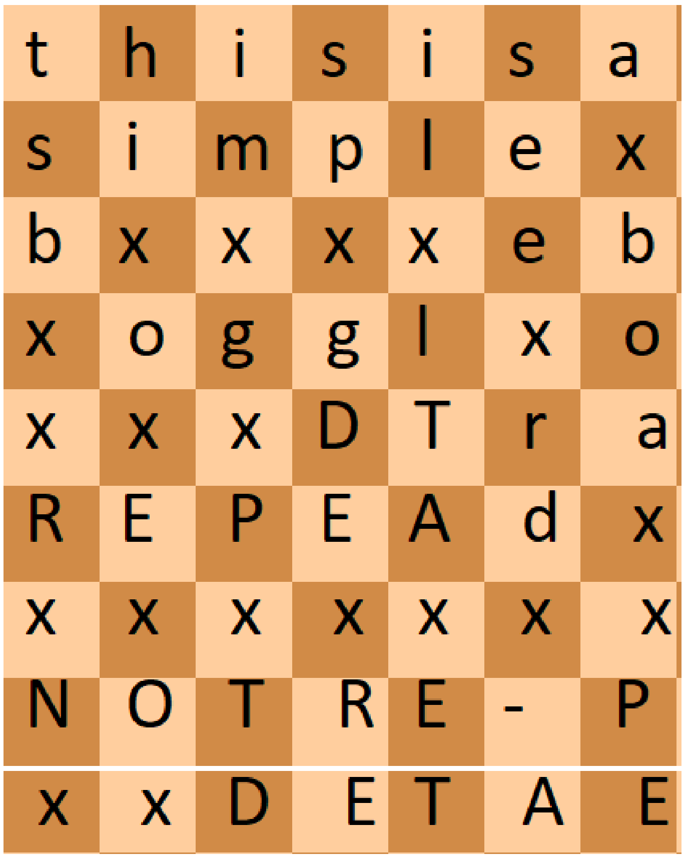


t	h	i	s	i	s	a
s	i	m	p	l	e	x
b	x	x	x	x	e	b
x	o	g	g	l	x	o
x	x	x	D	T	r	a
R	E	P	E	A	d	x
x	x	x	x	x	x	x
N	O	T	R	E	-	P
x	x	D	E	T	A	E

This is a boggle board.

We are given a board set as like as shown in the above figure(sample). We have to find the words contained in this boggle board which are also in the list of words we have....

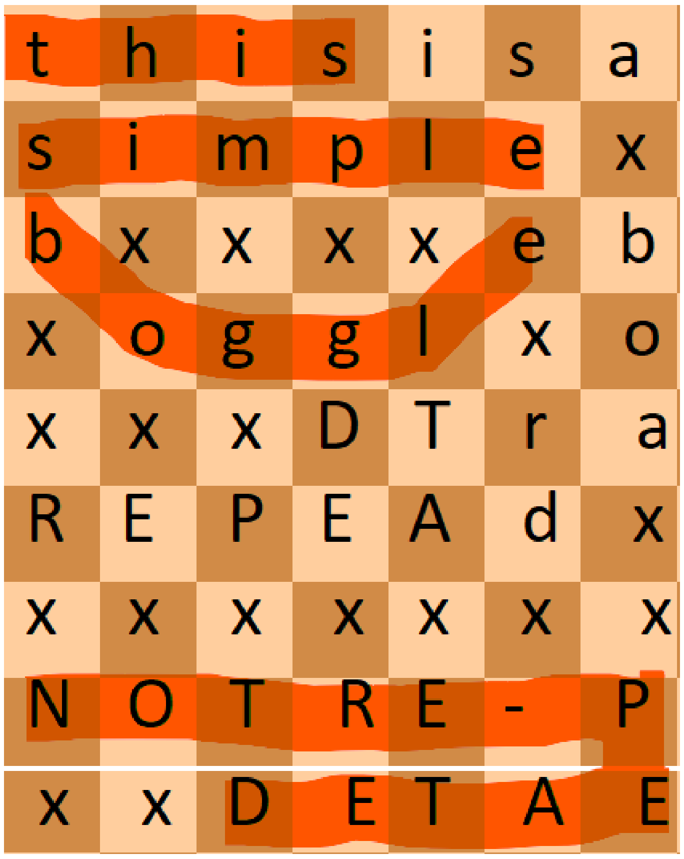
Here is the sample list of word pertinent to above sample figure:

```
[
  "this",
  "is",
  "not",
  "a",
  "simple",
  "boggle",
  "board",
  "test",
  "REPEATED",
  "NOTRE-PEATED"
]
```

Rules:

To find a word in the board, we can traverse from the board at any direction (horizontally, vertically or diagonally) but the same letter cannot be repeated.

For Example:



t	h	i	s	i	s	a
s	i	m	p	l	e	x
b	x	x	x	x	e	b
x	o	g	g	l	x	o
x	x	x	D	T	r	a
R	E	P	E	A	d	x
x	x	x	x	x	x	x
N	O	T	R	E	-	P
x	x	D	E	T	A	E

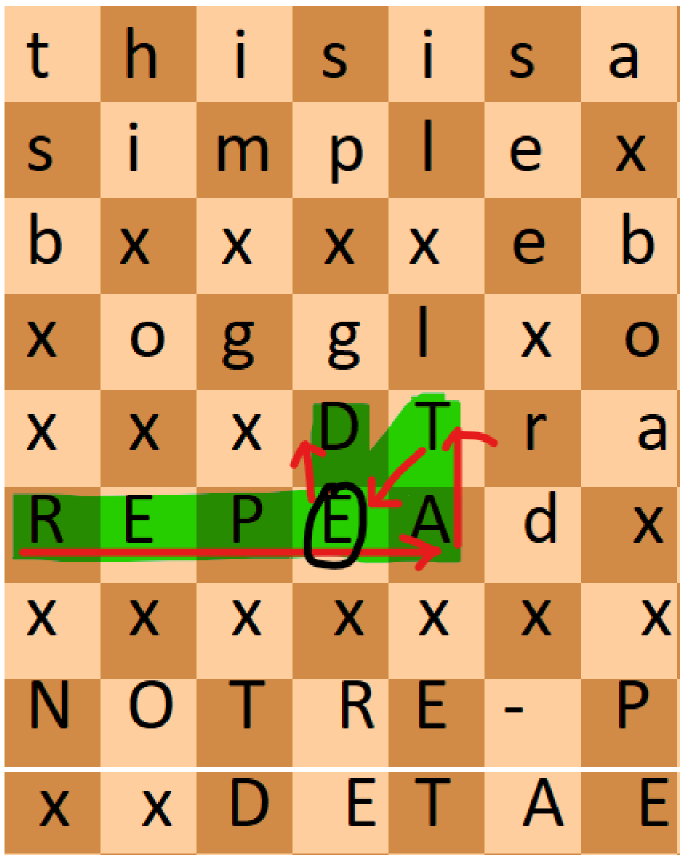
These are some valid words:

-> this

-> simple

-> boggle

-> NOTRE-PEATED



t	h	i	s	i	s	a
s	i	m	p	l	e	x
b	x	x	x	x	e	b
x	o	g	g	l	x	o
x	x	x	D	T	r	a
R	E	P	E	A	d	x
x	x	x	x	x	x	x
N	O	T	R	E	-	P
x	x	D	E	T	A	E

This is not a valid word selection as circled E is repeating and hence 'repeated' is not the valid selection.

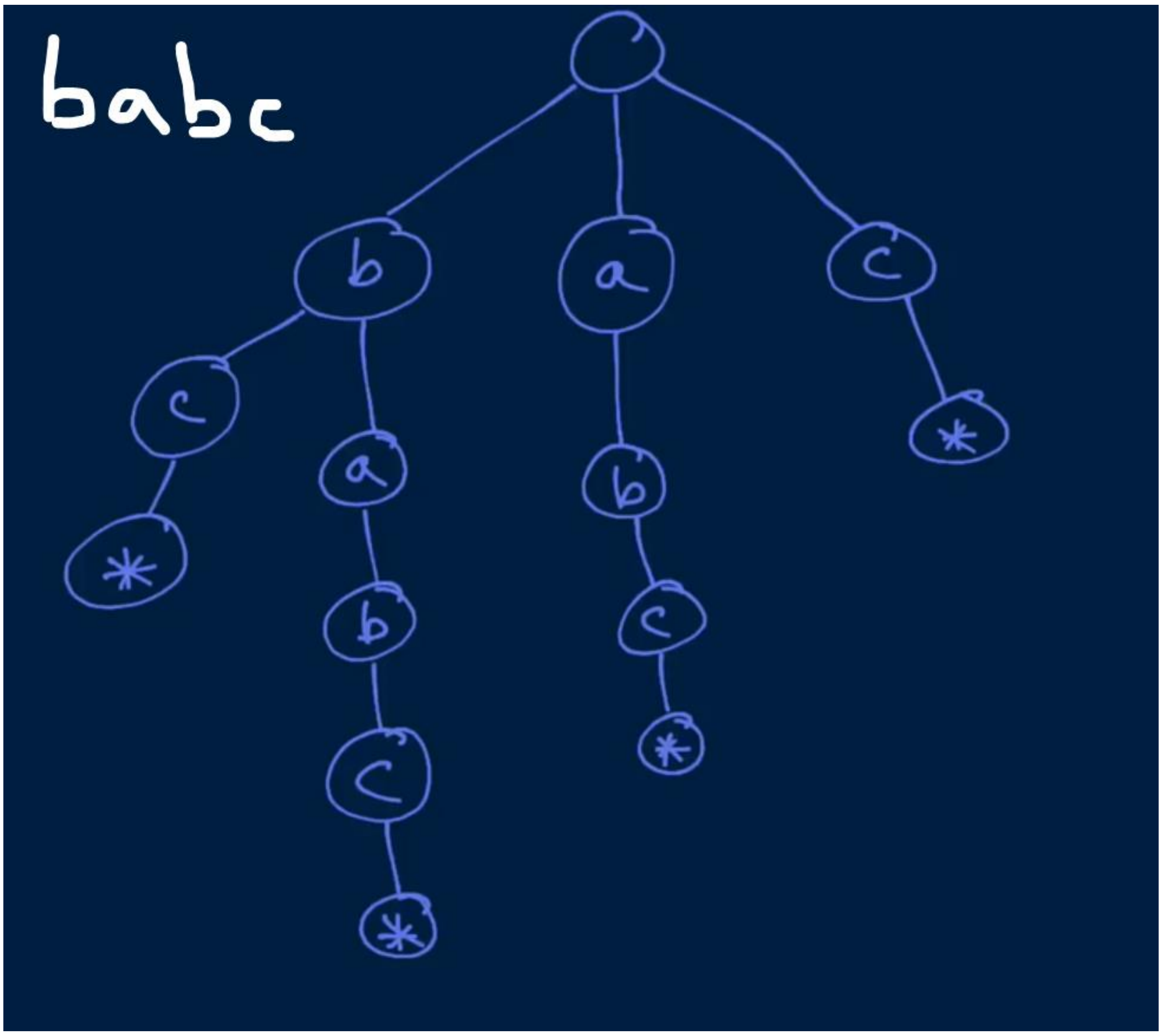
Let's see how it can be done.

A simple brute force traversing every letter in the board and then checking its neighbours and then storing it in call stack is the first thing that come to mind. But Imagine such a huge call stack, and even if we achieve doing that how would we keep track of word, if we found one. And even we somehow use an algorithmic back-tracking approach, the time complexity of this naïve algorithm would be huge.

Time Complexity: $O((nm)^{nm}) + f(n, m))$ which is huge for even small n and m.

So, to avoid that we will use a string-searching data-structure that is **suffix-tries**. And for traversing the neighbours we will use an efficient DFS-algorithm (searching clockwise).

Suffix Tries:



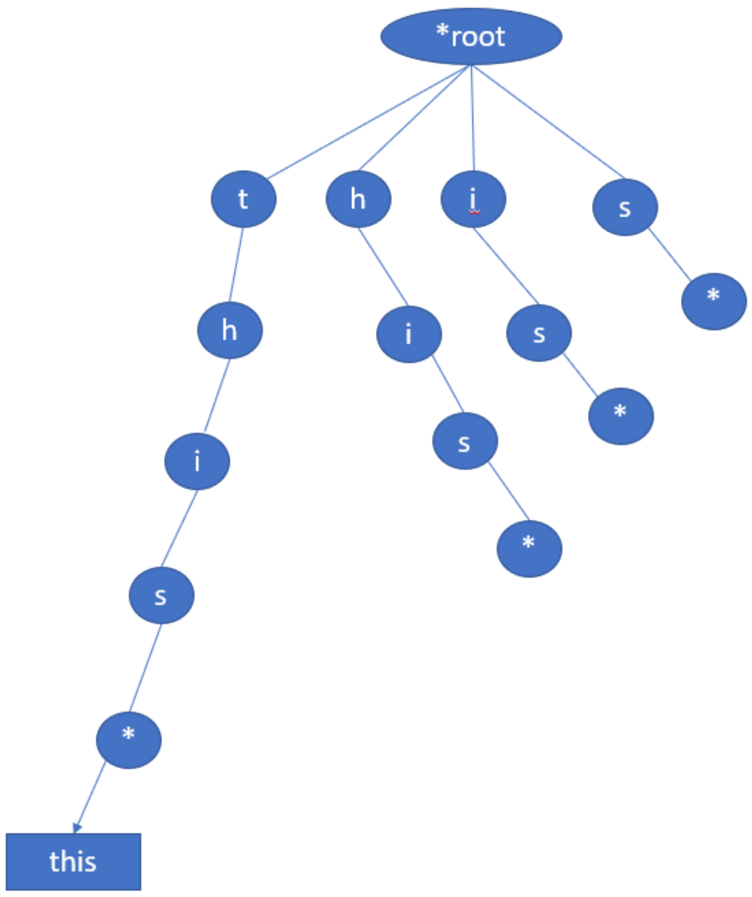
Here is a suffix trie of word "babc".

- ➔ All the circles are a hash-map and the line can be approximated as the pointers to other hash-map. Hence a *hash-map of hash-map of* ……..
- ➔ So here root is a hashmap containg b , a and c
- ➔ b is a hashmap having value a and c and so on……
- ➔ The last element is called endSymbol denoted by '*'

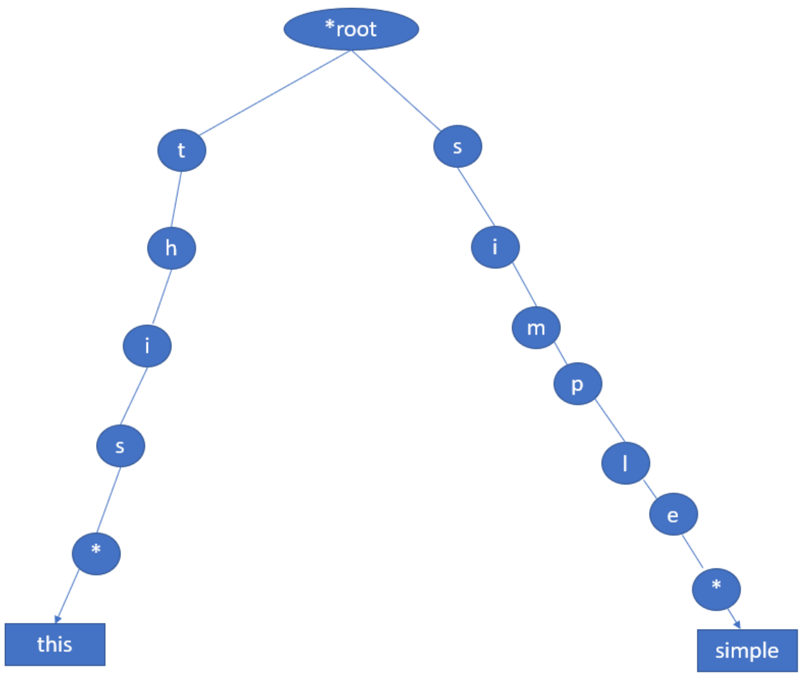
Suffix Tries are used primarily for String-searching in a efficient manner.

I've created suffix tries of the given words will be using them to search for the words in the board.

This is a modified version of suffix trie, where the last element will further point to the word(member of the class 'Trie').



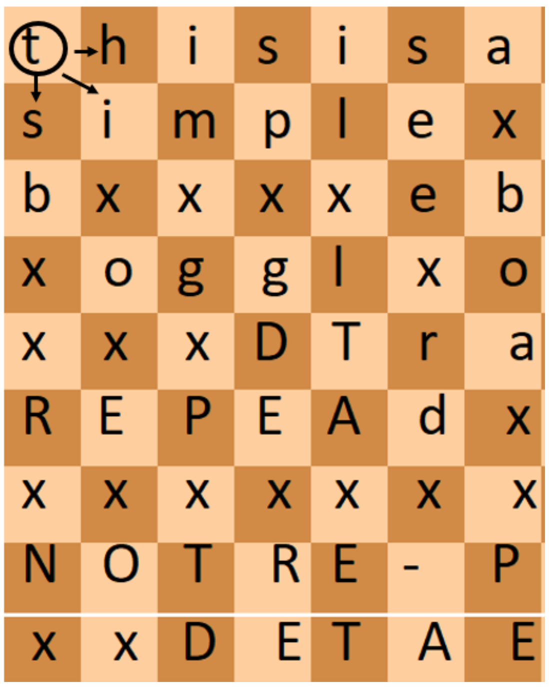
Let’s see an example to completely understand the working of code and see how we will traverse the board.



This is a part of final suffix trie.

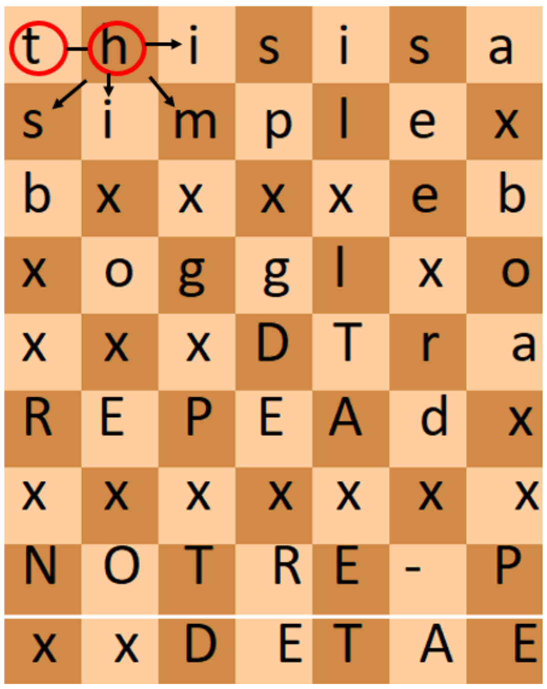
1) We will start from ‘t’ and search for ‘t’ in the suffix tree, and Since ‘t’ is a children of root **mark ‘t’ as visited(red)** and move the pointer from root to ‘t’.

2)



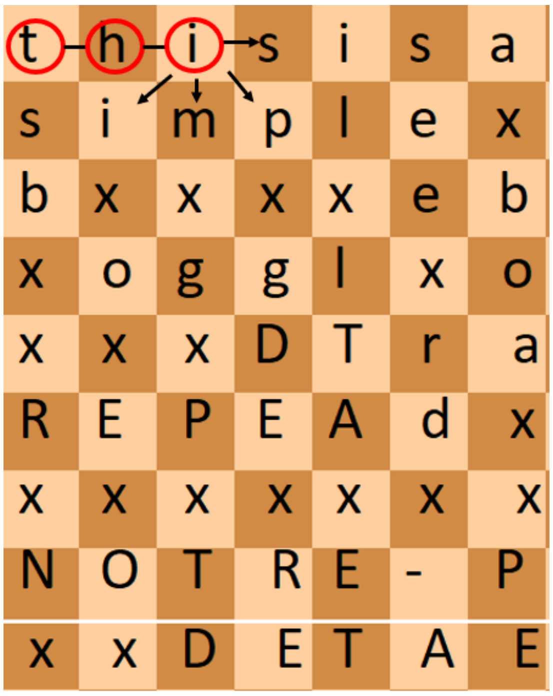
Search for t’s neighbour clockwise and recursively. If found in children of ‘t’.Mark them visited and Move pointer to the next letter.

3)

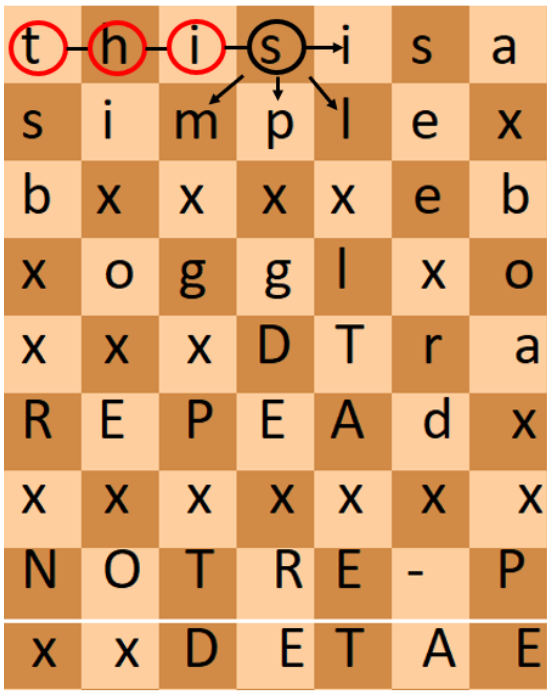


Since ‘h’ was found in the trie,we will mark ‘h’ as visited. Similarly search for h’

4)

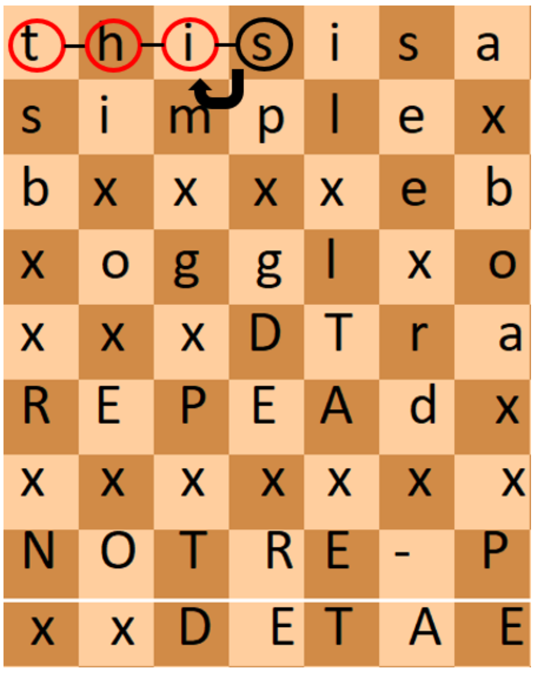


5)

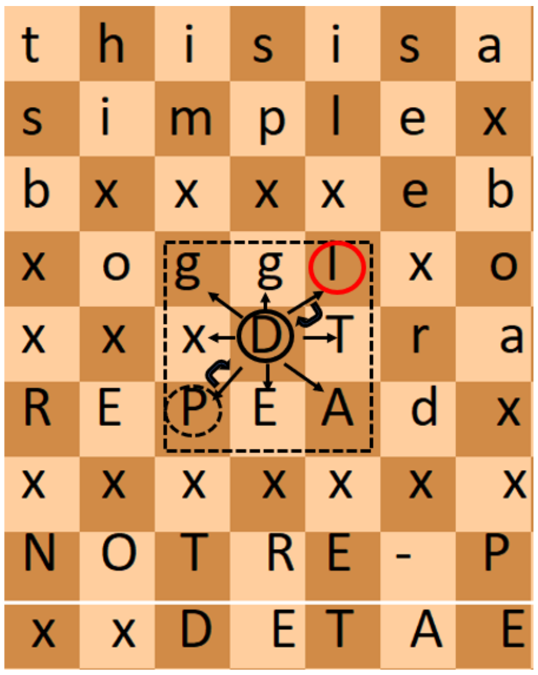


S is pointing to '*' and hence it is the end letter. Since here '*' is indeed pointing to a word, we have found our word "this".

6)



As the word was found, append it in the final list of found words. And now that we have reached the end, we will backtrack and mark the visited as unvisited ,as there is possibility of another word from previous suffixes.



This is a general representation of traversal to neighbour in the board.

- ➔ If call stack reaches a letter already marked as visited, it will return.
- ➔ If call stack reaches a letter which is not there in the suffix trie, it will return.
- ➔ If call stack reaches a letter found in the suffix trie, it will move to that letter and marked previous as visited.