

2. Trie Data Structure Notes

#tries #datastructures #tree

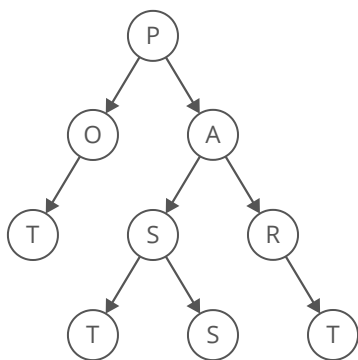
Introduction

A Trie is a tree-like data structure optimized for storing and searching strings or sequences of characters. It's excellent for prefix matching, autocomplete, and efficient word lookups. Tries are commonly used in search engines, spell-checkers, and any application involving text processing. They represent characters hierarchically, making them ideal for finding words with common prefixes.

Let's learn how Trie Data Structures work.

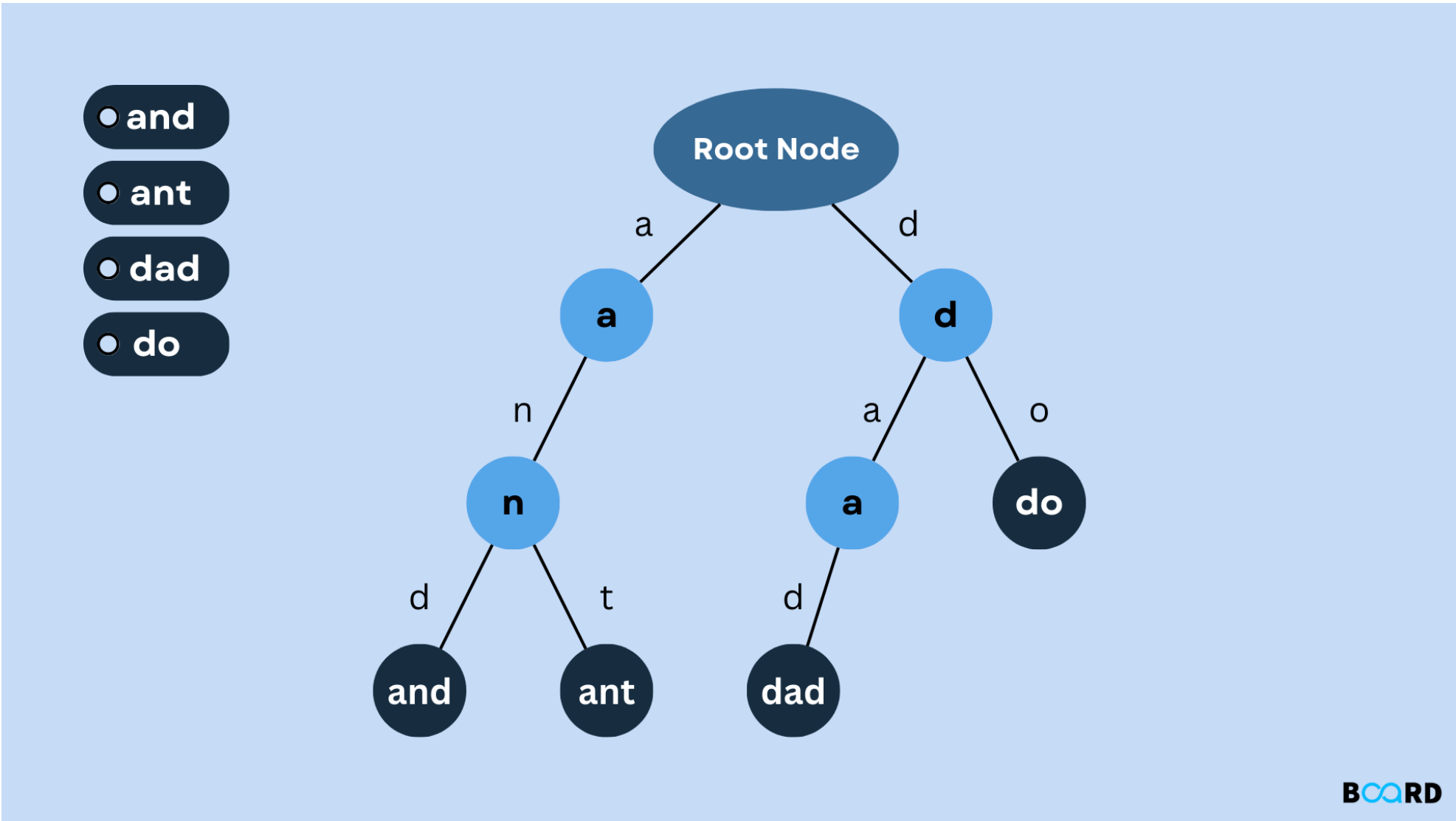
How words are stored in Trie

Example 1:



As we can see how the words "Pot" , "Past", "Pass", and "Part" are stored in a single Tree.

Example 2:



Characteristics of Trie DS

It's a tree like data structure, that stores data in a way that -

1. Words can be inserted character by character
2. Words can be inserted under same prefix
3. Search words easily (Similarity Suggestions etc.)

4. Prefix can be searched and identified as words.
5. Every node has two pointers - children -> str and isWord -> bool

Basically it's an efficient way to store data, because it can save us a lot of space.

Problems

Problem 1: Leetcode - [Implement Trie Prefix Tree](#)

Medium

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

class Trie:
    def __init__(self):
        self.root = TrieNode() #The root is an instance of the TrieNode() class

    #insert
    def insert(self, word:str) -> None:
        #starting from the root
        current = self.root #current is set to the root instance
        #for every character in the given word
        for c in word:
            #check if that character exists in the children
            if c not in current.children: # character doesn't exist
                current.children[c] = TrieNode() #add that character to node
            #character exists
            current = current.children[c] # skip it and move to the next child
        #after all of them are added, mark the end of the word as True
        current.endOfWord = True

    #search
    def search(self, word:str) -> bool:
        # starting from the root
        current = self.root #current is set to the root instance
        # going through every character of the word
        for c in word:
            # check if character exists in the children
            if c not in current.children:
                return False # the character doesn't exist
            #character exists
            current = current.children[c] # skip it and move to the next child
        # if it reaches here word exists and that means it's True anyway.
        return current.endOfWord

    def startsWith(self, prefix: str) -> bool:
        #start from the root
        current = self.root
        #for every character in the prefix
        for c in prefix:
            #if character is in children or not
            if c not in current.children:
                return False # not a prefix
            #move to the next child
            current = current.children[c]
        #if that for loop ends without returning that means we've checked all the characters
        return True

# Your Trie object will be instantiated and called as such:

obj=Trie()
param1 = ["Trie","insert","search","search","startsWith","insert","search"]
param2 = [[],["apple"],["apple"],["app"],["app"],["app"],["app"]]
output = []
operations={
    "Trie":None,
```

```

    "insert": obj.insert,
    "search": obj.search,
    "startsWith": obj.startsWith
}

for i in range(len(param1)):
    method = operations[param1[i]]
    if method is not None:
        output.append(method(param2[i][0]))
    else:
        output.append(None)
print(output)

```

```

Operation 1: "Trie" Word: None
Tree: {}
return None
output: [None]

Operation 2: "insert" Word: "apple"
Tree: {
  "root": {
    "children": {
      "a": {
        "children": {
          "p": {
            "children": {
              "l": {
                "children": {
                  "e": {
                    "children": {},
                    "endOfWord": true
                  }
                }
              },
              "endOfWord": false
            }
          },
          "endOfWord": false
        }
      },
      "endOfWord": false
    }
  },
  "endOfWord": false
}
return None
output: [None, None]

Operation 3: "search" Word: "apple"
return True
output: [None, None, True]

Operation 4: "search" Word: "app"
return False
ouput: [None, None, True, False]

Operation 5: "startsWith" Word: "app"
return True
output: [None, None, True, False, True]

Operation 6: "insert" Word: "app"
return None
output: [None, None, True, False, True, None]
Tree:{
  "root": {
    "children": {

```

```

        "a": {
            "children": {
                "p": {
                    "children": {
                        "p": {
                            "children": {
                                "l": {
                                    "children": {
                                        "e": {
                                            "children": {},
                                            "endOfWord": true
                                        }
                                    }
                                },
                                "endOfWord": false
                            }
                        },
                        "endOfWord": True
                    }
                },
                "endOfWord": false
            }
        },
        "endOfWord": false
    }
},
"endOfWord": false
}
}

```

Operation 7: "search" Word: "app"
 return True
 output: [None, None, True, False, True, None, True]

Alternate Solution: Using dictionary

```

class Trie(object):
    def __init__(self):
        self.data = {"children": {}, "endOfWord": False}
    def insert(self, word):
        current_node = self.data
        for char in word:
            if char not in current_node["children"]:
                current_node["children"][char] = {"children": {}, "endOfWord": False}
            current_node = current_node["children"][char]
        current_node["endOfWord"] = True

    def search(self, word):
        current = self.data
        for c in word:
            if c not in current['children']:
                return False
            current = current['children'][c]
        return current['endOfWord']

    def startsWith(self, prefix):
        #start from the root
        current = self.data
        #for every character in the prefix
        for c in prefix:
            #if character is in children or not
            if c not in current["children"]:
                return False # not a prefix
            #move to the next child
            current = current["children"][c]
        #if that for loop ends without returning that means we've checked all the characters
        return True

obj = Trie()
param1 = ["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
param2 = [[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
output = []
operations={
    "Trie": None,

```

```

        "insert": obj.insert,
        "search": obj.search,
        "startsWith": obj.startsWith
    }
    for i in range(len(param1)):
        method = operations[param1[i]]
        if method is not None:
            output.append(method(param2[i][0]))
        else:
            output.append(None)
    print(output)

```

Problem: 2 Leetcode [Design Add And Search Words Data Structure](#)

Medium

```

class TrieNode():
    def __init__(self):
        self.children = {}
        self.isWord = False
class WordDictionary(object):
    def __init__(self):
        self.root = TrieNode()

    def addWord(self, word):
        current = self.root
        for c in word:
            if c not in current.children:
                current.children[c] = TrieNode()
            current = current.children[c]
        current.isWord = True

    def search(self, word):

        def dfs(i, root):
            current = root
            for i in range(i, len(word)):
                if word[i] == ".":
                    for child in current.children.values():
                        if dfs(i+1, child):
                            return True
                if word[i] not in current.children:
                    return False
                current=current.children[word[i]]
            return current.isWord

        return dfs(0, self.root)

obj = WordDictionary()
param1=["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
param2=[[],"bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
output = []
operations={

        "WordDictionary":None,
        "addWord": obj.addWord,
        "search": obj.search
    }
    for i in range(len(param1)):
        method = operations[param1[i]]
        if method is not None:
            output.append(method(param2[i][0]))
        else:
            output.append(None)
    print(output)

```

```
>>> [None, None, None, None, False, True, True, True]
```

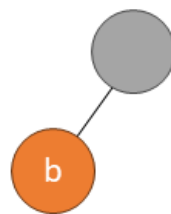
Explanation

“WordDictionary”



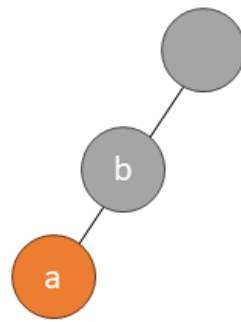
Output: [None]

addWord(“bad”)



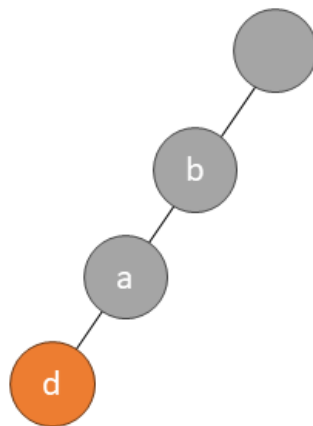
Output: [None]

addWord("bad")



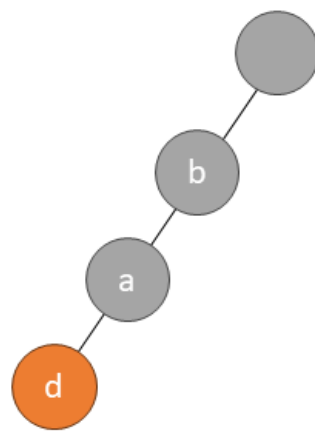
Output: [None]

addWord("bad")



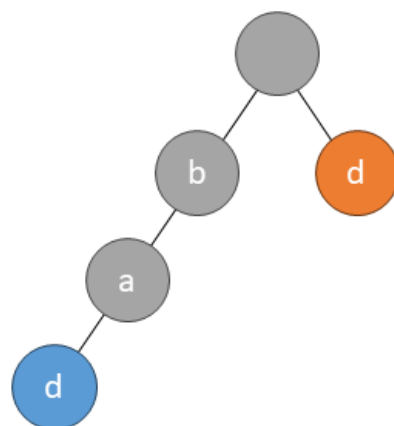
Output: [None]

addWord("bad")



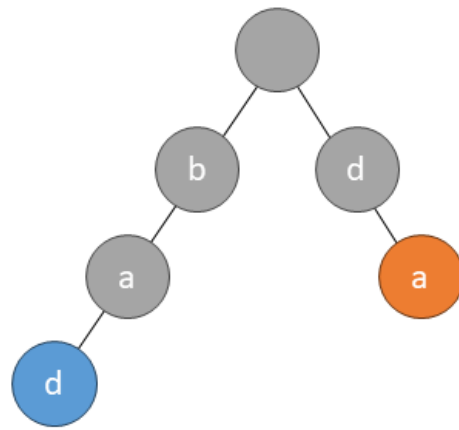
Output: [None]

addWord("dad")



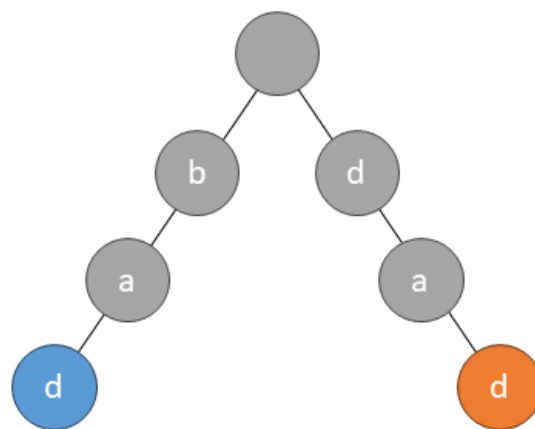
Output: [None, None]

addWord("dad")



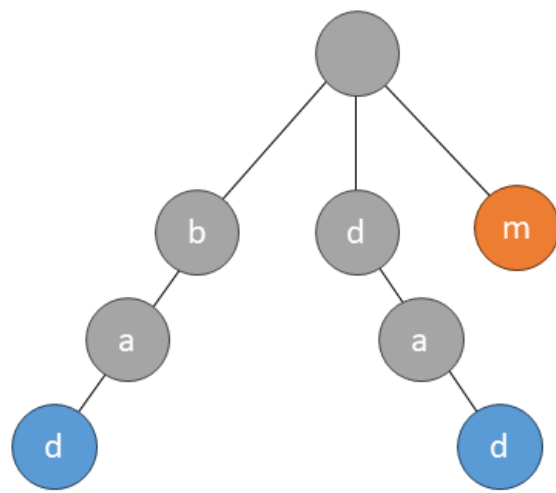
Output: [None, None]

addWord("dad")



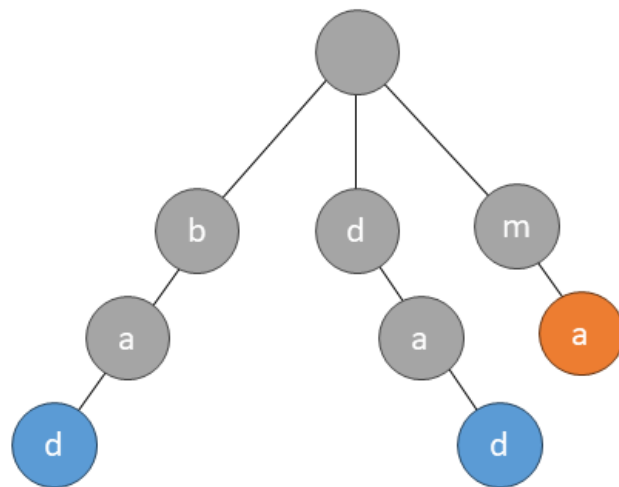
Output: [None, None]

addWord("mad")



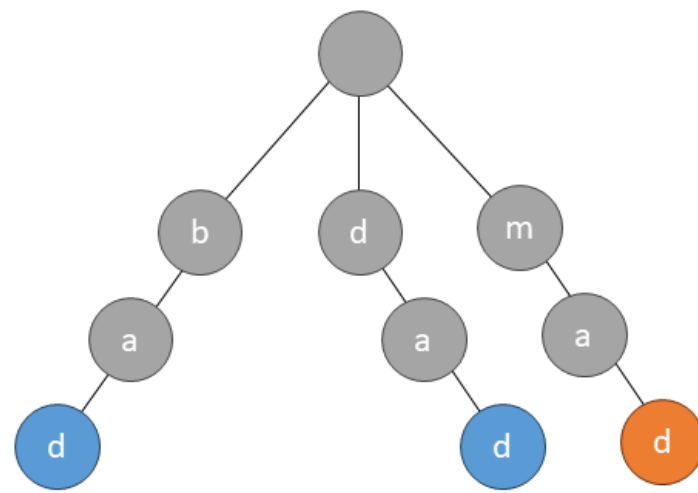
Output: [None, None, **None**]

addWord("mad")



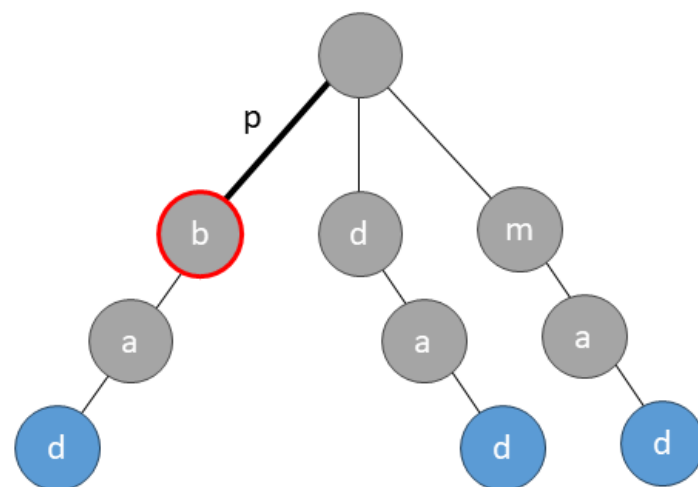
Output: [None, None, None]

addWord("mad")



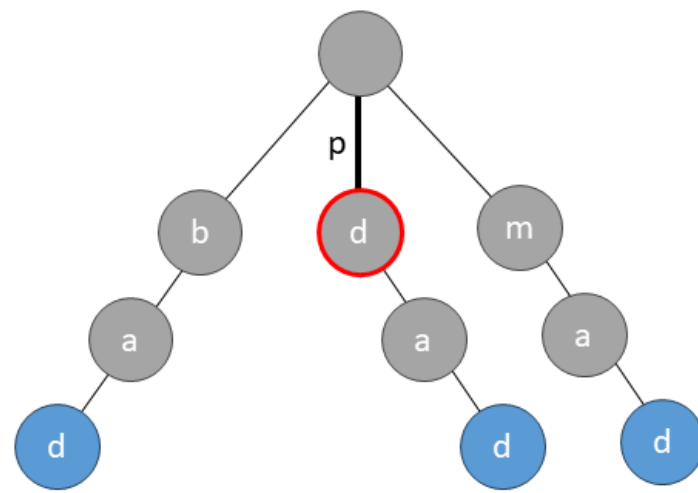
Output: [None, None, None, **None**]

search("pad")



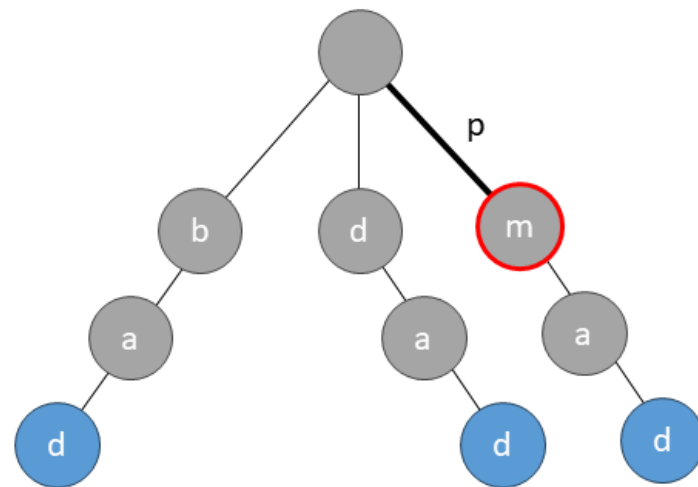
Output: [None, None, None, None]

search("pad")



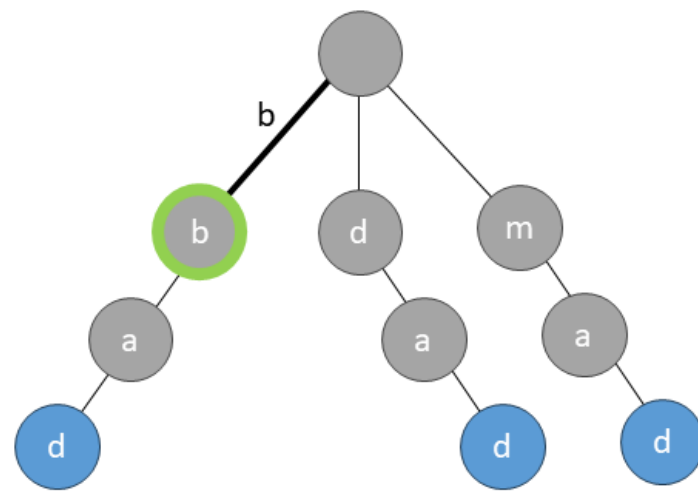
Output: [None, None, None, None]

search("pad")



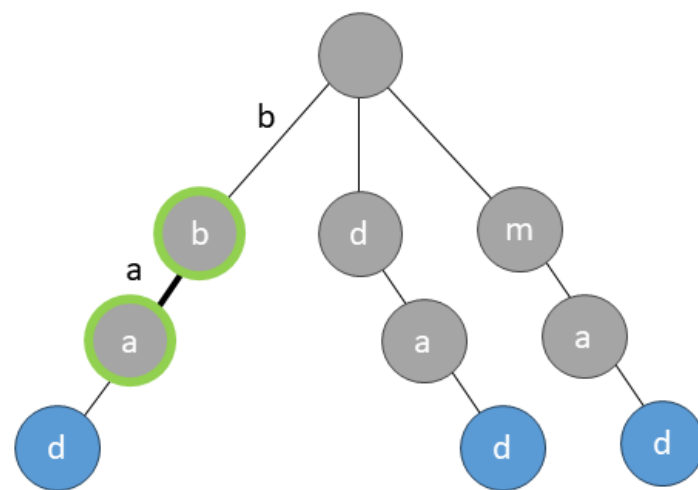
Output: [None, None, None, None, False]

search("bad")



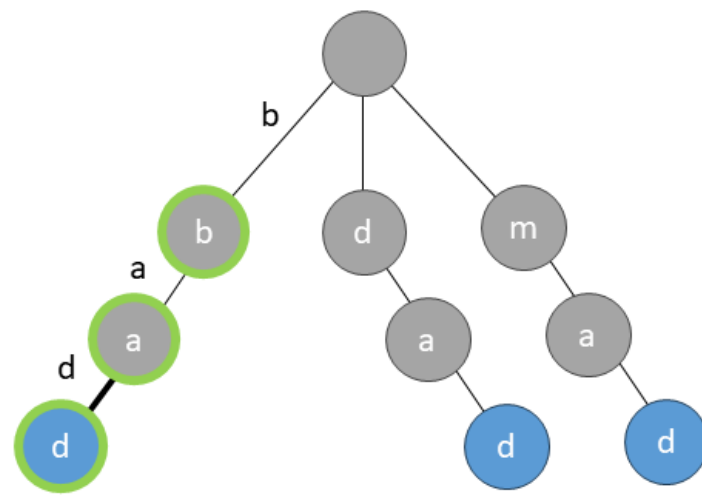
Output: [None, None, None, None, False]

search("bad")



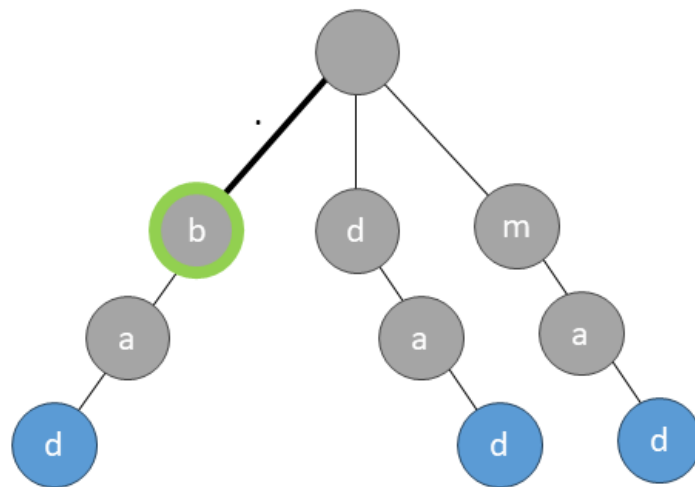
Output: [None, None, None, None, False]

search("bad")



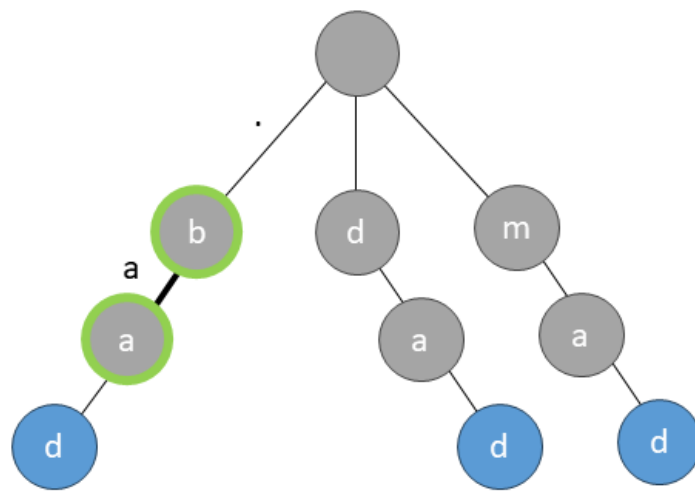
Output: [None, None, None, None, False, True]

search(".ad")



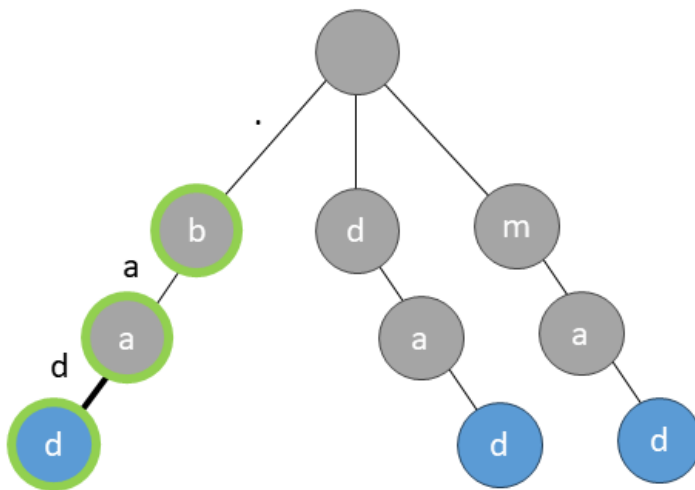
Output: [None, None, None, None, False, True]

search(".ad")



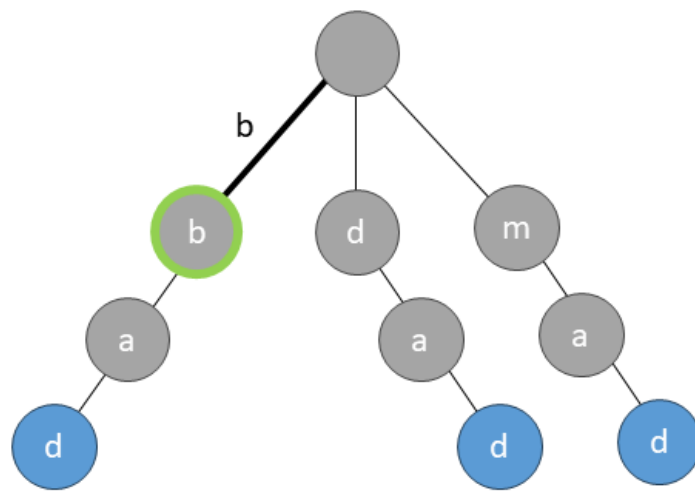
Output: [None, None, None, None, False, True]

search(".ad")



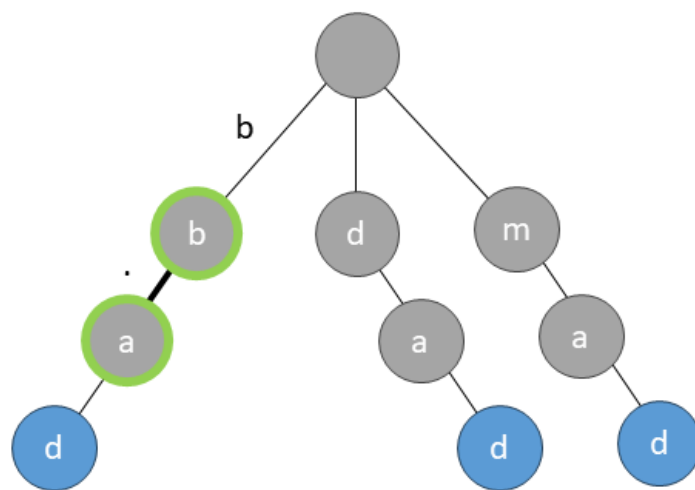
Output: [None, None, None, None, False, True, True]

search("b..")



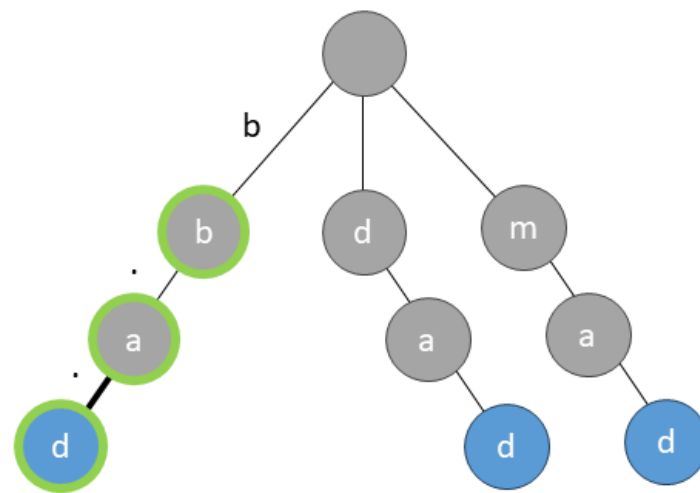
Output: [None, None, None, None, False, True, True]

search("b..")



Output: [None, None, None, None, False, True, True]

search("b..")



Output: [None, None, None, None, False, True, True, True]

Problem 3: Leetcode - [Word Search II](#)

#word_search_II

Hard

```
class TrieNode():
    def __init__(self):
        self.children = {}
        self.isWord = False
    def addWord(self, word):
        cur = self
        for c in word:
            if c not in cur.children:
                cur.children[c] = TrieNode()
            cur = cur.children[c]
        cur.isWord = True
class WordSearch(object):
    def findWords(self, board, words):
        root = TrieNode()
        for w in words:
            root.addWord(w)
        ROWS, COLS = len(board), len(board[0])
        res, visited = set(), set()
        def dfs(r, c, node, word):
            if (r<0 or c<0 or
                c == COLS or r == ROWS or
                (r,c) in visited or
                board[r][c] not in node.children):
                #if the current position is out of bounds or visited or not a child node
                return
            if node.children == {}:
                return
            visited.add((r,c))

            node = node.children[board[r][c]]
            word += board[r][c]
            # if current node isWord then we add it to the result
            if node.isWord == True:
                res.add(word)
            dfs(r-1,c, node, word) #down
            dfs(r+1,c, node, word) #up
            dfs(r,c+1, node, word) #right
            dfs(r,c-1, node, word) #left
```

```
        visited.remove((r,c))
    for r in range(ROWS):
        for c in range(COLS):
            dfs(r,c,root,"")

    return list(res)

board = [
    ["o","a","a","n"],
    ["e","t","a","e"],
    ["i","h","k","r"],
    ["i","f","l","v"]
]
words = ["oath","pea","eat","rain"]
obj = WordSearch()
print(obj.findWords(board, words))
```

Explanation

Question 1: What does the 'addWord' function do?

```
{
  "root": {
    "children": {
      "o": {
        "children": {
          "a": {
            "children": {
              "t": {
                "children": {
                  "h": {
                    "children": {},
                    "endOfWord": true
                  }
                }
              },
              "endOfWord": false
            }
          },
          "endOfWord": false
        }
      },
      "endOfWord": false
    },
    "p": {
      "children": {
        "e": {
          "children": {
            "a": {
              "children": {},
              "endOfWord": true
            }
          },
          "endOfWord": false
        }
      },
      "endOfWord": false
    },
    "e": {
      "children": {
        "a": {
          "children": {
            "t": {
              "children": {},
              "endOfWord": true
            }
          },
          "endOfWord": false
        }
      },
      "endOfWord": false
    },
    "r": {
      "children": {
        "a": {
          "children": {
            "i": {
              "children": {
                "n": {
```

```

    "children": {},
    "endOfWord": true
  },
},
"endOfWord": false
},
},
"endOfWord": false
},
},
"endOfWord": false
},
},
"endOfWord": false
}
}
```

Question 2: How does the grid traversal work?

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

start:

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	l	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	l	a	e
i	h	k	r
i	f	l	v

output: ["oath"]

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

Then eventually:

o	a	a	n
e	t	a	e
i	h	k	r

o	a	a	n
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v
o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

output: ["oath","eat"]

End