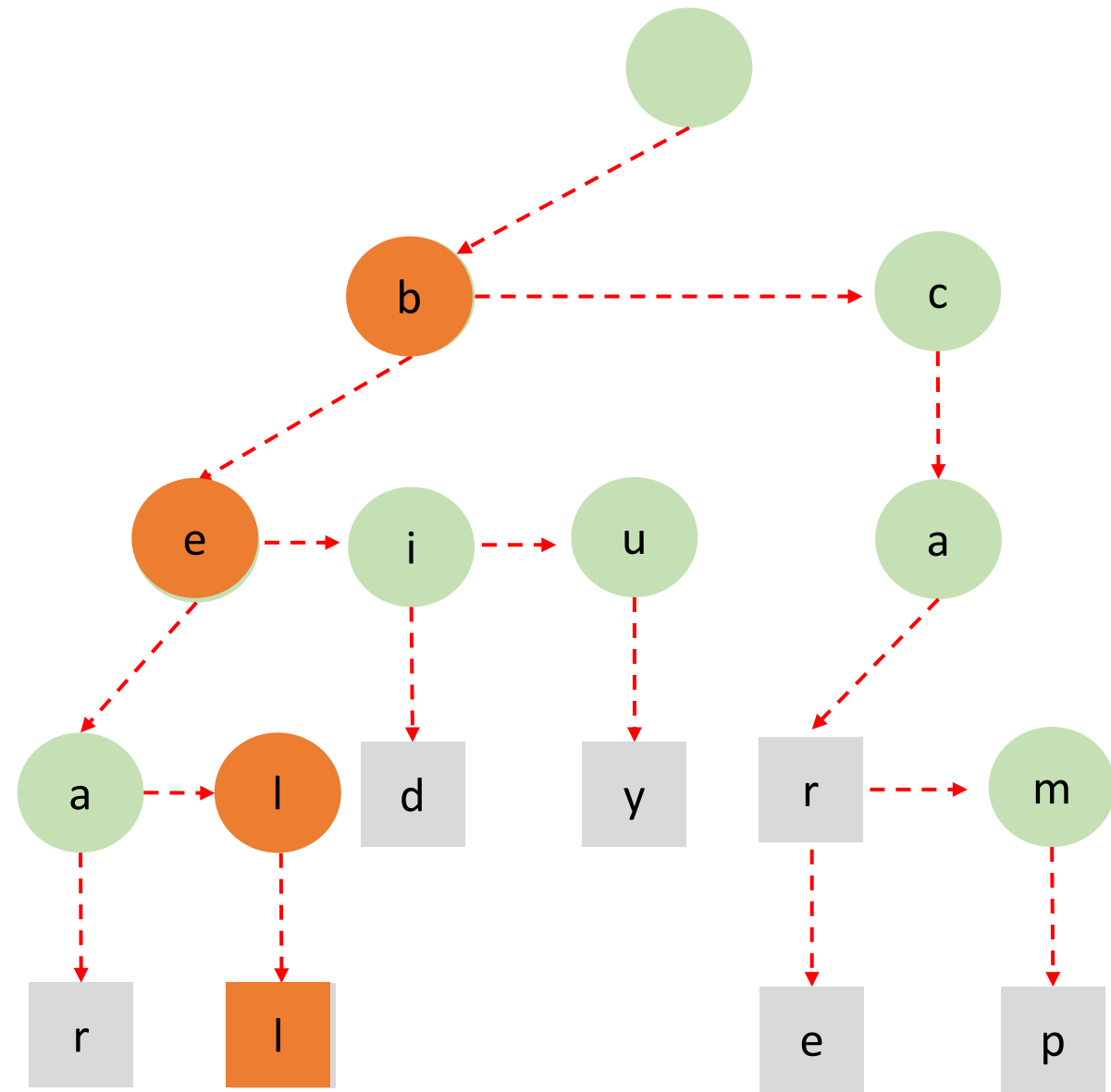
```
parent_node = root
for each character in the word
    if the current character is a
       child of parent_node:
        parent_node = current_node
        move on to the next character
    else:
        return False
return current_node.is_end_of_word
```



For example, search “bell”

Search a Word

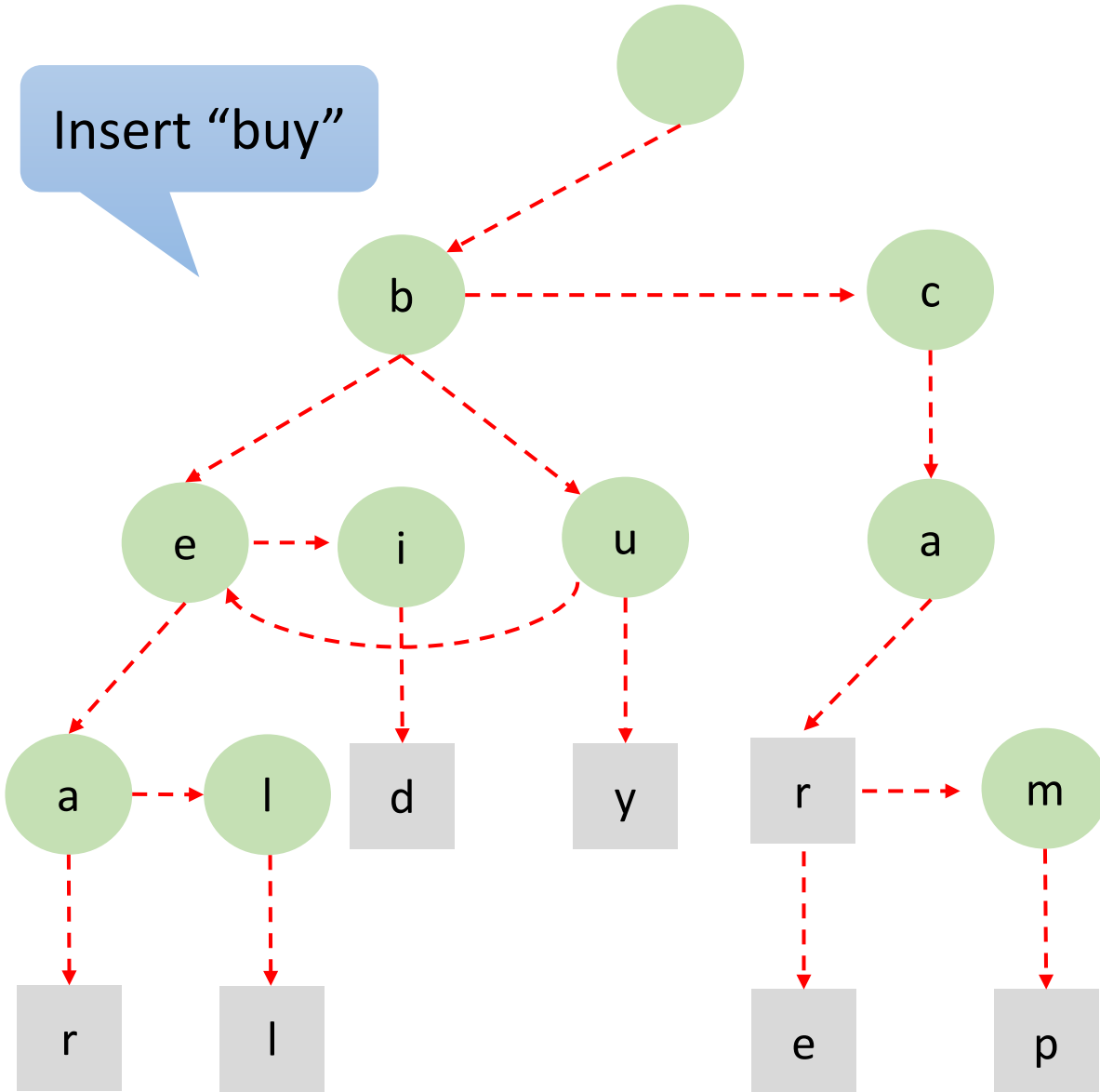
```
def _find_child(self, node, char):  
    current = node.child  
    while current:  
        if current.char == char:  
            return current  
        current = current.next  
    return None  
  
def search(self, word):  
    node = self.root  
    for char in word:  
        node = self._find_child(node, char)  
        if not node:  
            return False  
    return node.is_end_of_word
```



For example, search “bell”

Insert a Word

Insert "buy"

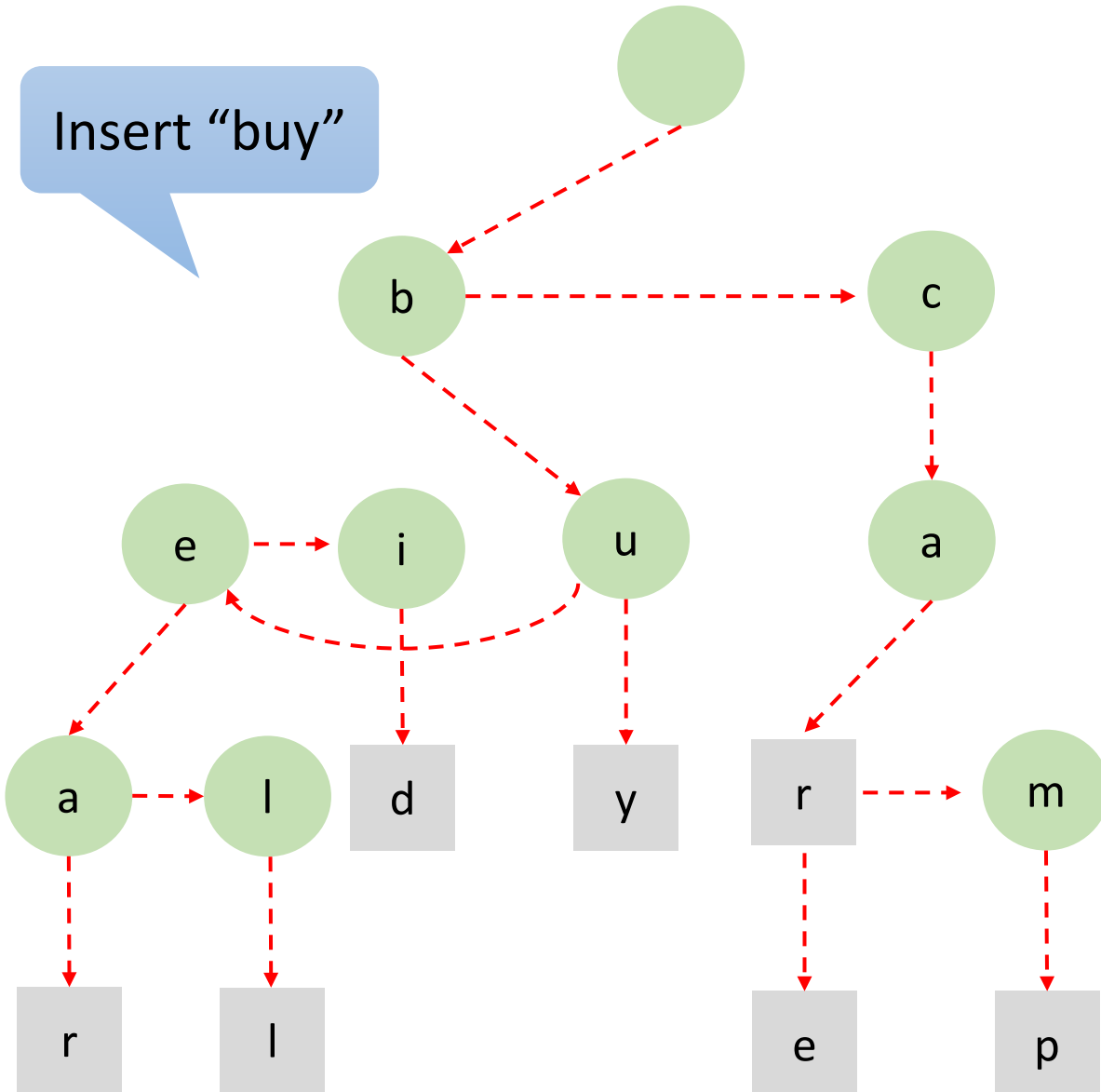


```
for each character in the word:
    if the character is a child node of
      the parent node:
        move to the next character
    else:
        new_node = create a new TrieNode
        #insert the child at the beginning
        #of the linked list
        set the new_node next be the
        parent_node's first child
        set the parent_node first child be
        the new_node
```

```
set end_of_word of the last_node as True
```

Insert a word

Insert "buy"



```
def _add_child(self, node, char):  
    new_node = TrieNode(char)  
    new_node.next = node.child  
    node.child = new_node  
    return new_node
```

```
def insert(self, word):  
    node = self.root  
    for char in word:  
        child = self._find_child(node, char)  
        if not child:  
            child = self._add_child(node, char)  
        node = child  
    node.is_end_of_word = True
```

Lab
Practice

Pre-order Depth First Traversal

- Pre-order
 - Process the current node's data
 - Visit the left child subtree
 - Visit the right child subtree

TreeTraversal(Node N):

Visit N;

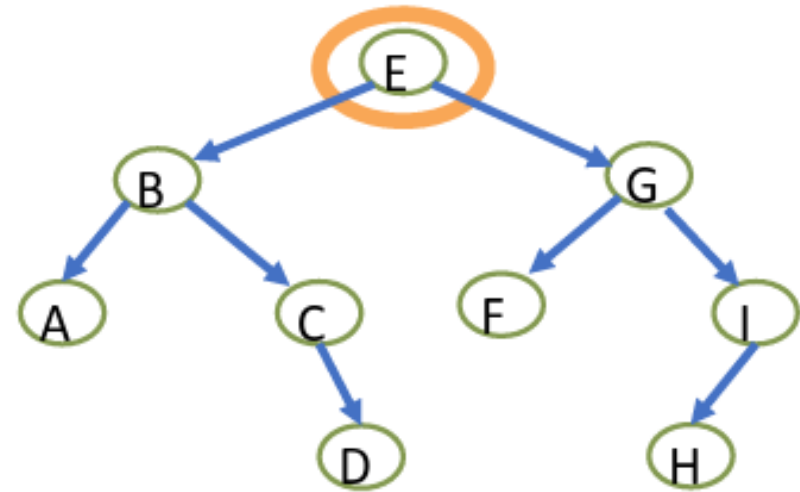
If (N has left child)

TreeTraversal(LeftChild);

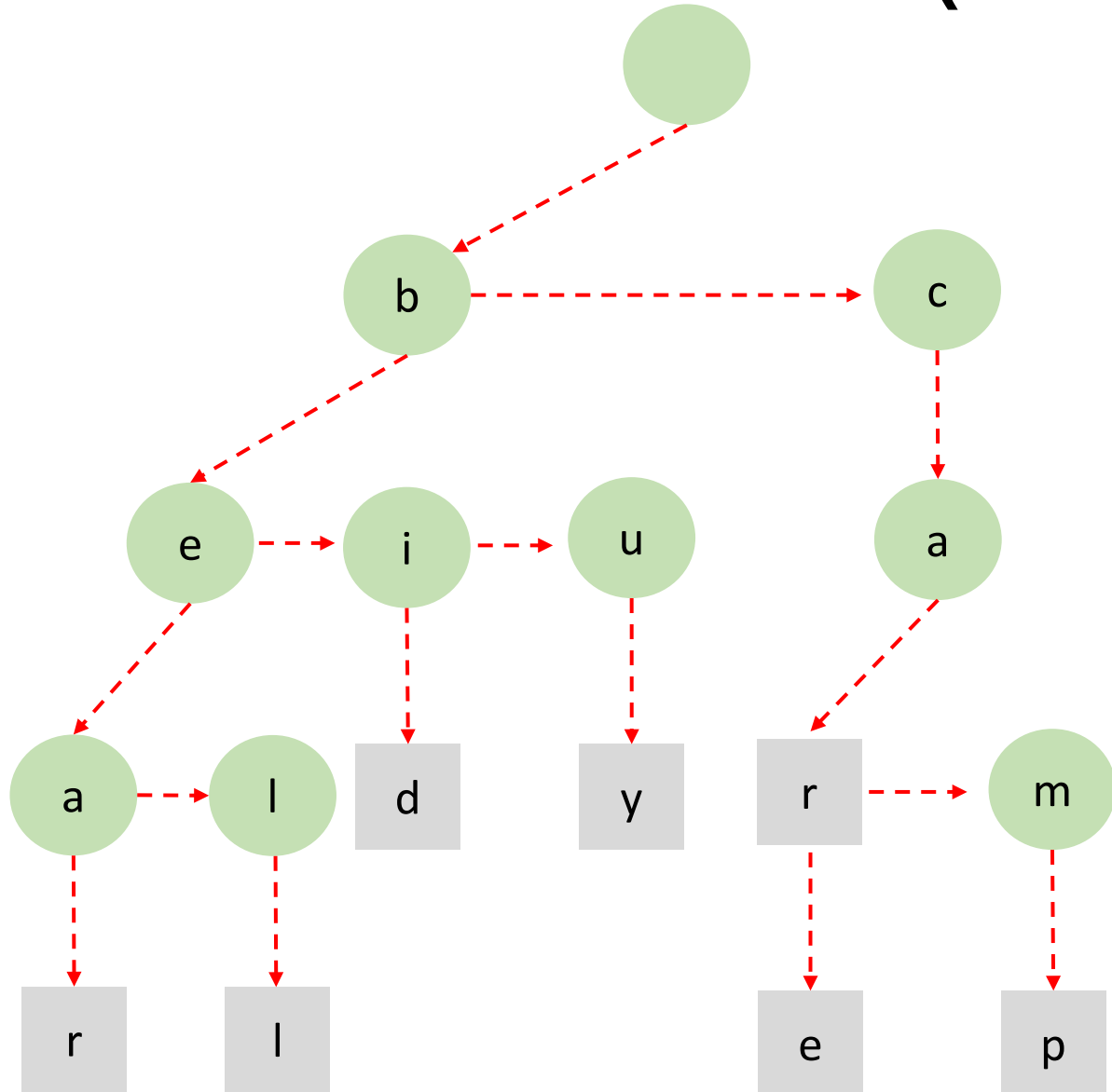
If (N has right child)

TreeTraversal(RightChild);

Return; // return to parent



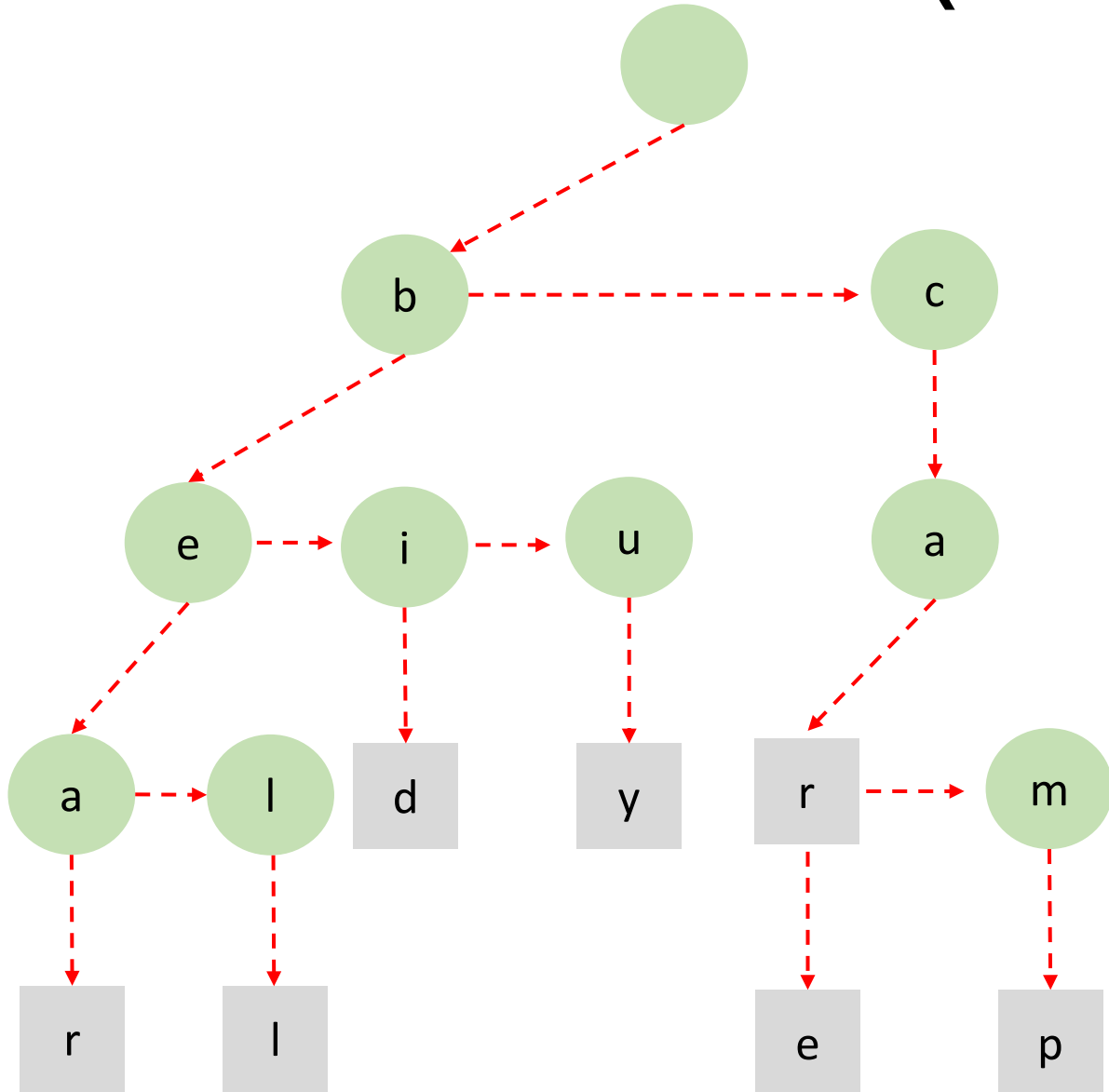
Preorder Traversal (DFS)



Instead of visiting left and right children,
visit each child of the TrieNode

```
dfs(TrieNode tn):  
    visit tn  
    child = tn.child  
    while child is not None:  
        dfs(child)  
        child = child.next
```

Preorder Traversal (DFS)



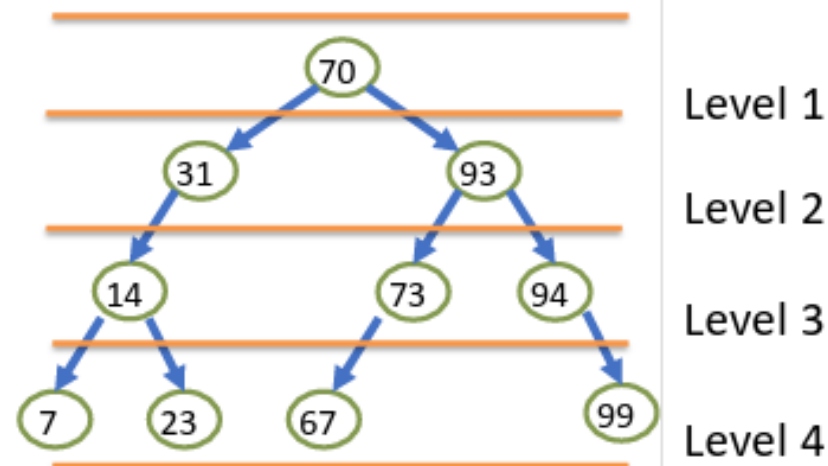
```
def dfs(self, node):  
    if node is not None:  
        print(node.char, end=" ")  
    child = node.child  
    while child:  
        self.dfs(child)  
        child = child.next
```

None b e a r l l d u y c a r e m p

Breadth-first Traversal: Level-by-level

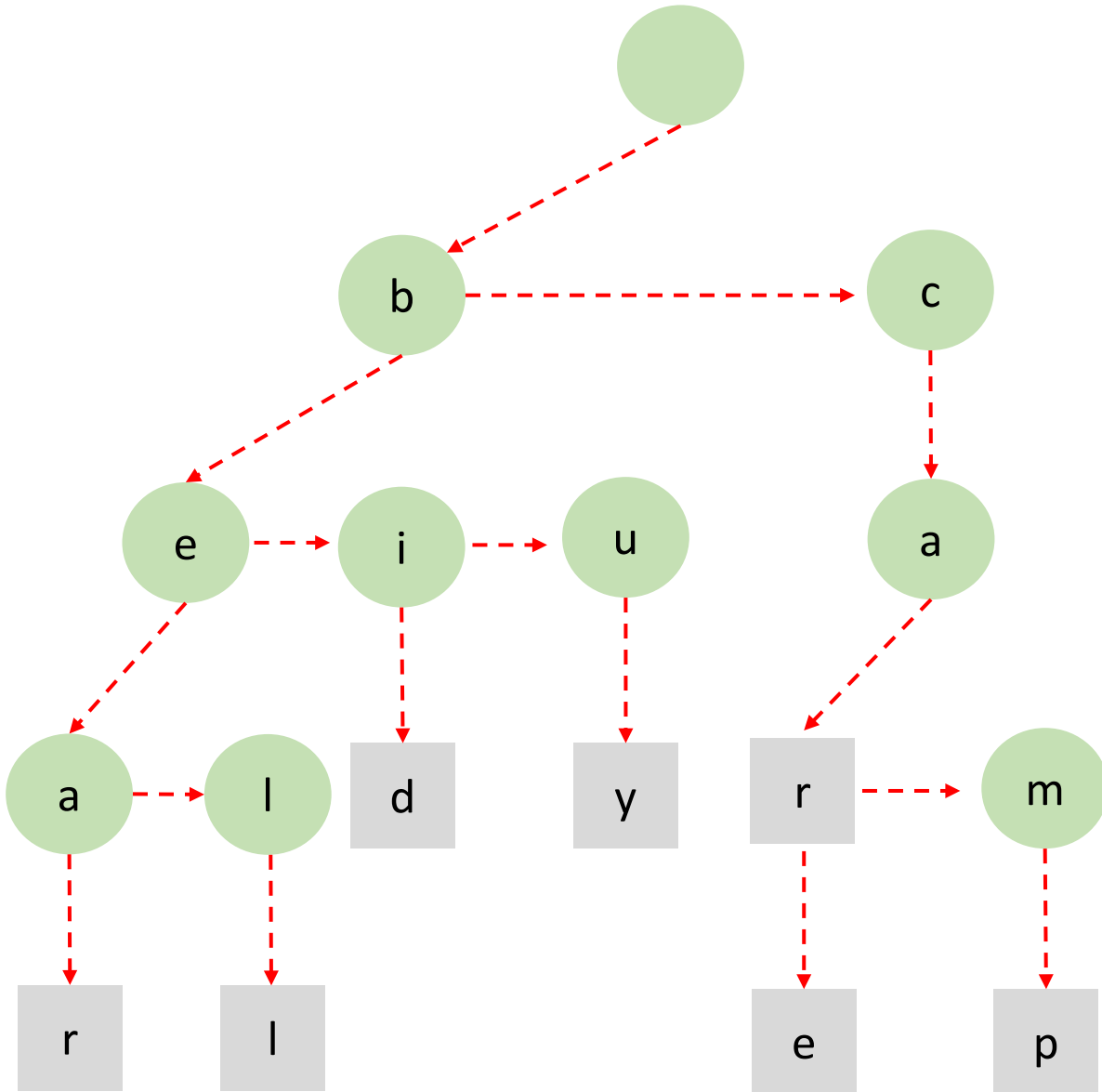
Level-By-Level Traversal:

- Visiting a node
- Remember all its children
 - Use a queue (FIFO structure)



1. Enqueue the current node
2. Dequeue a node
3. Enqueue its children if it is available
4. Repeat Step 2 until the queue is empty

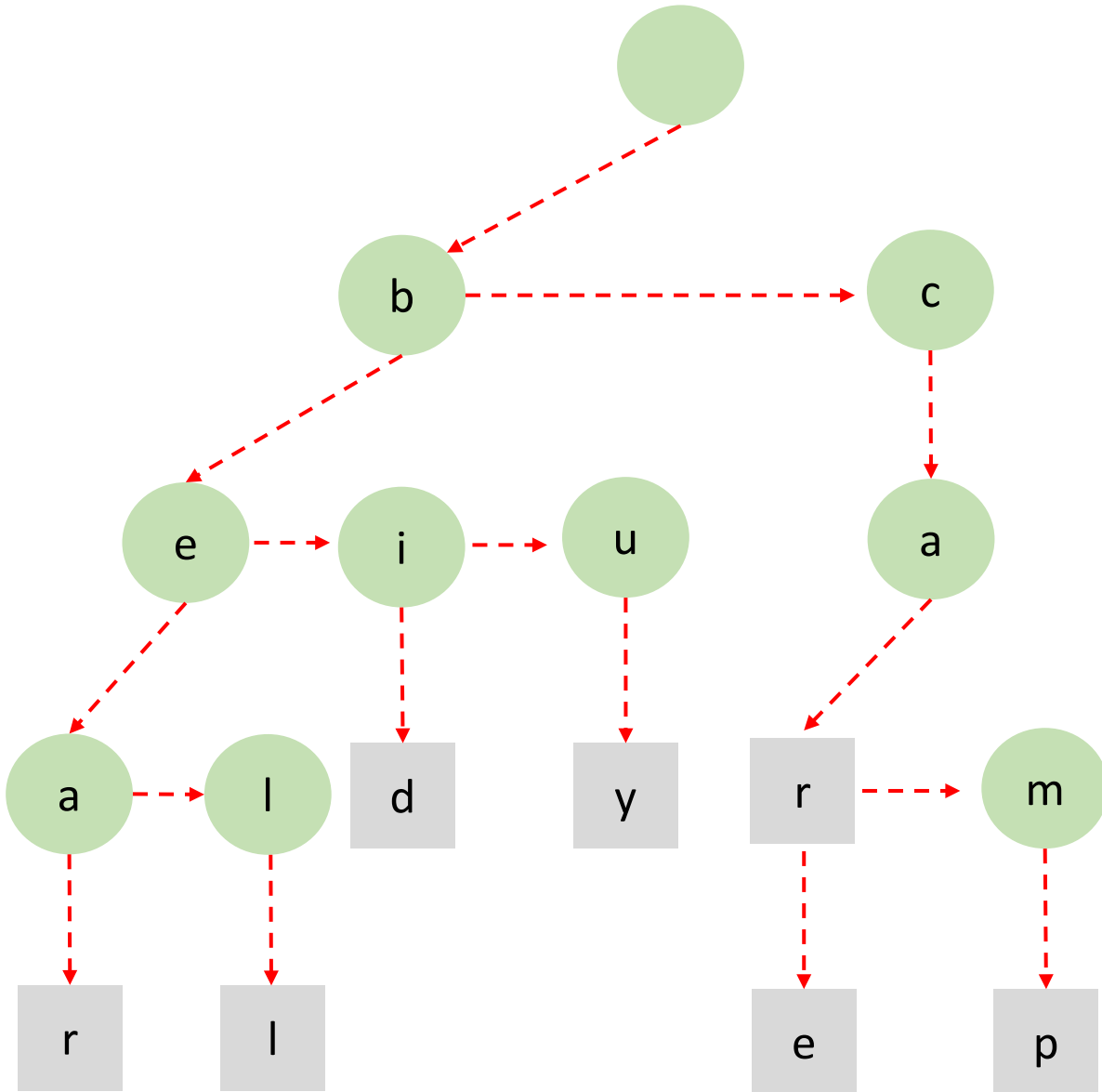
Level-by-Level Traversal (BFS)



```
def bfs(self):  
    queue = Queue()  
    queue.enqueue(self.root)  
    while not queue.is_empty():  
        node = queue.dequeue()  
        print(node.char, end=" ")  
        child = node.child  
        while child:  
            queue.enqueue(child)  
            child = child.next
```

None b c e i u a a l d y r m r l e p

Working Example: Print All Words

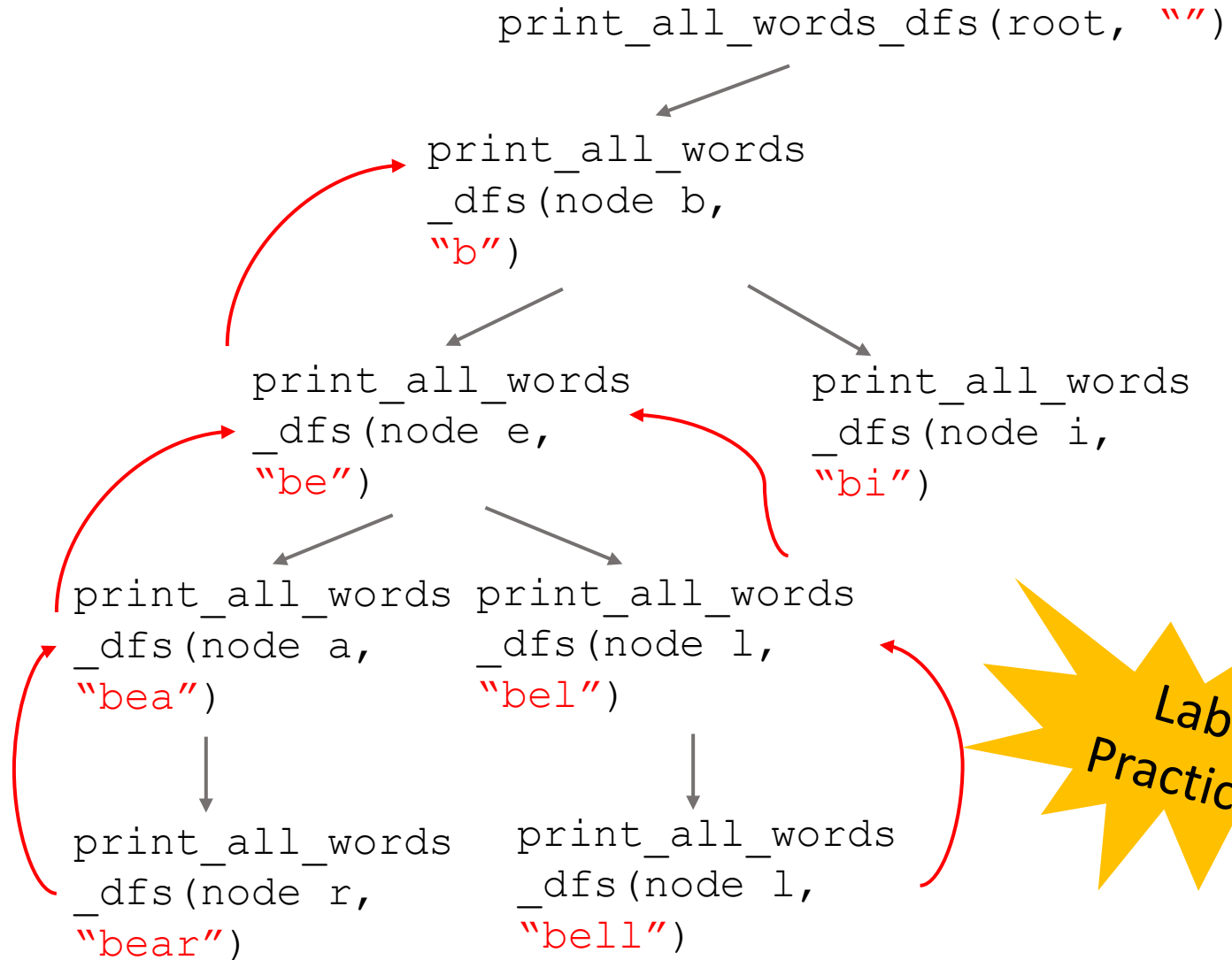
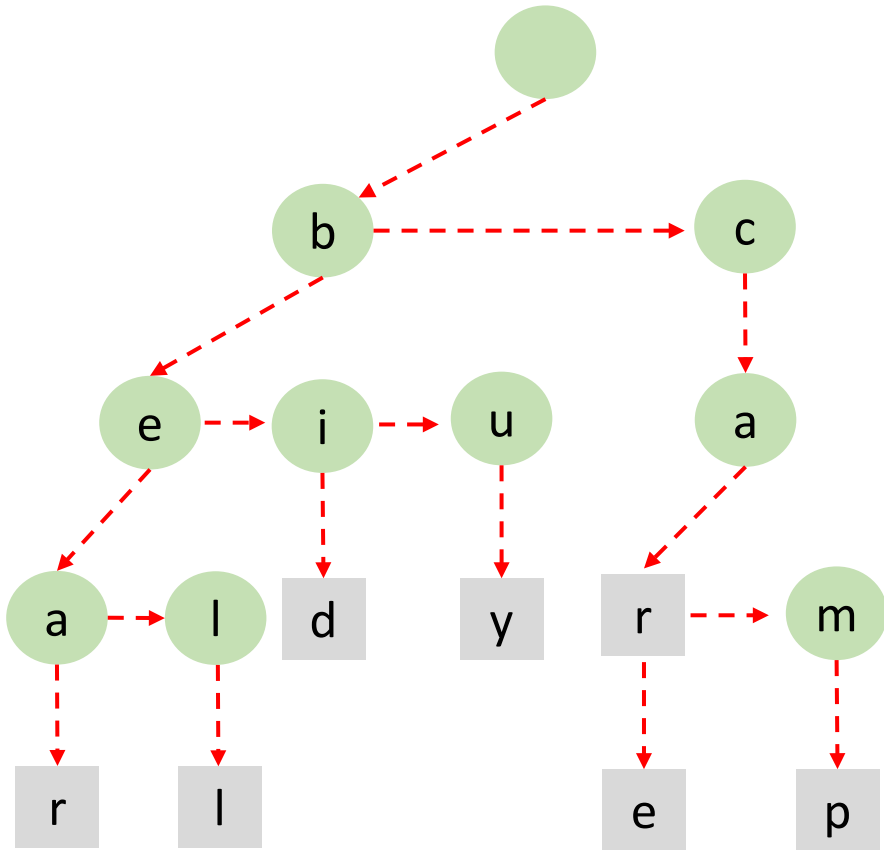


- Apply dfs
- When the node is the end of a word, print it
- Keep track of current nodes' ancestors

```
def print_all_words_dfs(self, node, prefix):
    if node.is_end_of_word:
        print(prefix)

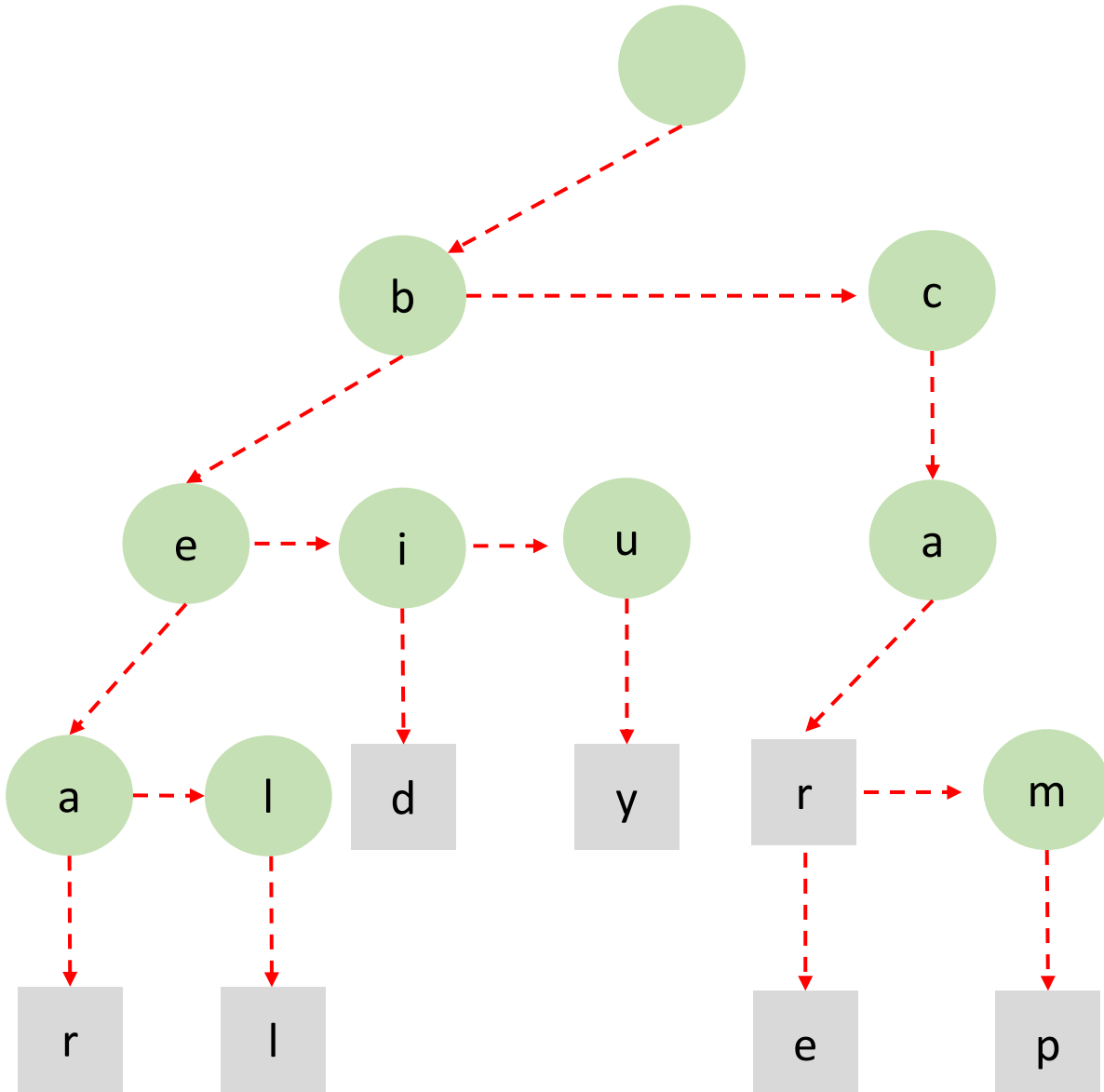
    child = node.child
    while child:
        self.print_all_words(child,
                              prefix+child.char)
        child = child.next
```

Working Example: Print All Words



Lab Practice

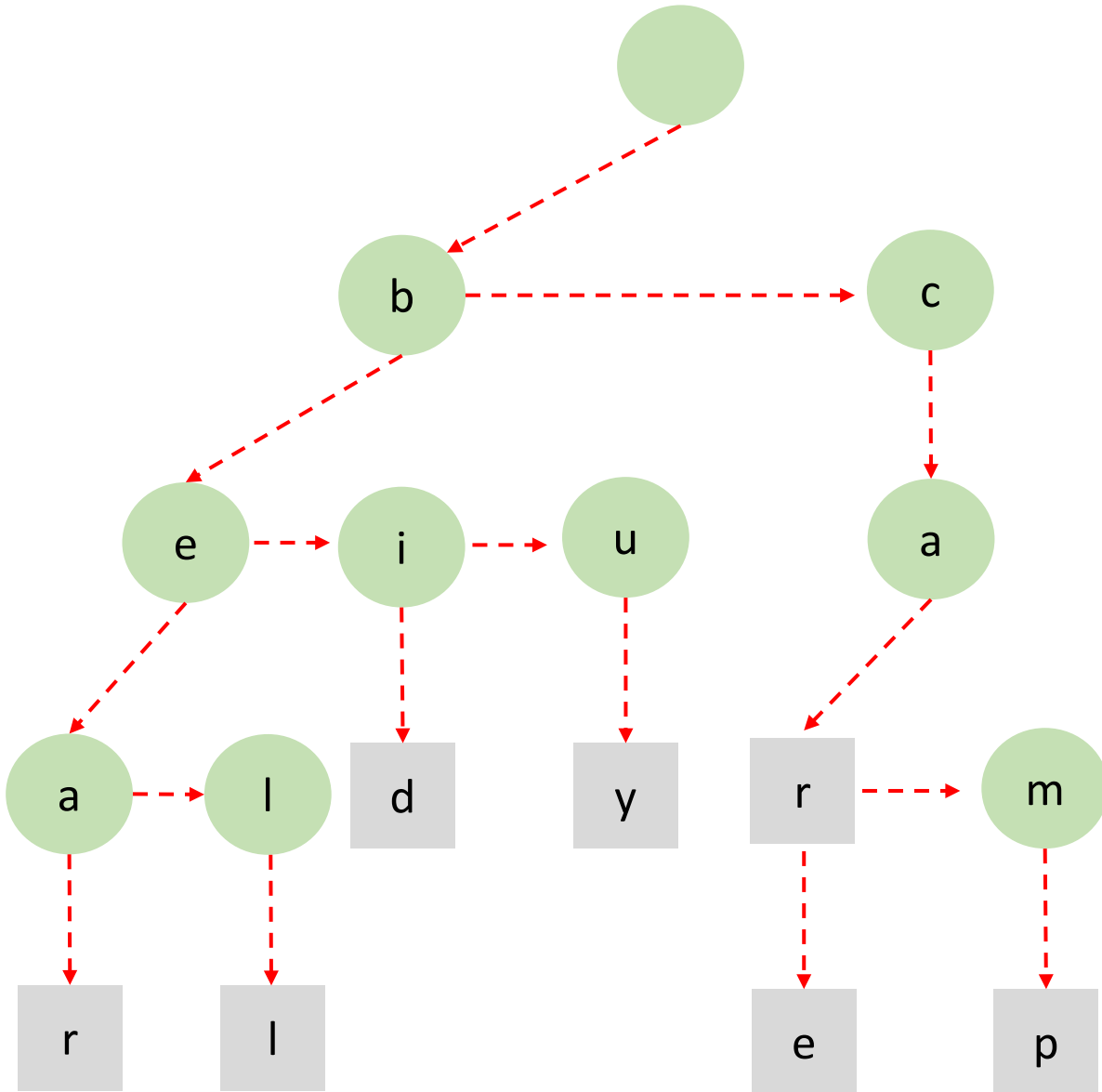
Working Example: Print All Words



- Apply bfs
- When enqueue a node, also enqueue the node's ancestors & the node's character
- When dequeue a node, if the node is end of a word, print the word

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        return None
    def is_empty(self):
        return len(self.items) == 0
```

Working Example: Print All Words

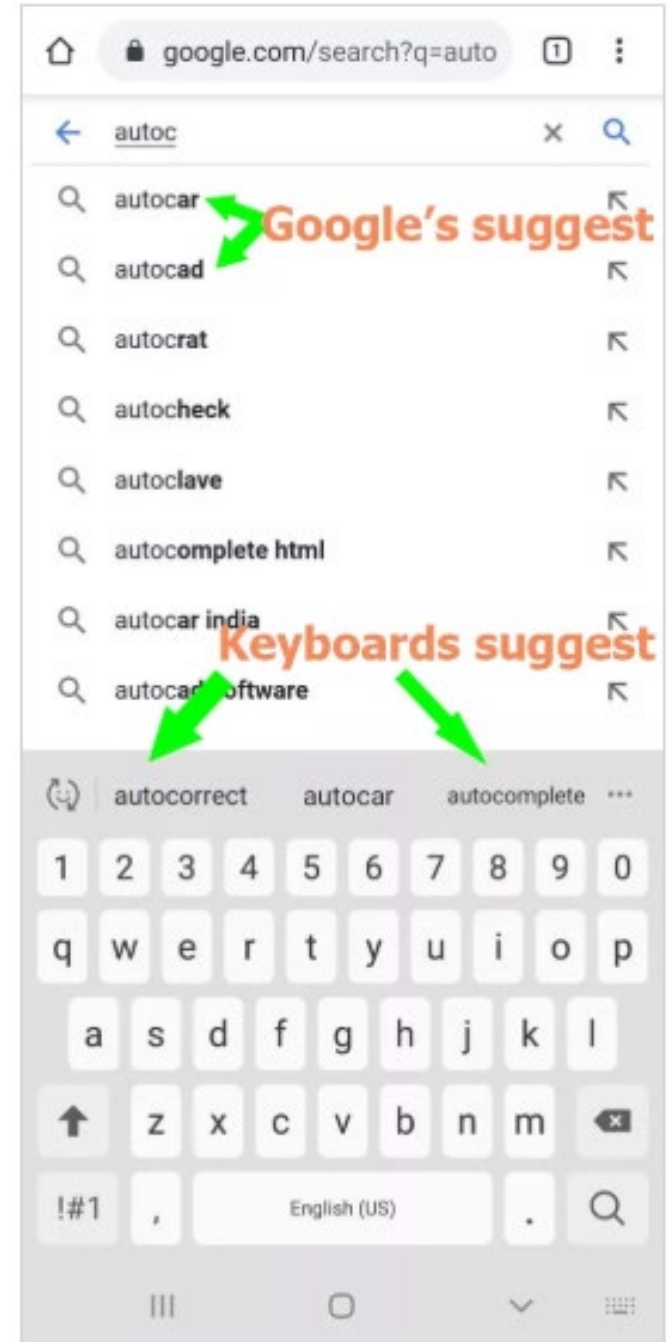


```
def print_all_words_bfs(self):  
    queue = Queue()  
    queue.enqueue((self.root, ""))  
    while not queue.is_empty():  
        node, prefix = queue.dequeue()  
        if node.is_end_of_word:  
            print(prefix)  
        child = node.child  
        while child:  
            queue.enqueue((child,  
                           prefix + child.char))  
            child = child.next
```

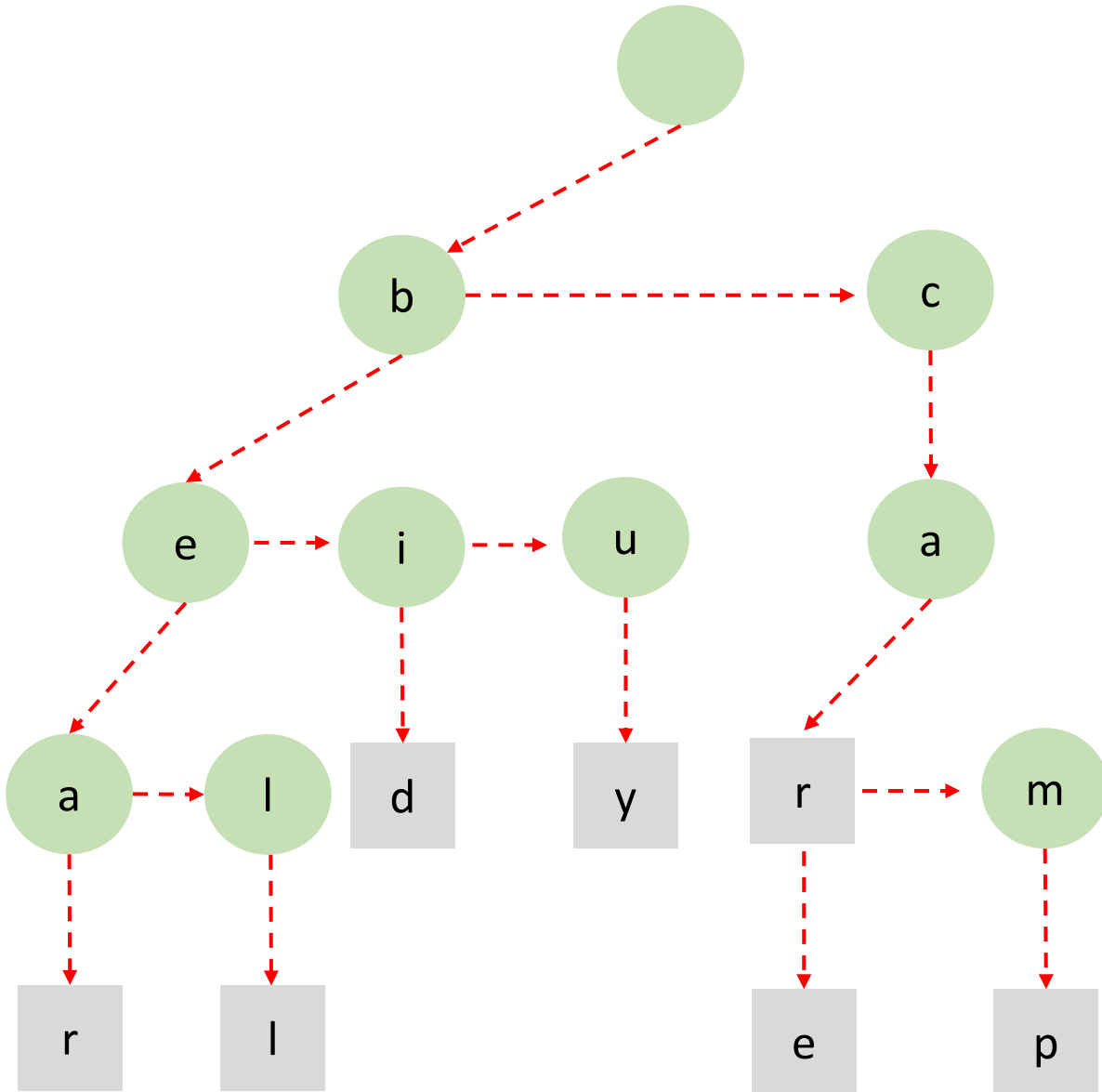
**Tutorial
Practice**

Application Example: Autocomplete

- Suggests possible words based on a given prefix
- Common in search bars, text editors, messaging apps
- Needs fast prefix lookup for responsiveness as you type.
- Use a trie
 - Efficient prefix-based search
 - Stores multiple words compactly using shared prefixes




Application Example: Autocomplete



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

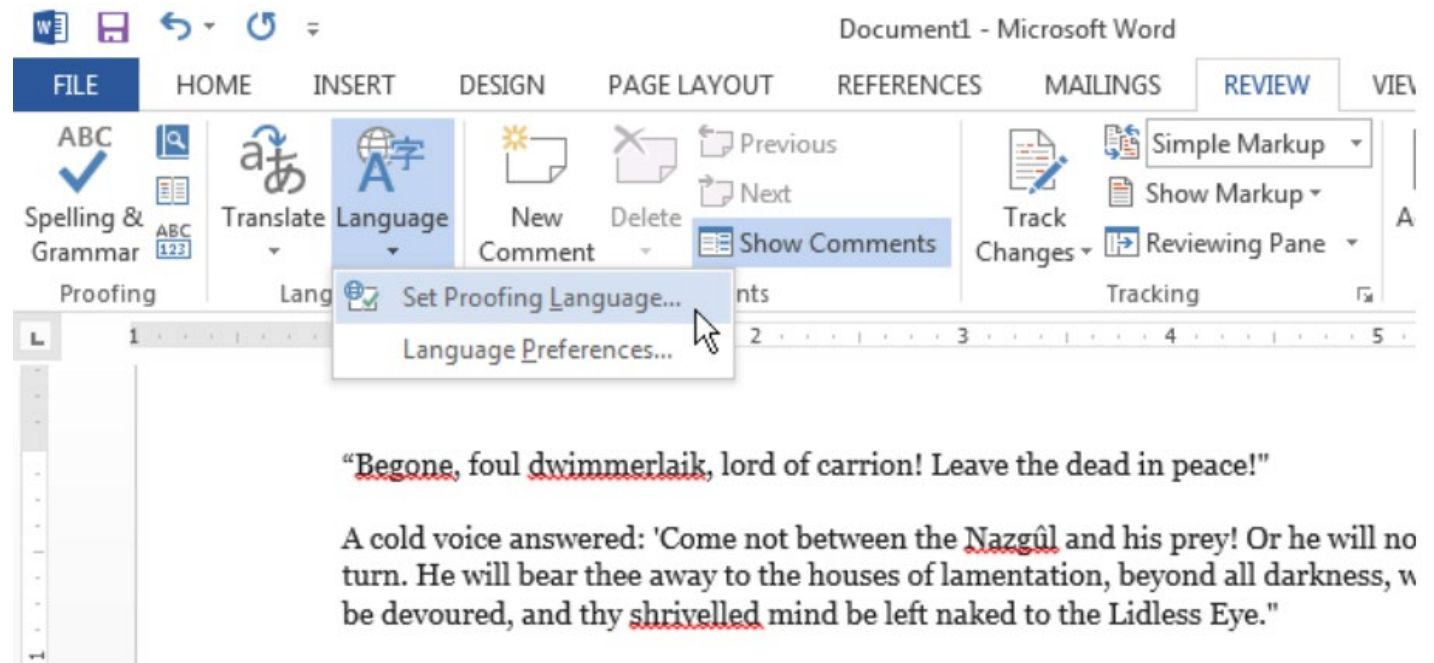
rules



Tutorial
Practice

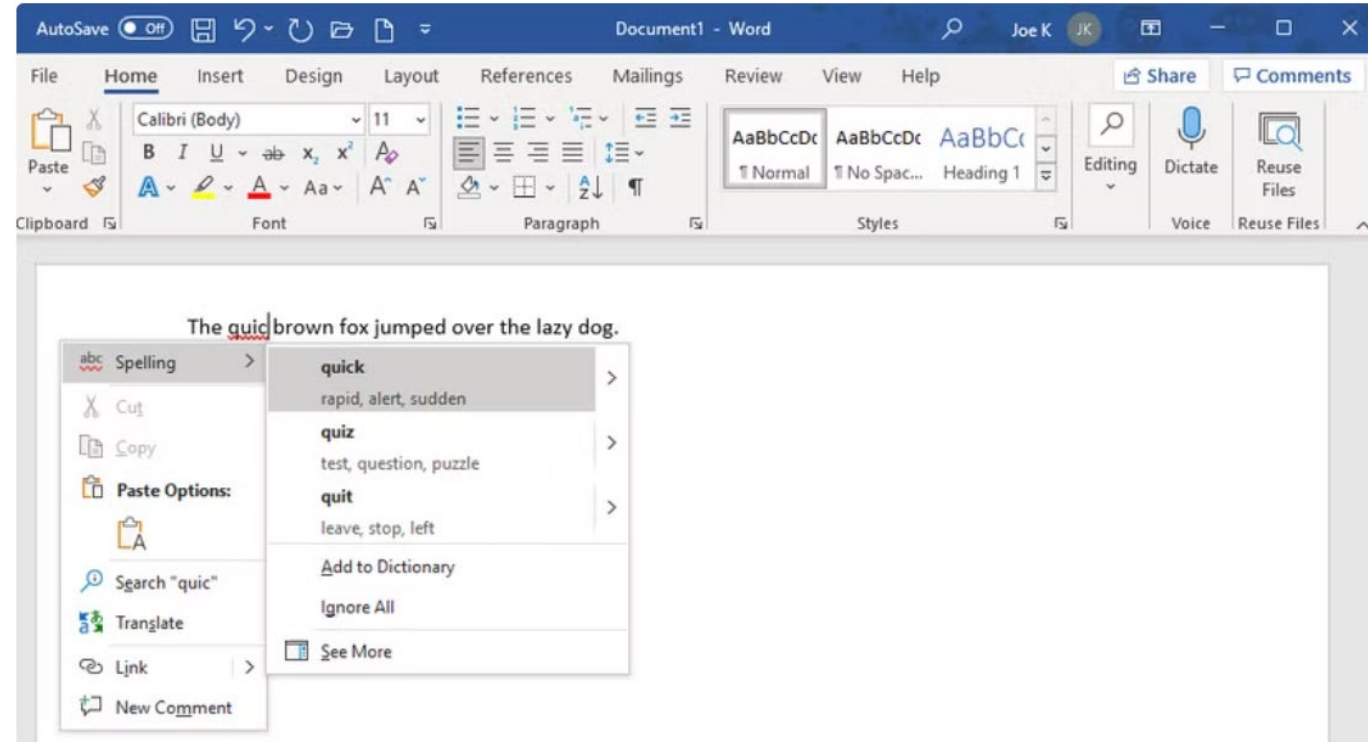
Application Example: Spell Checking

- Check if a word is valid
- Suggest corrections for misspelled words
- It is common in:
 - Word processors
 - Messaging apps
 - Search engines



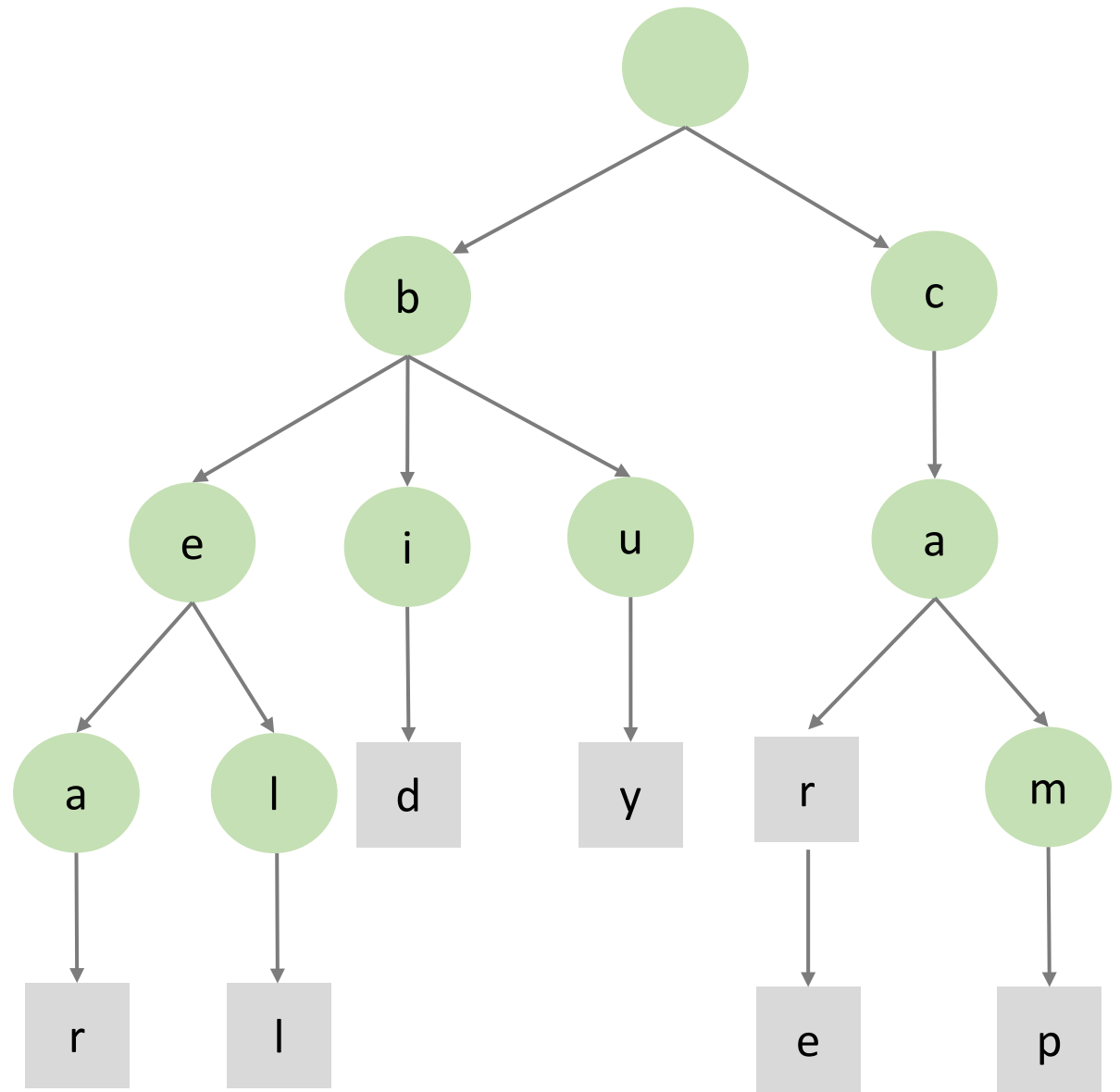
Application Example: Spell Checking

- Check if a word is valid
 - Search a word in the trie
- Suggest corrections for misspelled words
 - Prefix
 - Edit distance, e.g., Levenshtein distance
 - Frequency ranking
 - User history/context
 -



Summary

- A tree-based data structure used for efficient string operations.
- Implementations with linked list
 - Insert a word
 - Search a word
 - Traversal of a trie: dfs and bfs
- Examples
 - Print all words
 - Autocomplete
 - Spell checking



The trie structure for strings: bear, bell, bid, buy, car, care, camp