trie_introduction

May 8, 2020

1 Trie

You've learned about Trees and Binary Search Trees. In this notebook, you'll learn about a new type of Tree called Trie. Before we dive into the details, let's talk about the kind of problem Trie can help with.

Let's say you want to build software that provides spell check. This software will only say if the word is valid or not. It doesn't give suggested words. From the knowledge you've already learned, how would you build this?

The simplest solution is to have a hashmap of all known words. It would take O(1) to see if a word exists, but the memory size would be O(n*m), where n is the number of words and m is the length of the word. Let's see how a Trie can help decrease the memory usage while sacrificing a little on performance.

1.1 Basic Trie

Let's look at a basic Trie with the following words: "a", "add", and "hi"

```
In [ ]: basic_trie = {
            # a and add word
            'a': {
                'd': {
                    'd': {'word_end': True},
                    'word_end': False},
                'word_end': True},
            # hi word
            'h': {
                'i': {'word_end': True},
                'word_end': False}}


        print('Is "a"   a word: {}'.format(basic_trie['a']['word_end']))
        print('Is "ad"  a word: {}'.format(basic_trie['a']['d']['word_end']))
        print('Is "add" a word: {}'.format(basic_trie['a']['d']['d']['word_end']))
```

You can lookup a word by checking if `word_end` is `True` after traversing all the characters in the word. Let's look at the word "hi". The first letter is "h", so you would call `basic_trie['h']`.

The second letter is "i", so you would call `basic_trie['h']['i']`. Since there's no more letters left, you would see if this is a valid word by getting the value of `word_end`. Now you have `basic_trie['h']['i']['word_end']` with `True` or `False` if the word exists.

In `basic_trie`, words "a" and "add" overlapp. This is where a Trie saves memory. Instead of having "a" and "add" in different cells, their characters treated like nodes in a tree. Let's see how we would check if a word exists in `basic_trie`.

```
In [ ]: def is_word(word):
            """
            Look for the word in `basic_trie`
            """
            current_node = basic_trie

            for char in word:
                if char not in current_node:
                    return False

                current_node = current_node[char]

            return current_node['word_end']


        # Test words
        test_words = ['ap', 'add']
        for word in test_words:
            if is_word(word):
                print('"{}" is a word.'.format(word))
            else:
                print('"{}" is not a word.'.format(word))
```

The `is_word` starts with the root node, `basic_trie`. It traverses each character (`char`) in the word (`word`). If a character doesn't exist while traversing, this means the word doesn't exist in the trie. Once all the characters are traversed, the function returns the value of `current_node['word_end']`.

You might notice the function `is_word` is similar to a binary search tree traversal. Since Trie is a tree, it makes sense that we would use a type of tree traversal. Now that you've seen a basic example of a Trie, let's build something more familiar. ## Trie Using a Class Just like most tree data structures, let's use classes to build the Trie. Implement two functions for the `Trie` class below. Implement `add` to add a word to the Trie. Implement `exists` to return `True` if the word exist in the trie and `False` if the word doesn't exist in the trie.

```
In [ ]: class TrieNode(object):
            def __init__(self):
                self.is_word = False
                self.children = {}

        class Trie(object):
            def __init__(self):
```

```
        self.root = TrieNode()

    def add(self, word):
        """
        Add `word` to trie
        """
        pass

    def exists(self, word):
        """
        Check if word exists in trie
        """
        pass
```

Hide Solution

```
In [ ]: class TrieNode(object):
    def __init__(self):
        self.is_word = False
        self.children = {}


class Trie(object):
    def __init__(self):
        self.root = TrieNode()

    def add(self, word):
        """
        Add `word` to trie
        """
        current_node = self.root

        for char in word:
            if char not in current_node.children:
                current_node.children[char] = TrieNode()
            current_node = current_node.children[char]

        current_node.is_word = True

    def exists(self, word):
        """
        Check if word exists in trie
        """
        current_node = self.root

        for char in word:
            if char not in current_node.children:
                return False
```

```
                    current_node = current_node.children[char]

                return current_node.is_word
```

```
In [ ]: word_list = ['apple', 'bear', 'goo', 'good', 'goodbye', 'goods', 'goodwill', 'gooses'   ,
        word_trie = Trie()

        # Add words
        for word in word_list:
            word_trie.add(word)

        # Test words
        test_words = ['bear', 'goo', 'good', 'goos']
        for word in test_words:
            if word_trie.exists(word):
                print('"{}" is a word.'.format(word))
            else:
                print('"{}" is not a word.'.format(word))
```

1.2 Trie using Defaultdict (Optional)

This is an optional section. Feel free to skip this and go to the next section of the classroom.

A cleaner way to build a trie is with a Python default dictionary. The following `TrieNod` class is using `collections.defaultdict` instead of a normal dictionary.

```
In [ ]: import collections


        class TrieNode:
            def __init__(self):
                self.children = collections.defaultdict(TrieNode)
                self.is_word = False
```

Implement the `add` and `exists` function below using the new `TrieNode` class.

```
In [ ]: class Trie(object):
            def __init__(self):
                self.root = TrieNode()

            def add(self, word):
                """
                Add `word` to trie
                """
                pass

            def exists(self, word):
                """
                Check if word exists in trie
                """
                pass
```

Hide Solution

```
In [ ]: class Trie(object):
            def __init__(self):
                self.root = TrieNode()

            def add(self, word):
                """
                Add `word` to trie
                """
                current_node = self.root

                for char in word:
                    current_node = current_node.children[char]

                    current_node.is_word = True

            def exists(self, word):
                """
                Check if word exists in trie
                """
                current_node = self.root

                for char in word:
                    if char not in current_node.children:
                        return False

                    current_node = current_node.children[char]

                return current_node.is_word
```

```
In [ ]: # Add words
        valid_words = ['the', 'a', 'there', 'answer', 'any', 'by', 'bye', 'their']
        word_trie = Trie()
        for valid_word in valid_words:
            word_trie.add(valid_word)

        # Tests
        assert word_trie.exists('the')
        assert word_trie.exists('any')
        assert not word_trie.exists('these')
        assert not word_trie.exists('zzz')
        print('All tests passed!')
```

The Trie data structure is part of the family of Tree data structures. It shines when dealing with sequence data, whether it's characters, words, or network nodes. When working on a problem with sequence data, ask yourself if a Trie is right for the job.