class TrieNode:

def \_\_init\_\_(self):

self.children = {}

self.is\_word = False

class Trie:

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word):

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_word = True

def search(self, word):

node = self.root

for char in word:

if char not in node.children:

return False

node = node.children[char]

return node.is\_word

def starts\_with(self, prefix):

node = self.root

for char in prefix:

if char not in node.children:

return False

node = node.children[char]

return True

class WordSearchGame:

def \_\_init\_\_(self, grid, dictionary):

self.grid = grid

self.num\_rows = len(grid)

self.num\_cols = len(grid[0])

self.dictionary = dictionary

self.trie = Trie()

def build\_trie(self):

for word in self.dictionary:

self.trie.insert(word)

def search\_word(self, row, col, visited, prefix, found\_words):

directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1)]

if not self.trie.starts\_with(prefix):

return

visited[row][col] = True

prefix += self.grid[row][col]

if self.trie.search(prefix):

found\_words.add(prefix)

for direction in directions:

new\_row = row + direction[0]

new\_col = col + direction[1]

if 0 <= new\_row < self.num\_rows and 0 <= new\_col < self.num\_cols and not visited[new\_row][new\_col]:

self.search\_word(new\_row, new\_col, visited, prefix, found\_words)

visited[row][col] = False

def find\_words(self):

self.build\_trie()

visited = [[False] \* self.num\_cols for \_ in range(self.num\_rows)]

found\_words = set()

for i in range(self.num\_rows):

for j in range(self.num\_cols):

self.search\_word(i, j, visited, '', found\_words)

return found\_words

# Example usage

grid = [

['P', 'Y', 'T', 'H','O', 'N', 'C', 'D'],

['E', 'O', 'G', 'H','A', 'B', 'C', 'D'],

['I', 'J', 'O', 'L','I', 'V', 'E', 'R'],

['M', 'N', 'O', 'R','A', 'B', 'C', 'D']

]

dictionary = ['PYTHON', 'POOR', 'OLIVER']

while True:

print("1.Search Word and Append if not found\n2.delete\n3.quit")

Choice = int(input("ENTER CHOICE(1-3):"))

if Choice ==1:

word = input("Enter the word you want to search: ")

if word in dictionary:

print("successuful search")

else:

dictionary.append(word)

print("word added to dictionary")

elif Choice==2:

dictionary.pop()

print("word deleted successfully")

elif Choice ==3:

break

else:

print("wrong choice")

nextd =input("do you want to continues (yes\no):")

if nextd.lower()!="yes":

break

game = WordSearchGame(grid, dictionary)

words\_found = game.find\_words()

Algorithm:

The provided program implements a word search game using a trie data structure. Here's an algorithmic breakdown of the program:

Define the TrieNode class with a dictionary to store child nodes and a boolean flag to indicate if it's the end of a word.

Define the Trie class with a root node, which initializes an empty trie.

Implement the insert method in the Trie class:

Start at the root node.

Iterate over each character in the word.

If the character is not a child of the current node, create a new node and add it to the children dictionary.

Update the current node to the child node corresponding to the character.

Mark the last node as the end of a word.

Implement the search method in the Trie class:

Start at the root node.

Iterate over each character in the word.

If the character is not a child of the current node, return False.

Update the current node to the child node corresponding to the character.

After iterating through all characters, return the value of the is\_word flag of the last node.

Implement the starts\_with method in the Trie class:

Start at the root node.

Iterate over each character in the prefix.

If the character is not a child of the current node, return False.

Update the current node to the child node corresponding to the character.

After iterating through all characters, return True.

Define the WordSearchGame class with grid, dictionary, and trie attributes.

Implement the build\_trie method in the WordSearchGame class:

Iterate over each word in the dictionary.

Insert each word into the trie using the insert method.

Implement the search\_word method in the WordSearchGame class:

Define the possible directions: up, down, left, right, and diagonal.

Check if the current prefix exists as a word in the trie. If not, return.

Mark the current position in the grid as visited and add the current character to the prefix.

If the prefix exists as a word in the trie, add it to the found\_words set.

Recursively search in all possible directions from the current position, passing the updated prefix and visited matrix.

Mark the current position as unvisited to backtrack.

Implement the find\_words method in the WordSearchGame class:

Build the trie using the build\_trie method.

Initialize a visited matrix and an empty set for found\_words.

Iterate over each position in the grid.

Start the search\_word method from each position, passing the current position, empty prefix, visited matrix, and found\_words set.

Return the found\_words set.

Create a grid and dictionary.

Run a loop to interact with the user:

Ask the user for a choice (1 for search and append, 2 for delete, 3 for quit).

If the choice is 1:

Ask the user for a word to search.

Check if the word exists in the dictionary.

If it does, print a success message.

If it doesn't, append the word to the dictionary and print a confirmation message.

If the choice is 2:

Remove the last word from the dictionary.

Print a confirmation message.

If the choice is 3, break the loop.

Ask the user if they want to continue.

If the answer is not "yes," break the loop.

Create an instance of the WordSearchGame class with the grid and dictionary.

Call the find\_words