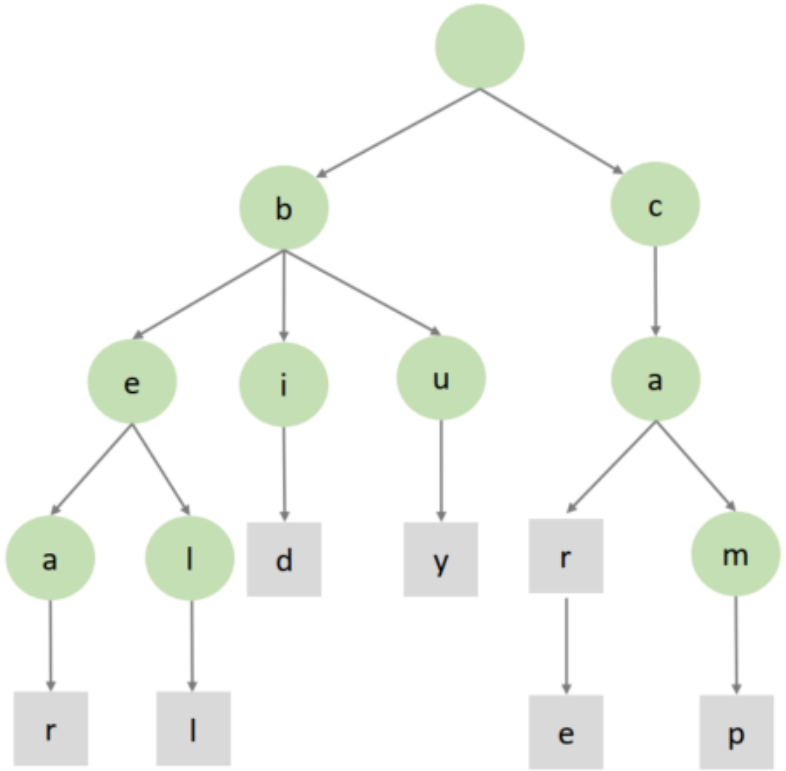
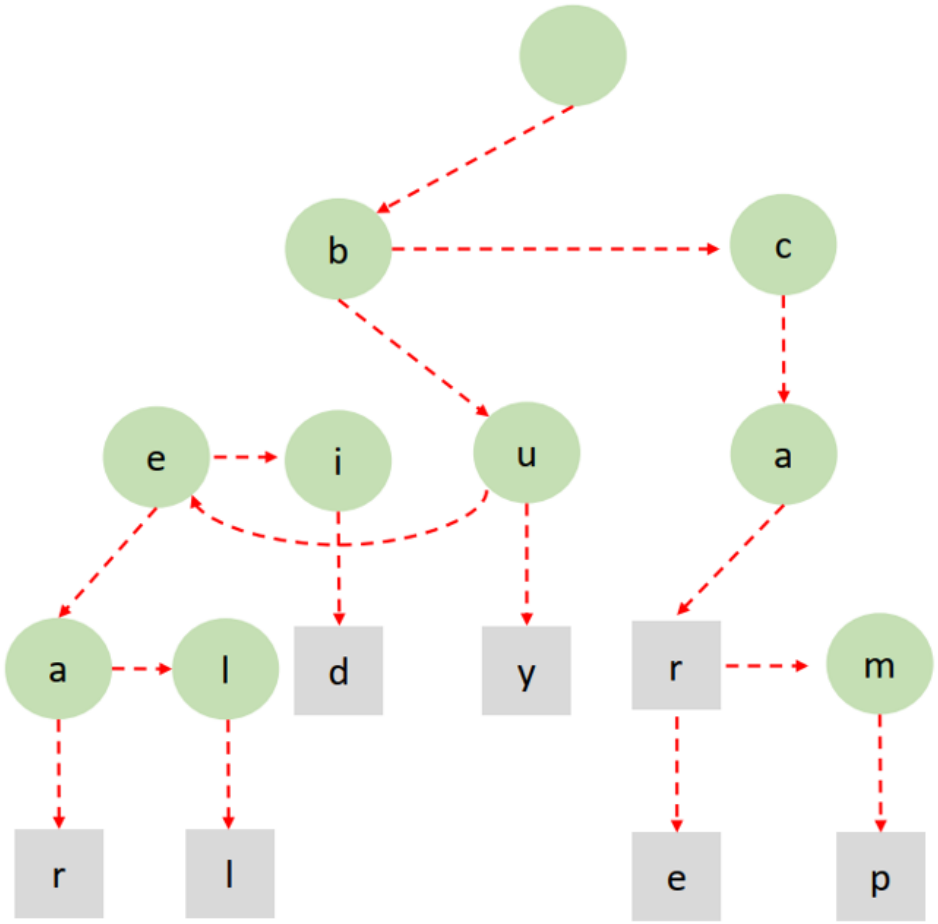
Trie

A **tree-based** data structure used for efficient **string operations**. Also called prefix tree or digital tree.



- The structure represents the prefix relationships between the keys
- Each node can be the end of a word or the prefix of another word
- The path from the root to a node represents a prefix of some string(s) in the collection.
- Multiple Children
- Can store duplicate keys

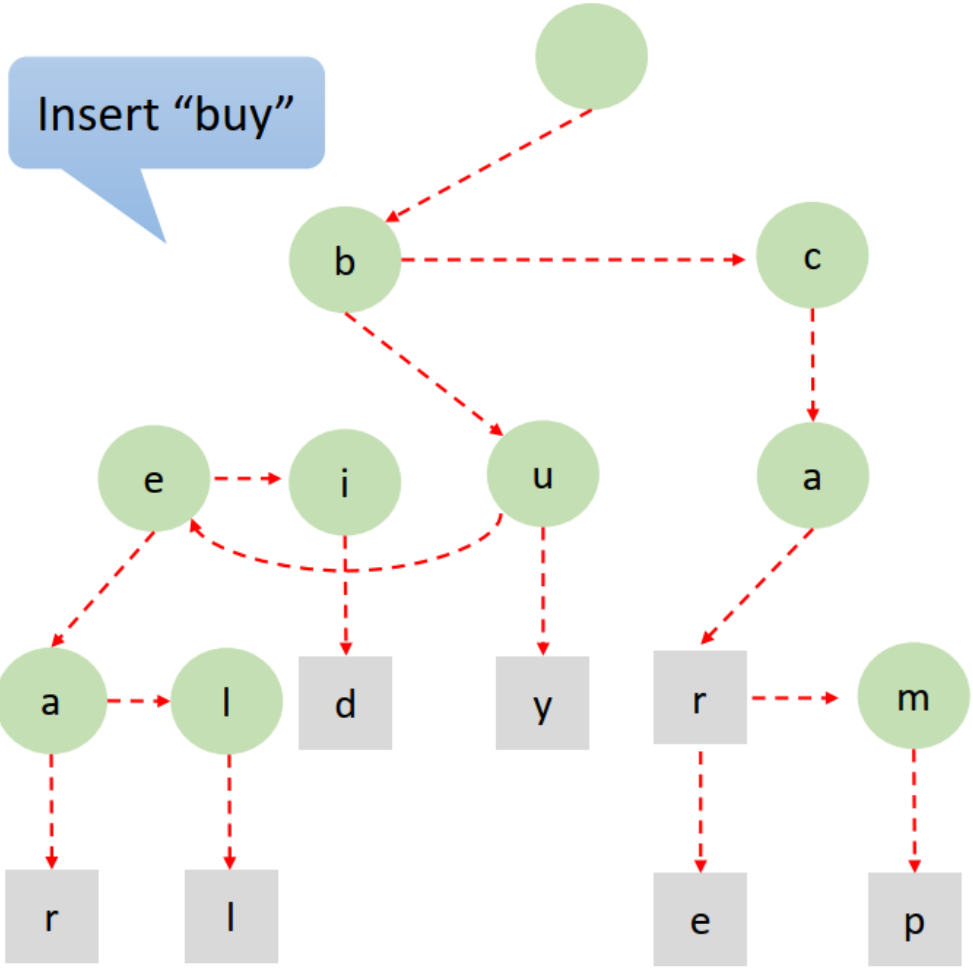
Trie




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

Q1 Insert a Word

Modify the _add_child() method in the Trie class so that it always inserts characters in an **alphabetical order**

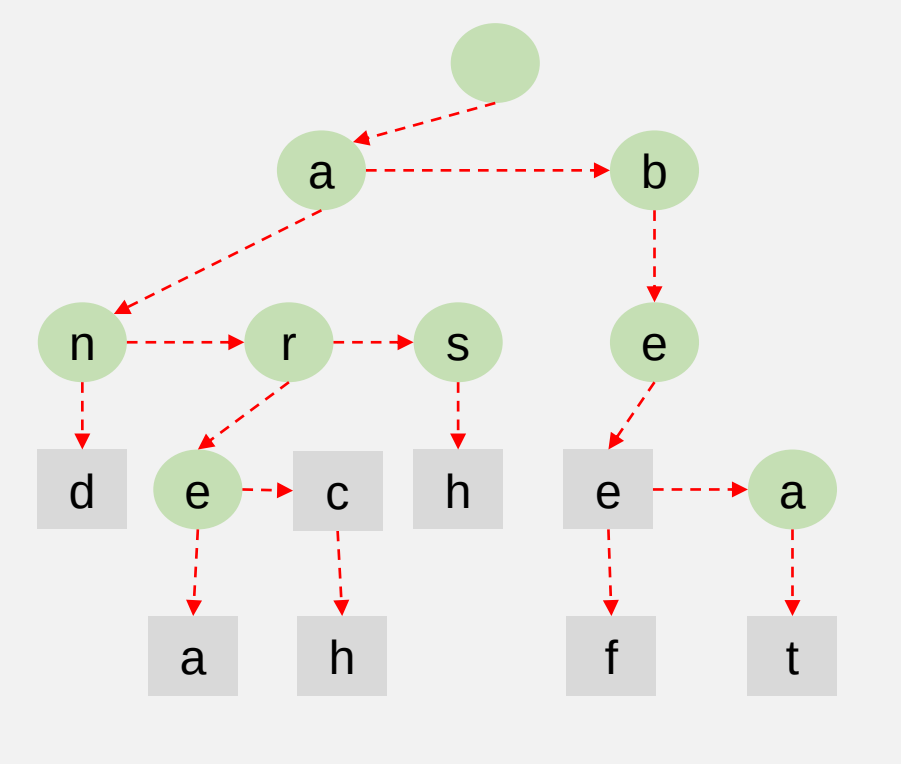



```
def _add_child(self, node, char):
     new_node = TrieNode(char)
     new_node.next = node.child
     node.child = new_node
     return new_node

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
```

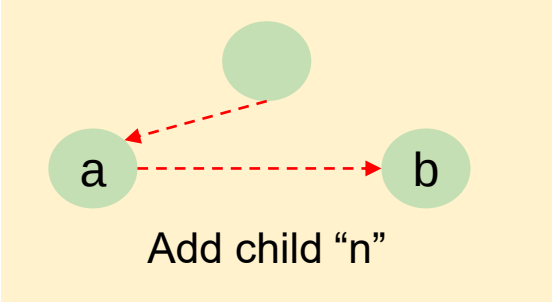
Lab Practice

Q1 Insert a Word

Modify the _add_child() method in the Trie class so that it always inserts characters in an **alphabetical order**

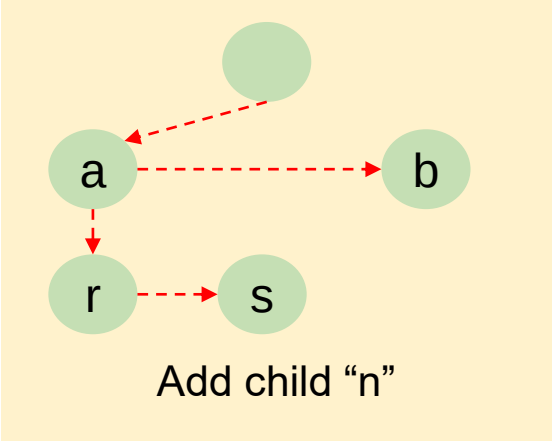


Case 1: No child



- Create a new_node
- node.first_child = new_node

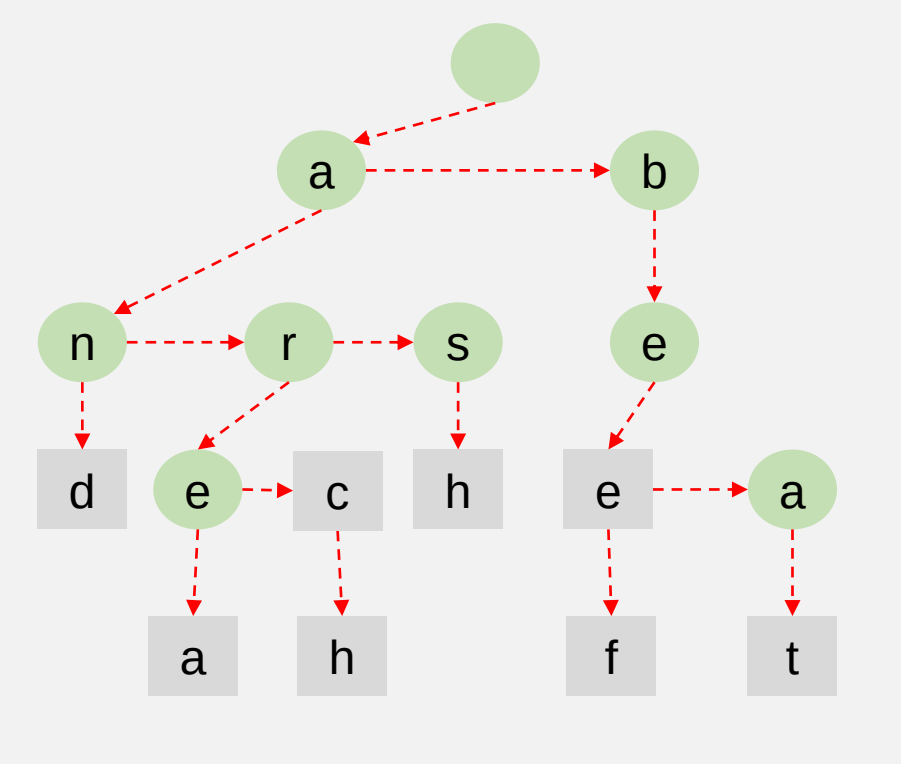
Case 2: Has children, but the new_node should be added as the new “first_child”



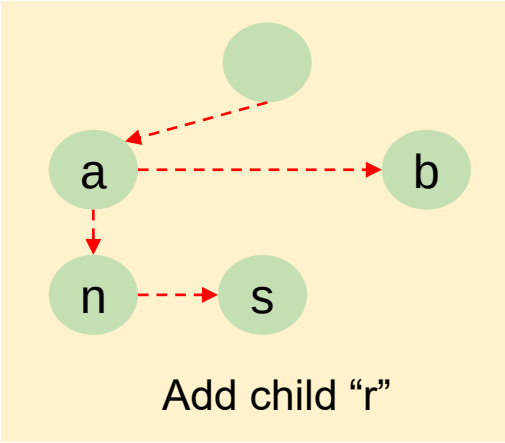
- Create a new_node
- new_node.next_sibling = node.first_child
- node.first_child = new_node

Q1 Insert a Word

Modify the _add_child() method in the Trie class so that it always inserts characters in an **alphabetical order**



Case 3: Has children, but the new_node should be added to the middle



- Create a new_node
- Locate the first child and start comparing
- Find the position and reconnect siblings

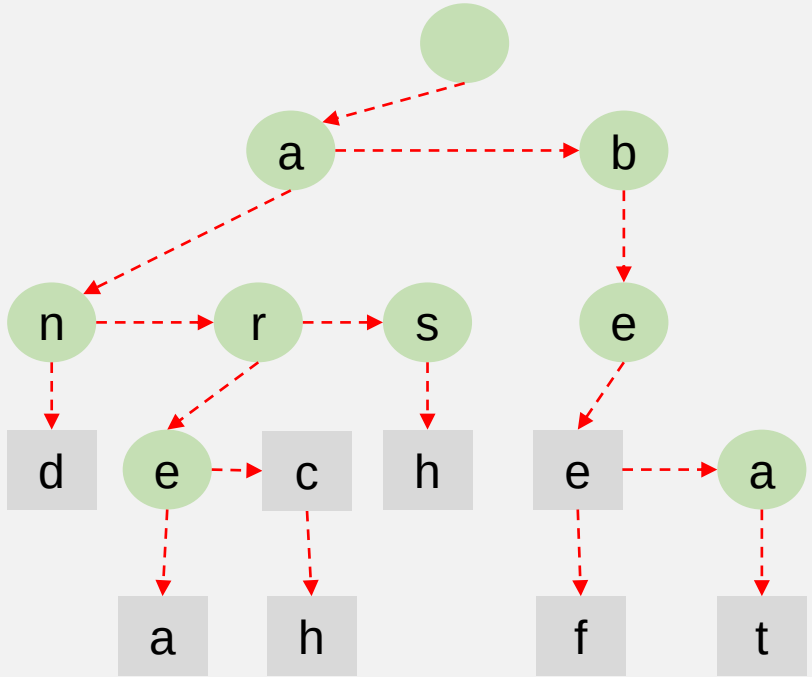
Q2 Print Words in a Lexicographical Order

With the modified _add_child() method, implement a method to print all words stored in a trie in a **lexicographical order.**

def print_words_alphabetically(self):

Insert: "banana", "apple", "bat", "ball", "band", "cat"

The results to print words in a lexicographical order: apple, ball, band, banana, bat, cat



Print out: “and”, “area”, “arc”, “arch”, “ash”, “bee”, “beef”, “beat”

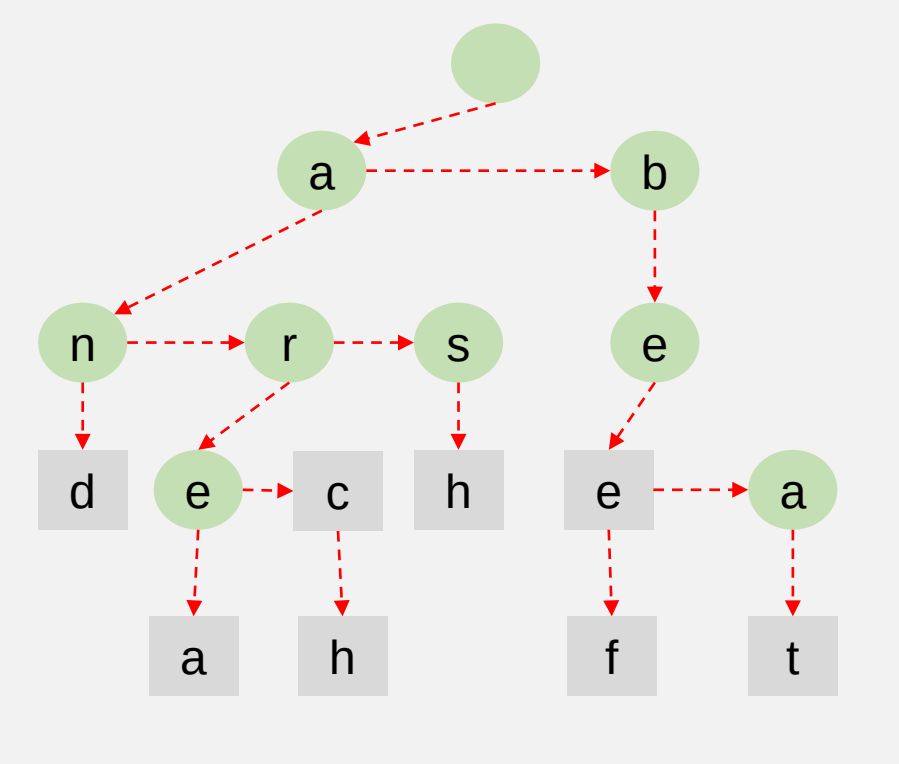
```
def dfs(self, node):
    if node is not None:
        print(node.char, end=" ")

    child = node.child
    while child:
        self.dfs(child)
        child = child.next
```

1. Add one more parameter
2. Print out the words if the letter is the end

Q2 Print Words in a Lexicographical Order

With the modified _add_child() method, implement a method to print all words stored in a trie in a **lexicographical order.**

def print_words_alphabetically(self):



Print out: “and”, “area”, “arc”, “arch”, “ash”, “bee”, “beef”, “beat”

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

    child = node.child
    while child:
        self.dfs(child)
        child = child.next
```

```
def dfs(self, node, path):
    if node.is_end_of_word:
        print(path)

    child = node.first_child

    while child:
        self.dfs(child, path + child.char)
        child = child.next_sibling
```

1. Add one more parameter
2. Print out the words if the letter is the end

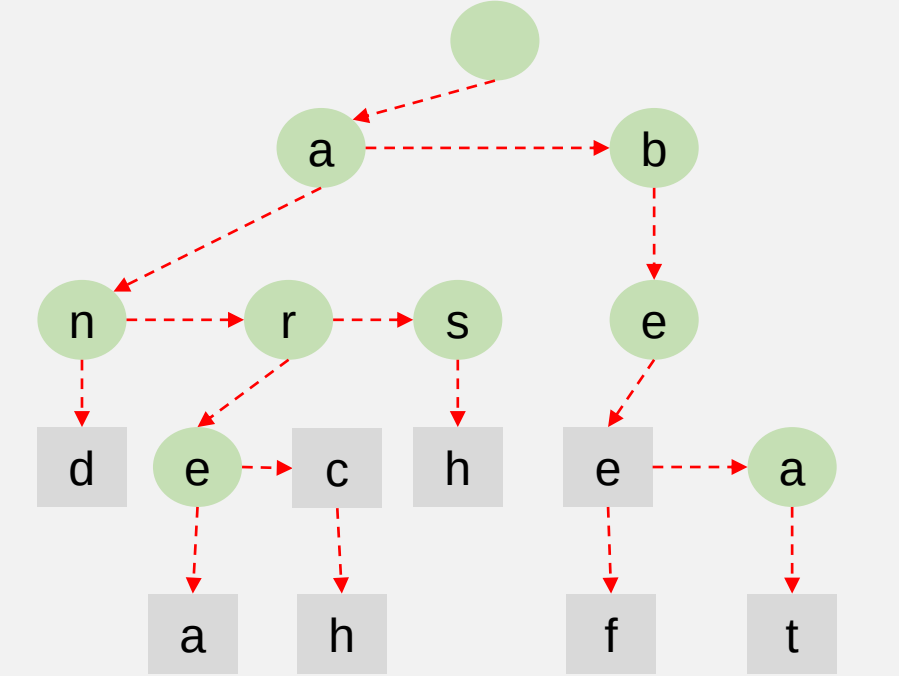
Q3 Print Words in a Reversed Lexicographical Order

Implement a method to print all words stored in a trie in a **reverse lexicographical order**.

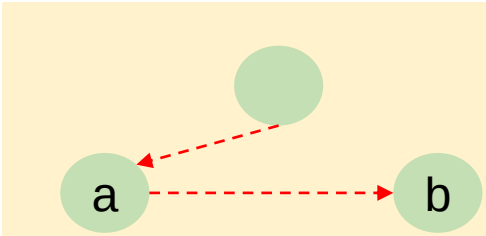
def print_words_reverse_alphabetically(self):

Insert: "banana", "apple", "bat", "ball", "band", "cat"

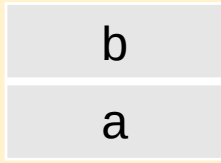
The results to print words in a revers lexicographical order: cat, bat, banana, band, ball, apple

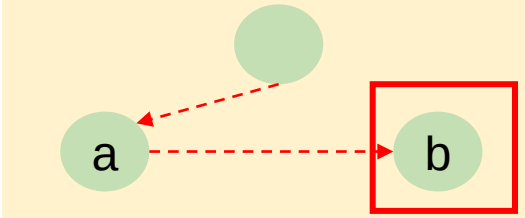


Print out: “beat”, “beef”, “bee”, “ash”, “arch”, “arc”, “area”, “and”



Push siblings into a stack





Pop nodes from the stack and conduct a DFS search