Tries are a tree-type of structure where each node represents a single character of a given string. This makes it a very efficient data structure to retrieve data, for example to show word suggestions as we type along on our smartphone keyboards. Unlike binary trees, a trie can have more than two children, generally equal to the number of letters in the alphabet. Also, each node consists of an "end_of_word" variable, which tells us whether it marks the end of a word or not

Doing this task using a list or a balanced binary search tree costs O(nm) and O(mlog N) respectively, where "m" is the length of the string being searched for. However, the same operation can be done in O(m) time using tries.

Here we will implement two of its major operations - insert and search -, which are both of O(m) time complexity.

In [1]:

```
# We start by defining a Trie_node class containing 26 children each initi

class TrieNode():
    def __init__(self):
        self.children = [None]*26
        self.is_end_of_word = False
```

In [2]:

```
# Then, we create the Trie class itself, which has a constructor that init

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

# To calculate the numerical index of each character in the range of 0-25,
    def _character_index(self, char):
        if char.isupper():
            return ord(char) - ord('A')
        else:
            return ord(char) - ord('a')


    def insert(self, string):
        pointer = self.root
        for character in string:
            index = self._character_index(character)
            if not pointer.children[index]:
                pointer.children[index] = TrieNode()
            pointer = pointer.children[index]
        pointer.is_end_of_word = True
        return

# Above, we created the insert function. It starts by building a pointer wi
# Then, for every character in the word to be inserted, the method will che
# If it does, the method will update the pointer to that child of the curre
# Otherwise, it will initialize a new node at the index of the character th
# After that, the method will update the pointer to refer to this newly cre
# Once it reach the end of the word, the method will set the "is_end_of_wor


    def search(self, string):
        pointer = self.root
        for character in string:
            index = self._character_index(character)
            if not pointer.children[index]:
                return False
            pointer = pointer.children[index]
        return pointer and pointer.is_end_of_word

# Finally, for the search method, we will follow the exact same approach.
# The only difference is that, this time, instead of creating a new TrieNod
# If after the loop terminates, "is_end_of_word" equals True and the node
```

Once finished, we initialize a new trie using the newly created Trie class:

In [3]:

```
new_trie = Trie()

# Inserting data into the newly created trie
new_trie.insert('Data')
new_trie.insert("Structures")
new_trie.insert("and")
new_trie.insert("Algorithms")
```

In [4]:
```
# Showing its place in memory
print(new_trie)
```

```
<__main__.Trie object at 0x7fd7b05f44f0>
```

In [6]:
```
# Looking for data using the search method
print(new_trie.search("and"))
```

```
True
```

In [7]:
```
print(new_trie.search("Data"))
```

```
True
```

In [8]:
```
# Returns false if data is not found
print(new_trie.search("woohoo"))
```

```
False
```

In [10]:
```
print(new_trie.search("Structures"))
```

```
True
```