Problem 208: Implement Trie (Prefix Tree)

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

A

trie

(pronounced as "try") or

prefix tree

is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

Trie()

Initializes the trie object.

void insert(String word)

Inserts the string

word

into the trie.

boolean search(String word)

Returns

true

if the string

word

is in the trie (i.e., was inserted before), and

false

otherwise.

boolean startsWith(String prefix)

Returns

true

if there is a previously inserted string

word

that has the prefix

prefix

, and

false

otherwise.

Example 1:

Input

["Trie", "insert", "search", "search", "startsWith", "insert", "search"] [[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]

Output

[null, null, true, false, true, null, true]

Explanation

Trie trie = new Trie(); trie.insert("apple"); trie.search("apple"); // return True trie.search("app"); // return False trie.startsWith("app"); // return True trie.insert("app"); trie.search("app"); // return True

Constraints:

1 <= word.length, prefix.length <= 2000

word

and

prefix

consist only of lowercase English letters.

At most

3 * 10

4

calls

in total

will be made to

insert

,

search

, and

startsWith

.

Code Snippets

C++:

```
class Trie {
public:
Trie() {

}

void insert(string word) {

}

bool search(string word) {

}

bool startsWith(string prefix) {

}
};

/**
* Your Trie object will be instantiated and called as such:
* Trie* obj = new Trie();
* obj->insert(word);
* bool param_2 = obj->search(word);
* bool param_3 = obj->startsWith(prefix);
*/
```

Java:

```
class Trie {

public Trie() {

}

public void insert(String word) {

}

public boolean search(String word) {

}

public boolean startsWith(String prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
* Trie obj = new Trie();
* obj.insert(word);
* boolean param_2 = obj.search(word);
* boolean param_3 = obj.startsWith(prefix);
*/
```

Python3:

```
class Trie:

def __init__(self):


def insert(self, word: str) -> None:


def search(self, word: str) -> bool:
```

```
def startsWith(self, prefix: str) -> bool:



# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

Python:

```
class Trie(object):

def __init__(self):


def insert(self, word):
"""
:type word: str
:rtype: None
"""


def search(self, word):
"""
:type word: str
:rtype: bool
"""


def startsWith(self, prefix):
"""
:type prefix: str
:rtype: bool
"""



# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
```

```
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

JavaScript:

```
var Trie = function() {

};

/**
* @param {string} word
* @return {void}
*/
Trie.prototype.insert = function(word) {

};

/**
* @param {string} word
* @return {boolean}
*/
Trie.prototype.search = function(word) {

};

/**
* @param {string} prefix
* @return {boolean}
*/
Trie.prototype.startsWith = function(prefix) {

};

/**
* Your Trie object will be instantiated and called as such:
* var obj = new Trie()
* obj.insert(word)
* var param_2 = obj.search(word)
* var param_3 = obj.startsWith(prefix)
*/
```

TypeScript:

```
class Trie {
constructor() {

}

insert(word: string): void {

}

search(word: string): boolean {

}

startsWith(prefix: string): boolean {

}
}

/**
* Your Trie object will be instantiated and called as such:
* var obj = new Trie()
* obj.insert(word)
* var param_2 = obj.search(word)
* var param_3 = obj.startsWith(prefix)
*/
```

C#:

```
public class Trie {

public Trie() {

}

public void Insert(string word) {

}

public bool Search(string word) {
```

```
}

public bool StartsWith(string prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
* Trie obj = new Trie();
* obj.Insert(word);
* bool param_2 = obj.Search(word);
* bool param_3 = obj.StartsWith(prefix);
*/
```

C:

```
typedef struct {

} Trie;


Trie* trieCreate() {

}

void trieInsert(Trie* obj, char* word) {

}

bool trieSearch(Trie* obj, char* word) {

}

bool trieStartsWith(Trie* obj, char* prefix) {

}

void trieFree(Trie* obj) {
```

```
}

/**
* Your Trie struct will be instantiated and called as such:
* Trie* obj = trieCreate();
* trieInsert(obj, word);

* bool param_2 = trieSearch(obj, word);

* bool param_3 = trieStartsWith(obj, prefix);

* trieFree(obj);
*/
```

Go:

```
type Trie struct {

}


func Constructor() Trie {

}


func (this *Trie) Insert(word string) {

}


func (this *Trie) Search(word string) bool {

}


func (this *Trie) StartsWith(prefix string) bool {

}
```

```
/**
* Your Trie object will be instantiated and called as such:
* obj := Constructor();
* obj.Insert(word);
* param_2 := obj.Search(word);
* param_3 := obj.StartsWith(prefix);
*/
```

Kotlin:

```
class Trie() {

fun insert(word: String) {

}

fun search(word: String): Boolean {

}

fun startsWith(prefix: String): Boolean {

}

}

/**
* Your Trie object will be instantiated and called as such:
* var obj = Trie()
* obj.insert(word)
* var param_2 = obj.search(word)
* var param_3 = obj.startsWith(prefix)
*/
```

Swift:

```
class Trie {

init() {

}
```

```
func insert(_ word: String) {

}

func search(_ word: String) -> Bool {

}

func startsWith(_ prefix: String) -> Bool {

}
}

/**
* Your Trie object will be instantiated and called as such:
* let obj = Trie()
* obj.insert(word)
* let ret_2: Bool = obj.search(word)
* let ret_3: Bool = obj.startsWith(prefix)
*/
```

Rust:

```
struct Trie {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Trie {

fn new() -> Self {

}

fn insert(&self, word: String) {

}
```

```
fn search(&self, word: String) -> bool {

}

fn starts_with(&self, prefix: String) -> bool {

}
}

/**
* Your Trie object will be instantiated and called as such:
* let obj = Trie::new();
* obj.insert(word);
* let ret_2: bool = obj.search(word);
* let ret_3: bool = obj.starts_with(prefix);
*/
```

Ruby:

```
class Trie
def initialize()

end


=begin
:type word: String
:rtype: Void
=end
def insert(word)

end


=begin
:type word: String
:rtype: Boolean
=end
def search(word)

end
```

```
=begin
:type prefix: String
:rtype: Boolean
=end
def starts_with(prefix)

end



end

# Your Trie object will be instantiated and called as such:
# obj = Trie.new()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.starts_with(prefix)
```

PHP:

```
class Trie {
/**
*/
function __construct() {

}

/**
* @param String $word
* @return NULL
*/
function insert($word) {

}

/**
* @param String $word
* @return Boolean
*/
function search($word) {
```

```
}

/**
* @param String $prefix
* @return Boolean
*/
function startsWith($prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
* $obj = Trie();
* $obj->insert($word);
* $ret_2 = $obj->search($word);
* $ret_3 = $obj->startsWith($prefix);
*/
```

Dart:

```
class Trie {

Trie() {

}

void insert(String word) {

}

bool search(String word) {

}

bool startsWith(String prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
```

```
* Trie obj = Trie();
* obj.insert(word);
* bool param2 = obj.search(word);
* bool param3 = obj.startsWith(prefix);
*/
```

Scala:

```
class Trie() {

def insert(word: String): Unit = {

}

def search(word: String): Boolean = {

}

def startsWith(prefix: String): Boolean = {

}

}

/**
* Your Trie object will be instantiated and called as such:
* val obj = new Trie()
* obj.insert(word)
* val param_2 = obj.search(word)
* val param_3 = obj.startsWith(prefix)
*/
```

Elixir:

```
defmodule Trie do
@spec init_() :: any
def init_() do

end

@spec insert(word :: String.t) :: any
def insert(word) do
```

```
end

@spec search(word :: String.t) :: boolean
def search(word) do

end

@spec starts_with(prefix :: String.t) :: boolean
def starts_with(prefix) do

end
end

# Your functions will be called as such:
# Trie.init_()
# Trie.insert(word)
# param_2 = Trie.search(word)
# param_3 = Trie.starts_with(prefix)

# Trie.init_ will be called before every test case, in which you can do some
necessary initializations.
```

Erlang:

```
-spec trie_init_() -> any().
trie_init_() ->
.

-spec trie_insert(Word :: unicode:unicode_binary()) -> any().
trie_insert(Word) ->
.

-spec trie_search(Word :: unicode:unicode_binary()) -> boolean().
trie_search(Word) ->
.

-spec trie_starts_with(Prefix :: unicode:unicode_binary()) -> boolean().
trie_starts_with(Prefix) ->
.
```

```
%% Your functions will be called as such:
%% trie_init_(),
%% trie_insert(Word),
%% Param_2 = trie_search(Word),
%% Param_3 = trie_starts_with(Prefix),

%% trie_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket:

```
(define trie%
(class object%
(super-new)

(init-field)

; insert : string? -> void?
(define/public (insert word)
)
; search : string? -> boolean?
(define/public (search word)
)
; starts-with : string? -> boolean?
(define/public (starts-with prefix)
)))

;; Your trie% object will be instantiated and called as such:
;; (define obj (new trie%))
;; (send obj insert word)
;; (define param_2 (send obj search word))
;; (define param_3 (send obj starts-with prefix))
```

Solutions

C++ Solution:

```
/*
* Problem: Implement Trie (Prefix Tree)
* Difficulty: Medium
```

```
* Tags: string, tree, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Trie {
public:
Trie() {

}

void insert(string word) {

}

bool search(string word) {

}

bool startsWith(string prefix) {

}
};

/**
* Your Trie object will be instantiated and called as such:
* Trie* obj = new Trie();
* obj->insert(word);
* bool param_2 = obj->search(word);
* bool param_3 = obj->startsWith(prefix);
*/
```

Java Solution:

```
/**
* Problem: Implement Trie (Prefix Tree)
* Difficulty: Medium
* Tags: string, tree, hash, search
*
```

```
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Trie {

public Trie() {

}

public void insert(String word) {

}

public boolean search(String word) {

}

public boolean startsWith(String prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
* Trie obj = new Trie();
* obj.insert(word);
* boolean param_2 = obj.search(word);
* boolean param_3 = obj.startsWith(prefix);
*/
```

Python3 Solution:

```
"""
Problem: Implement Trie (Prefix Tree)
Difficulty: Medium
Tags: string, tree, hash, search

Approach: String manipulation with hash map or two pointers
Time Complexity: O(n) or O(n log n)
```

```
Space Complexity: O(h) for recursion stack where h is height
"""

class Trie:

def __init__(self):


def insert(self, word: str) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Trie(object):

def __init__(self):


def insert(self, word):
"""
:type word: str
:rtype: None
"""


def search(self, word):
"""
:type word: str
:rtype: bool
"""


def startsWith(self, prefix):
"""
:type prefix: str
:rtype: bool
"""
```

```
# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

JavaScript Solution:

```
/**
* Problem: Implement Trie (Prefix Tree)
* Difficulty: Medium
* Tags: string, tree, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/


var Trie = function() {

};

/**
* @param {string} word
* @return {void}
*/
Trie.prototype.insert = function(word) {

};

/**
* @param {string} word
* @return {boolean}
*/
Trie.prototype.search = function(word) {

};

/**
* @param {string} prefix
```

```
* @return {boolean}
*/
Trie.prototype.startsWith = function(prefix) {

};

/**
* Your Trie object will be instantiated and called as such:
* var obj = new Trie()
* obj.insert(word)
* var param_2 = obj.search(word)
* var param_3 = obj.startsWith(prefix)
*/
```

TypeScript Solution:

```
/**
* Problem: Implement Trie (Prefix Tree)
* Difficulty: Medium
* Tags: string, tree, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

class Trie {
constructor() {

}

insert(word: string): void {

}

search(word: string): boolean {

}

startsWith(prefix: string): boolean {
```

```
}
}

/**
* Your Trie object will be instantiated and called as such:
* var obj = new Trie()
* obj.insert(word)
* var param_2 = obj.search(word)
* var param_3 = obj.startsWith(prefix)
*/
```

C# Solution:

```
/*
* Problem: Implement Trie (Prefix Tree)
* Difficulty: Medium
* Tags: string, tree, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

public class Trie {

public Trie() {

}

public void Insert(string word) {

}

public bool Search(string word) {

}

public bool StartsWith(string prefix) {

}
}
```

```
/**
* Your Trie object will be instantiated and called as such:
* Trie obj = new Trie();
* obj.Insert(word);
* bool param_2 = obj.Search(word);
* bool param_3 = obj.StartsWith(prefix);
*/
```

C Solution:

```
/*
* Problem: Implement Trie (Prefix Tree)
* Difficulty: Medium
* Tags: string, tree, hash, search
*
* Approach: String manipulation with hash map or two pointers
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/




typedef struct {

} Trie;


Trie* trieCreate() {

}

void trieInsert(Trie* obj, char* word) {

}

bool trieSearch(Trie* obj, char* word) {

}
```

```
bool trieStartsWith(Trie* obj, char* prefix) {

}

void trieFree(Trie* obj) {

}

/**
* Your Trie struct will be instantiated and called as such:
* Trie* obj = trieCreate();
* trieInsert(obj, word);

* bool param_2 = trieSearch(obj, word);

* bool param_3 = trieStartsWith(obj, prefix);

* trieFree(obj);
*/
```

Go Solution:

```
// Problem: Implement Trie (Prefix Tree)
// Difficulty: Medium
// Tags: string, tree, hash, search
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

type Trie struct {

}


func Constructor() Trie {

}


func (this *Trie) Insert(word string) {
```

```
}


func (this *Trie) Search(word string) bool {

}


func (this *Trie) StartsWith(prefix string) bool {

}


/**
* Your Trie object will be instantiated and called as such:
* obj := Constructor();
* obj.Insert(word);
* param_2 := obj.Search(word);
* param_3 := obj.StartsWith(prefix);
*/
```

Kotlin Solution:

```
class Trie() {

fun insert(word: String) {

}

fun search(word: String): Boolean {

}

fun startsWith(prefix: String): Boolean {

}

}

/**
```

```
* Your Trie object will be instantiated and called as such:
* var obj = Trie()
* obj.insert(word)
* var param_2 = obj.search(word)
* var param_3 = obj.startsWith(prefix)
*/
```

Swift Solution:

```
class Trie {

init() {

}

func insert(_ word: String) {

}

func search(_ word: String) -> Bool {

}

func startsWith(_ prefix: String) -> Bool {

}
}

/**
* Your Trie object will be instantiated and called as such:
* let obj = Trie()
* obj.insert(word)
* let ret_2: Bool = obj.search(word)
* let ret_3: Bool = obj.startsWith(prefix)
*/
```

Rust Solution:

```
// Problem: Implement Trie (Prefix Tree)
// Difficulty: Medium
```

```
// Tags: string, tree, hash, search
//
// Approach: String manipulation with hash map or two pointers
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

struct Trie {

}


/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl Trie {

fn new() -> Self {

}

fn insert(&self, word: String) {

}

fn search(&self, word: String) -> bool {

}

fn starts_with(&self, prefix: String) -> bool {

}
}

/**
* Your Trie object will be instantiated and called as such:
* let obj = Trie::new();
* obj.insert(word);
* let ret_2: bool = obj.search(word);
* let ret_3: bool = obj.starts_with(prefix);
*/
```

Ruby Solution:

```
class Trie
def initialize()

end


=begin
:type word: String
:rtype: Void
=end
def insert(word)

end


=begin
:type word: String
:rtype: Boolean
=end
def search(word)

end


=begin
:type prefix: String
:rtype: Boolean
=end
def starts_with(prefix)

end


end

# Your Trie object will be instantiated and called as such:
# obj = Trie.new()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.starts_with(prefix)
```

PHP Solution:

```
class Trie {
/**
*/
function __construct() {

}

/**
* @param String $word
* @return NULL
*/
function insert($word) {

}

/**
* @param String $word
* @return Boolean
*/
function search($word) {

}

/**
* @param String $prefix
* @return Boolean
*/
function startsWith($prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
* $obj = Trie();
* $obj->insert($word);
* $ret_2 = $obj->search($word);
* $ret_3 = $obj->startsWith($prefix);
*/
```

Dart Solution:

```
class Trie {

Trie() {

}

void insert(String word) {

}

bool search(String word) {

}

bool startsWith(String prefix) {

}
}

/**
* Your Trie object will be instantiated and called as such:
* Trie obj = Trie();
* obj.insert(word);
* bool param2 = obj.search(word);
* bool param3 = obj.startsWith(prefix);
*/
```

Scala Solution:

```
class Trie() {

def insert(word: String): Unit = {

}

def search(word: String): Boolean = {

}

def startsWith(prefix: String): Boolean = {
```

```
}

}

/**
* Your Trie object will be instantiated and called as such:
* val obj = new Trie()
* obj.insert(word)
* val param_2 = obj.search(word)
* val param_3 = obj.startsWith(prefix)
*/
```

Elixir Solution:

```
defmodule Trie do
@spec init_() :: any
def init_() do

end

@spec insert(word :: String.t) :: any
def insert(word) do

end

@spec search(word :: String.t) :: boolean
def search(word) do

end

@spec starts_with(prefix :: String.t) :: boolean
def starts_with(prefix) do

end
end

# Your functions will be called as such:
# Trie.init_()
# Trie.insert(word)
# param_2 = Trie.search(word)
# param_3 = Trie.starts_with(prefix)
```

```
# Trie.init_ will be called before every test case, in which you can do some
necessary initializations.
```

Erlang Solution:

```
-spec trie_init_() -> any().
trie_init_() ->
.

-spec trie_insert(Word :: unicode:unicode_binary()) -> any().
trie_insert(Word) ->
.

-spec trie_search(Word :: unicode:unicode_binary()) -> boolean().
trie_search(Word) ->
.

-spec trie_starts_with(Prefix :: unicode:unicode_binary()) -> boolean().
trie_starts_with(Prefix) ->
.


%% Your functions will be called as such:
%% trie_init_(),
%% trie_insert(Word),
%% Param_2 = trie_search(Word),
%% Param_3 = trie_starts_with(Prefix),

%% trie_init_ will be called before every test case, in which you can do some
necessary initializations.
```

Racket Solution:

```
(define trie%
(class object%
(super-new)

(init-field)

; insert : string? -> void?
```

```
(define/public (insert word)
)
; search : string? -> boolean?
(define/public (search word)
)
; starts-with : string? -> boolean?
(define/public (starts-with prefix)
)))

;; Your trie% object will be instantiated and called as such:
;; (define obj (new trie%))
;; (send obj insert word)
;; (define param_2 (send obj search word))
;; (define param_3 (send obj starts-with prefix))
```