Prefix Tree/Trie

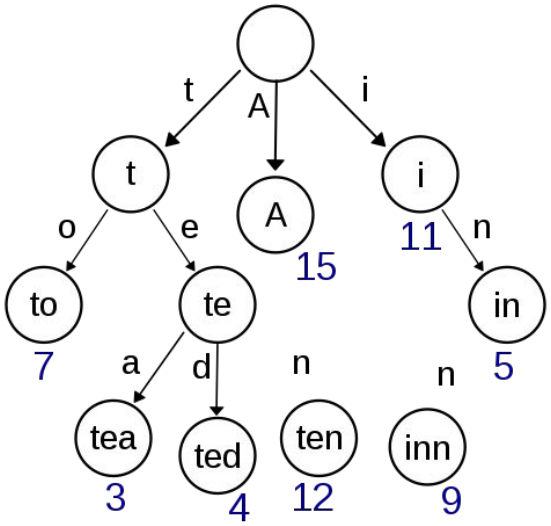
208. Implement Trie
211. Design Add and Search Words Data Structure
212. Word Search II
425. Word Square
642. Design Search Autocomplete System
676. Implement Magic Dictionary
745. Prefix and Suffix Search
1032. Stream of Characters
1233. Remove Sub-Folders from the Filesystem
421. Maximum XOR of Two Numbers in an Array
1707. Maximum XOR With an Element From Array

Yifan Fei, Feb 22nd 2021

Definition

In computer science, a **trie**, also called **digital tree** or **prefix tree**, is a type of search tree, a tree data structure used for locating specific keys from within a set.

A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn". Each complete English word has an arbitrary integer value associated with it.



208. Implement Trie

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

class Trie: #  prefix tree(字典🎄)
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = TrieNode()

    def insert(self, word):
        """
        Inserts a word into the trie.
        :type word: str
        :rtype: void
        """
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch] # move on to the nxt level
        node.word = True # loop to the end, store the word as true
```

```python
    def search(self, word):
        """
        Returns if the word is in the trie.
        :type word: str
        :rtype: bool
        """
        node = self.root
        for ch in word:
            if ch not in node.children:
                return False # go down the tree
            node = node.children[ch]
        return node.word

    def startsWith(self, prefix):
        """
        Returns if there is any word in the trie that starts with the gi
        :type prefix: str
        :rtype: bool
        """
        node = self.root
        for ch in prefix:
            if ch not in node.children:
                return False # go down the tree
            node = node.children[ch]
        return True # loop to the end and all good!
```

211. Design Add and Search Words Data Structure

```python
class WordDictionary:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        node = self.root
        for ch in word:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch] # move on to the nxt level
        node.word = True # loop to the end, store the word as true

    def search(self, word: str) -> bool:
        def dfs(node, i, word):
            if i == len(word): # base case, word till end(empty)
                return node.word
            if word[i] == '.':
                for c in node.children:
                    if dfs(node.children[c], i + 1, word):
                        return True
                return False
            else:
                if word[i] not in node.children:
                    return False
                else:
                    return dfs(node.children[word[i]], i + 1, word)
        return dfs(self.root, 0, word)
```

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the `WordDictionary` class:

- `WordDictionary()` Initializes the object.
- `void addWord(word)` Adds `word` to the data structure, it can be matched later.
- `bool search(word)` Returns `true` if there is any string in the data structure that matches `word` or `false` otherwise. `word` may contain dots `'.'` where dots can be matched with any letter.

Input

```
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]

[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
```

Output

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

212. Word Search II

```python
class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[s
        res, trie, m, n = [], Trie(), len(board), len(board[0])
        for word in words:
            trie.insert(word)
        for i in range(m):
            for j in range(n):
                self.dfs(board, trie.root, i, j, "", res, m, n)
        return res

    def dfs(self, board, node, x, y, path, res, m, n):
        if node.word:
            res.append(path)
            node.word = False # find a word and marked as visited
        if 0 <= x < m and 0 <= y < n:
            tmp = board[x][y]
            if tmp not in node.children:
                return
            node = node.children[tmp] # go down
            board[x][y] = '#'
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                self.dfs(board, node, nx, ny, path + tmp, res, m, n)
            board[x][y] = tmp
```

Given an `m x n board` of characters and a list of strings `words`, return *all words on the board*.

- `m == board.length`
- `n == board[i].length`
- `1 <= m, n <= 12`
- `board[i][j]` is a lowercase English letter.
- `1 <= words.length <= 3 * 10^4`
- `1 <= words[i].length <= 10`
- `words[i]` consists of lowercase English letters.
- All the strings of `words` are unique

425. Word Square

Given a set of words (without duplicates), find all word squares you can build from them.

```
Input: ["area","lead","wall","lady","ball"]

wall

area

lead

lady
```

```python
class TrieNode:
    def __init__(self):
        self.children = defaultdict(TrieNode)
        self.isWord = False
        self.words = []

class Trie: #  prefix tree(字典🎄)
    def __init__(self, words):
        self.root = TrieNode()
        for word in words:
            self.insert(word)

    def insert(self, word):
        node = self.root
        node.words.append(word)
        for ch in word:
            node = node.children[ch] # move on to the nxt level
            node.words.append(word)
        node.isWord = True # loop to the end, store the word as true
        # node.string = word

    def allWords(self, prefix):
        # return all possible words with current prefix
        node = self.root
        for ch in prefix:
            if not node.children.get(ch, None):
                return [] # go down the tree
            node = node.children[ch]
        return node.words
```

```python
class Solution:
    def wordSquares(self, words: List[str]) -> List[List[str]]:
        res, k, trie = [], len(words[0]), Trie(words)

        def dfs(row, matrix):
            # row means last row index + 1
            if row == k:
                res.append(matrix)
                return
            prefix = ''.join(r[row] for r in matrix)
            for word in trie.allWords(prefix):
                dfs(row + 1, matrix + [word]) # pass by val

        def dfs2(row, matrix):
            # row means last row index + 1
            if row == k:
                res.append(matrix[:]) # matrix is passed by ref, need a shallow-copy
                return
            prefix = ''.join(r[row] for r in matrix)
            for word in trie.allWords(prefix):
                matrix.append(word)
                dfs(row + 1, matrix) # pass by ref
                matrix.pop()

        dfs(0, [])
        return res
```

642. Design Search Autocomplete System

```python
class AutocompleteSystem1:
    '''
    storing all the words at each node, it increases space but makes the solution around 5 times faster.
    '''

    def __init__(self, sentences: List[str], times: List[int]):
        self.trie = TrieNode()
        self.cache_count = defaultdict(int)
        self.keyword = ""
        for i, sen in enumerate(sentences):
            self._insert(sen, self.trie)
            self.cache_count[sen] = times[i]

    def _insert(self, word, trie):
        for ch in word:
            if ch not in trie.children:
                trie.children[ch] = TrieNode()
            trie = trie.children[ch]
            trie.words.append(word)
        return True

    def _search(self, word):
        trie = self.trie
        for ch in word:
            if ch not in trie.children:
                return []
            trie = trie.children[ch]
        return trie.words

    def input(self, c: str) -> List[str]:
        if c != '#':
            self.keyword += c
            sens = self._search(self.keyword)
            res = []
            for sen in sens:
                res.append((self.cache_count[sen], sen))
            res = list(set(res))
            return [sen for cnt, sen in sorted(res, key=lambda x:(-x[0], x[1]))[:3]]
        else:
            self.cache_count[self.keyword] += 1
            self._insert(self.keyword, self.trie)
            self.keyword = ""
        return []
```

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character '#'). For each character they type except '#', you need to return the top 3 historical hot sentences that have prefix the same as the part of sentence already typed. Here are the specific rules:

Operation: AutocompleteSystem(["i love you", "island","ironman", "i love leetcode"], [5,3,2,2])

The system have already tracked down the following sentences and their corresponding times:

"i love you" : 5 times

"island" : 3 times

"ironman" : 2 times

"i love leetcode" : 2 times

676. Implement Magic Dictionary

```python
class MagicDictionary:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = Trie()

    def buildDict(self, dictionary: List[str]) -> None:
        for word in dictionary:
            self.trie.insert(word)

    def search(self, searchWord: str) -> bool:
        self.change_once_flag = False
        return self.dfs(self.trie.root, 0, searchWord)

    def dfs(self, node, pos, word):
        if pos == len(word): return node.word and self.change_once_flag
        if self.change_once_flag:
            if word[pos] in node.children:
                return self.dfs(node.children[word[pos]], pos+1, word)
            else:
                return False
        else:
            for c in node.children: # try to change in this level
                self.change_once_flag = (c != word[pos])
                if self.dfs(node.children[c], pos+1, word):
                    return True
            return False
```

`MagicDictionary()` Initializes the object.

`void buildDict(String[] dictionary)` Sets the data structure with an array of distinct strings `dictionary`.

`bool search(String searchWord)` Returns `true` if you can change exactly one character in `searchWord` to match any string in the data structure, otherwise returns `false`.

745. Prefix and Suffix Search

```python
class TrieNode():
    def __init__(self):
        self.children = {}
        self.weights = []

# create two Tries, one for prefix search,
# another one for suffix search
# then find the maximal common weight
class Trie():
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word, i):
        node = self.root
        node.weights.append(i)
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            node.weights.append(i)

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return []
            node = node.children[char]
        return node.weights
```

- `WordFilter(string[] words)` Initializes the object with the `words` in the dictionary.
- `f(string prefix, string suffix)` Returns *the index of the word in the dictionary* which has the prefix `prefix` and the suffix `suffix`. If there is more than one valid index, return the largest of them. If there is no such word in the dictionary, return `-1`.

```python
class WordFilter1:
    def __init__(self, words: List[str]):
        self.prefix, self.suffix = Trie(), Trie()
        i, n = 0, len(words)
        while i < n:
            w = words[i]
            self.prefix.insert(w, i)
            self.suffix.insert(w[::-1], i)
            i += 1

    def f(self, prefix: str, suffix: str) -> int:
        pre = self.prefix.search(prefix)
        suf = self.suffix.search(suffix[::-1])
        i, j = len(pre) - 1, len(suf) - 1
        while i >= 0 and j >= 0:
            if pre[i] == suf[j]:
                return pre[i]
            elif pre[i] < suf[j]:
                j -= 1
            else:
                i -= 1
        return -1
```

1032. Stream of Characters

```python
class StreamChecker:

    def __init__(self, words: List[str]):
        self.letters = []
        self.trie = Trie()
        for word in words:
            self.trie.insert(word[::-1])
            # for example, stream is a, b, c, d, and word = 'cd', reverse both

    def query(self, letter: str) -> bool:
        self.letters.append(letter)
        i = len(self.letters) - 1
        node = self.trie.root
        while i >= 0: # reverse search
            if node.isWord:
                return True
            if self.letters[i] not in node.children:
                return False
            node = node.children[self.letters[i]]
            i -= 1
        return node.isWord
```

`StreamChecker(words)`: Constructor, init the data structure with the given words.

`query(letter)`: returns true if and only if for some `k >= 1`, **the last `k` characters queried** (in order from oldest to newest, including this letter just queried) spell one of the words in the given list.

```
StreamChecker streamChecker = new StreamChecker(["cd","f","kl"]);

// init the dictionary.

streamChecker.query('a');          // return false

streamChecker.query('b');          // return false

streamChecker.query('c');          // return false

streamChecker.query('d');          // return true, because 'cd' is in the wordlist
```

1233. Remove Sub-Folders from the Filesystem

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

    def insert(self, word):
        node = self.root
        for w in word:
            if w not in node.children:
                node.children[w] = TrieNode()
            node = node.children[w]
        node.word = True

    def find(self): # find prefix
        res = []
        def dfs(dirs, node):
            if node.word: # current node is leaf, append prev dirs
                res.append('/' + '/'.join(dirs))
                # print(dirs, res)
                return
            for nxt_node in node.children:
                dfs(dirs + [nxt_node], node.children[nxt_node])
        dfs([], self.root)
        return res

class Solution:
    def removeSubfolders(self, folder: List[str]) -> List[str]:
        trie = Trie()
        for f in folder:
            f = f.split('/')[1:] # split into list, remove empty
            trie.insert(f)
        return trie.find()
```

Input: folder = ["/a","/a/b","/c/d","/c/d/e","/c/f"]

Output: ["/a","/c/d","/c/f"]

421. Maximum XOR of Two Numbers in an Array

```python
class Solution0:
    def findMaximumXOR(self, nums: List[int]) -> int:
        trie, res = Trie(), 0
        for num in nums:
            trie.insert(num)

        for num in nums:
            node = trie.root
            res = max(res, trie.query(num))
        return res

class Trie:
    def __init__(self):
        self.root = {}

    def insert(self, num):
        p = self.root
        for i in range(31, -1, -1):
            cur = (num >> i) & 1
            if cur not in p:
                p[cur] = {}
            p = p[cur]

    def query(self, num):
        if not self.root:
            return -1
        p, res = self.root, 0
        for i in range(31, -1, -1):
            cur = (num >> i) & 1
            if 1 - cur in p:
                p = p[1 - cur]
                res |= (1 << i)
            else:
                p = p[cur]
        return res
```

Given an integer array `nums`, return *the maximum result of* `nums[i] XOR nums[j]`, where $0 \leqslant i \leqslant j < n$.

Follow up: Could you do this in `O(n)` runtime?

```python
class Solution1:
    def findMaximumXOR(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        https://www.youtube.com/watch?v=ZHtZfkAcPKc&t=54s
        """
        # Build the answer bit by bit from left to right
        # a ^ b = res => a ^ res = b
        res = 0
        for i in range(31, -1, -1):
            res <<= 1
            prefixes = {num >> i for num in nums} # get prefix(bits) for each num
            nxt = res + 1 # get next biggest mask based on last round res
            if any(nxt ^ p in prefixes for p in prefixes):
                res = nxt
            # res += any(nxt ^ p in prefixes for p in prefixes)
            # print(bin(res)[2:], [bin(pre)[2:] for pre in prefixes])
        return res
```

1707. Maximum XOR With an Element From Array

```python
class Solution:
    def maximizeXor(self, nums: List[int], queries: List[List[int]]) -> List[int]:
        nums.sort()
        trie, n = Trie(), len(queries)
        queries_sort = sorted(enumerate(queries), key=lambda x: x[1][1]) # sorted by m

        res, j = [-1] * n, 0
        for i, (x, m) in queries_sort:
            while j < len(nums) and nums[j] < m:
                trie.insert(nums[j])
                j += 1
            res[i] = trie.query(x)
        return res


class Trie:
    def __init__(self):
        self.root = {}

    def insert(self, num):
        node = self.root
        for i in range(31, -1, -1):
            bit = (num >> i) & 1 # get i-th bit
            if bit not in node:
                node[bit] = {}
            node = node[bit]

    def query(self, num):
        # O(32)
        if not self.root: # no node
            return -1
        node, res = self.root, 0
        for i in range(31, -1, -1):
            bit = (num >> i) & 1 # get i-th bit
            if 1 - bit in node: # greedy, chose complement
                # find a 1/0, go to node.zero/node.one
                node = node[1 - bit]
                res |= (1 << i)
            else: # only one path, follow it
                node = node[bit]
        return res
```

You are given an array `nums` consisting of non-negative integers. You are also given a `queries` array, where `queries[i] = [xi, mi]`.

The answer to the `ith` query is the maximum bitwise `XOR` value of `xi` and any element of `nums` that does not exceed `mi`. In other words, the answer is `max(nums[j] XOR xi)` for all `j` such that `nums[j] <= mi`. If all elements in `nums` are larger than `mi`, then the answer is `-1`.

Return *an integer array* `answer` *where* `answer.length == queries.length` *and* `answer[i]` *is the answer to the* `ith` *query.*

`Input: nums = [0,1,2,3,4], queries = [[3,1],[1,3],[5,6]]`

`Output: [3,3,7]`

`Explanation:`

`1) 0 and 1 are the only two integers not greater than 1. 0 XOR 3 = 3 and 1 XOR 3 = 2. The larger of the two is 3.`

`2) 1 XOR 2 = 3.`