TRIE

Dictionary, Autocompletion

DSA Project

Anahita Heydari

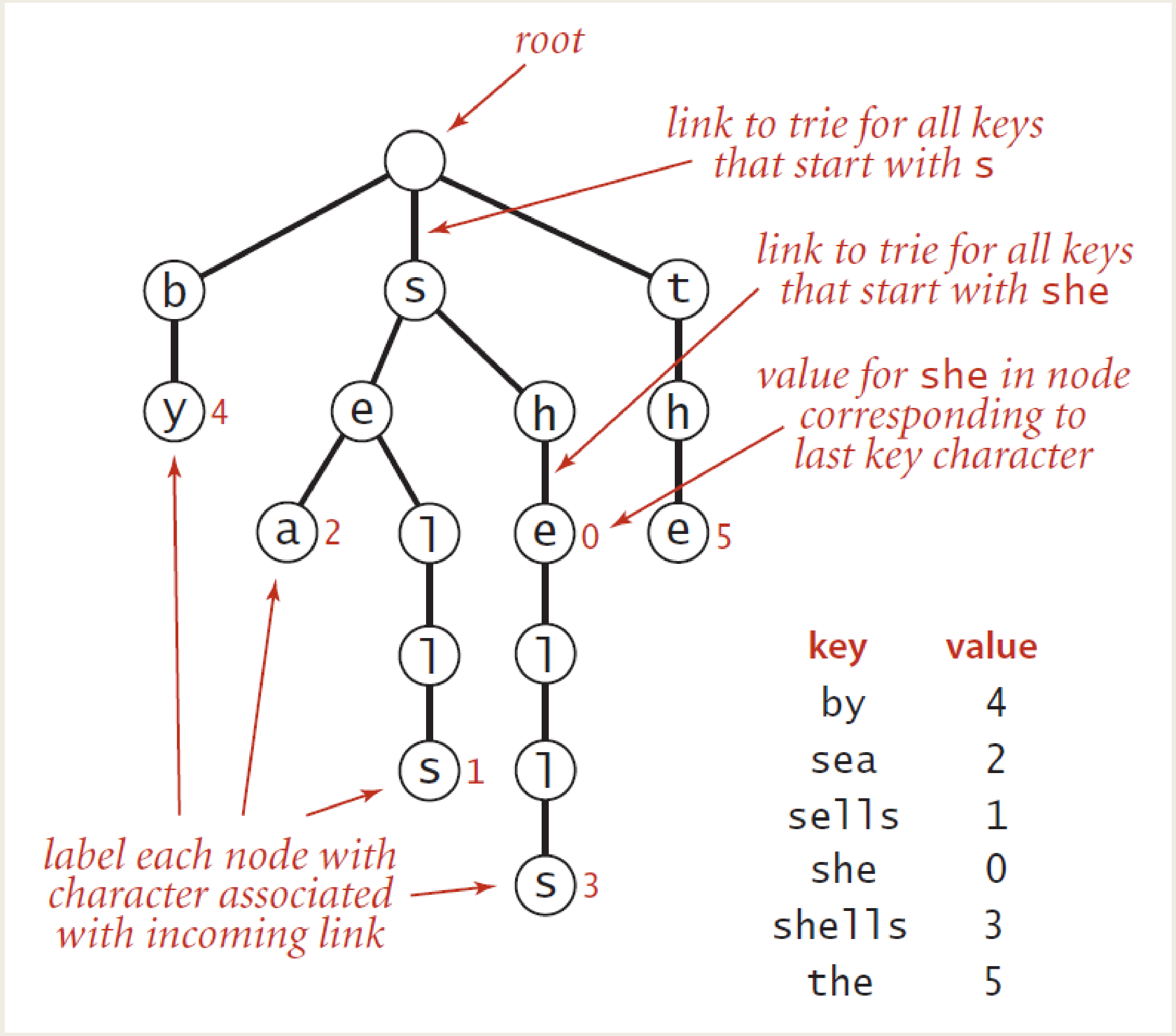
2022

History and Basic Definition

- Trie (also known as the digital tree or prefix tree) is a sorted and efficient tree-based special data structure that is used to store and retrieve keys in a dataset of strings. The basic ideology behind this is retrieving information. It is based on the prefix of a string. It can be visualized as a graph consisting of nodes and edges. Another name for Trie is the digital tree. Each node of a trie can have as many as 26 pointers/references. These 26 pointers represent the 26 characters of the English language.
- **Basic Operations of Trie:**
  - **Insertion**
  - **Deletion**
  - **Searching**
- The idea of a trie for representing a set of strings was first abstractly described by Axel Thue in 1912. Tries were first described in a computer context by René de la Briandais in 1959.
- The idea was independently described in 1960 by Edward Fredkin, who coined the term *trie*, pronouncing it /ˈtriː/ (as "tree"), after the middle syllable of *retrieval*. However, other authors pronounce it /ˈtraɪ/ (as "try"), in an attempt to distinguish it verbally from "tree".

Basic properties

As with search trees, tries are data structures composed of *nodes* that contain *links* that are either *null* or references to other nodes. Each node is pointed to by just one other node, which is called its *parent* (except for one node, the *root*, which has no nodes pointing to it), and each node has *R* links, where *R* is the alphabet size.


root
link to trie for all keys that start with s
link to trie for all keys that start with she
value for she in node corresponding to last key character
label each node with character associated with incoming link
b
s
t
y 4
e
h
h
a 2
l
e 0
e 5
l
l
s 1
l
s 3

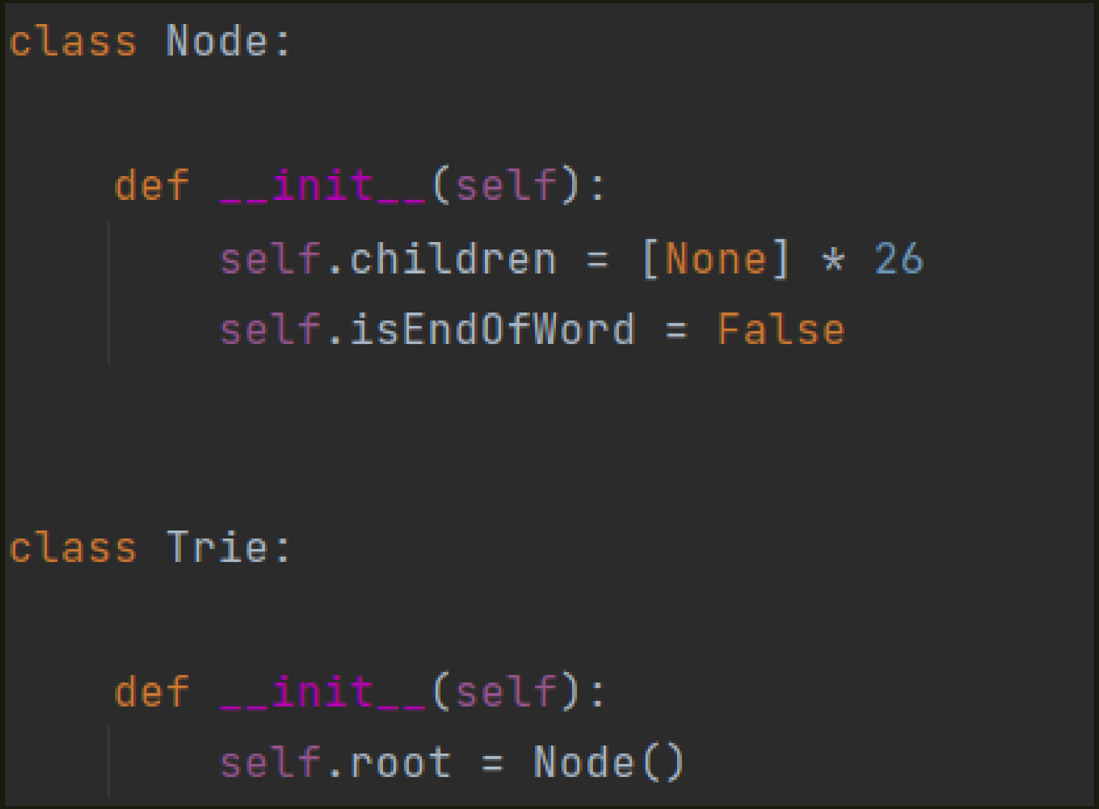
key	value
by	4
sea	2
sells	1
she	0
shells	3
the	5

Anatomy of a Trie

Often, tries have a substantial number of null links, so when we draw a trie, we typically omit null links. Although links point to nodes, we can view each link as pointing to a trie, the trie whose root is the referenced node. Each link corresponds to a character value since each link points to exactly one node, we label each node with the character value corresponding to the link that points to it (except for the root, which has no link pointing to it).

Each node also has a corresponding *value*, which may be null or the value associated with one of the string keys in the symbol table. Specifically, we store the value associated with each key in the node corresponding to its last character. It is very important to bear in mind the following fact: *nodes with null values exist to facilitate search in the trie and do not correspond to keys.*

PYTHON IMPLEMENTATION OF TRIE NODE



```python
class Node:

    def __init__(self):
        self.children = [None] * 26
        self.isEndOfWord = False


class Trie:

    def __init__(self):
        self.root = Node()
```

Insertion into a Trie

As with binary search trees, we insert by first doing a search: in a trie that means using the characters of the key to guide us down the trie until reaching the last character of the key or a null link. At this point, one of the following two conditions holds:

- We encountered a null link before reaching the last character of the key. In this case, there is no trie node corresponding to the last character in the key, so we need to create nodes for each of the characters in the key not yet encountered

and set the value in the last one to the value to be associated with the key.

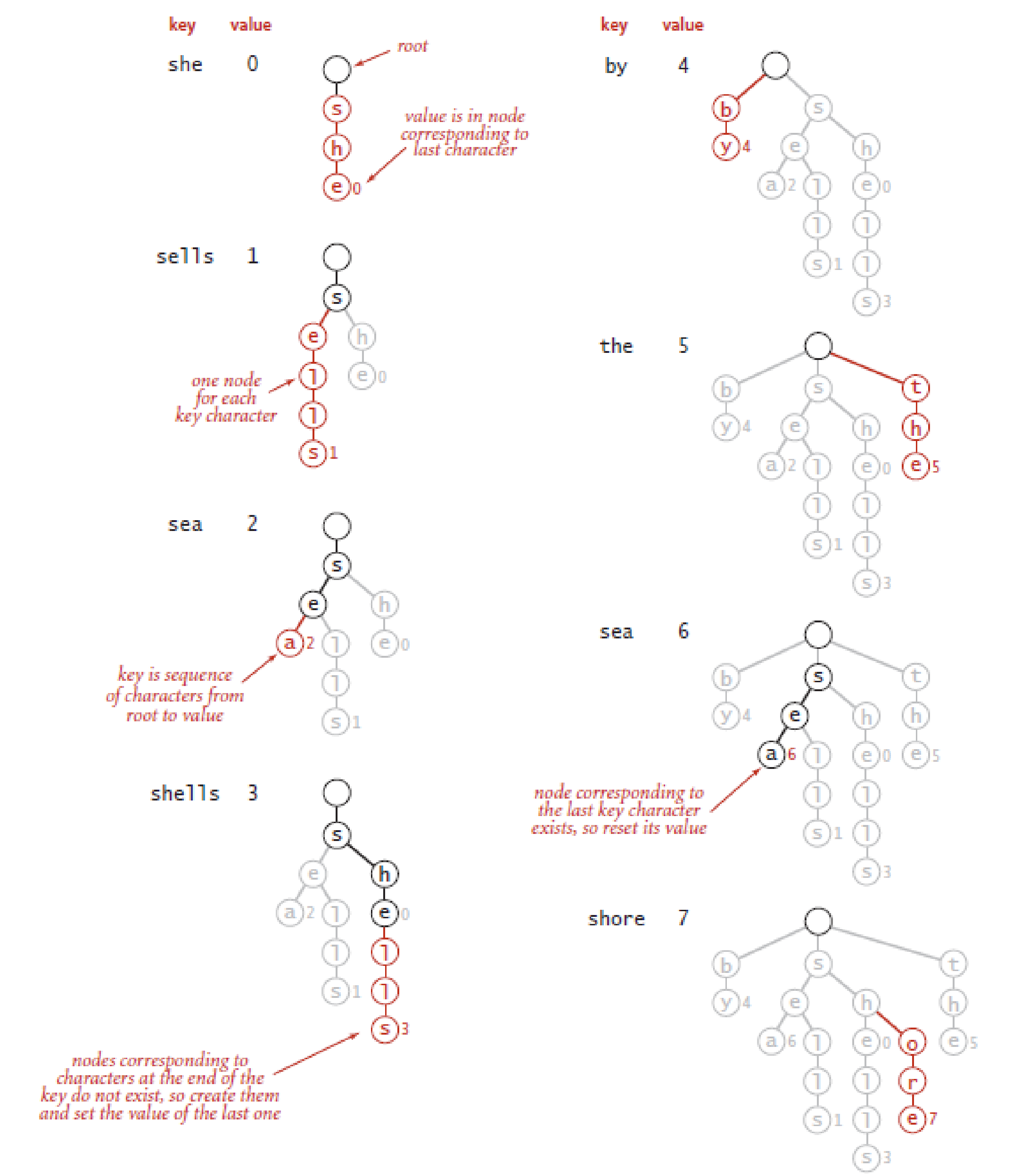
- We encountered the last character of the key before reaching a null link. In this case, we set that node's value to the value to be associated with the key (whether or not that value is null), as usual with our associative array convention.

In all cases, we examine or create a node in the trie for each key character.

The construction of a trie with the input

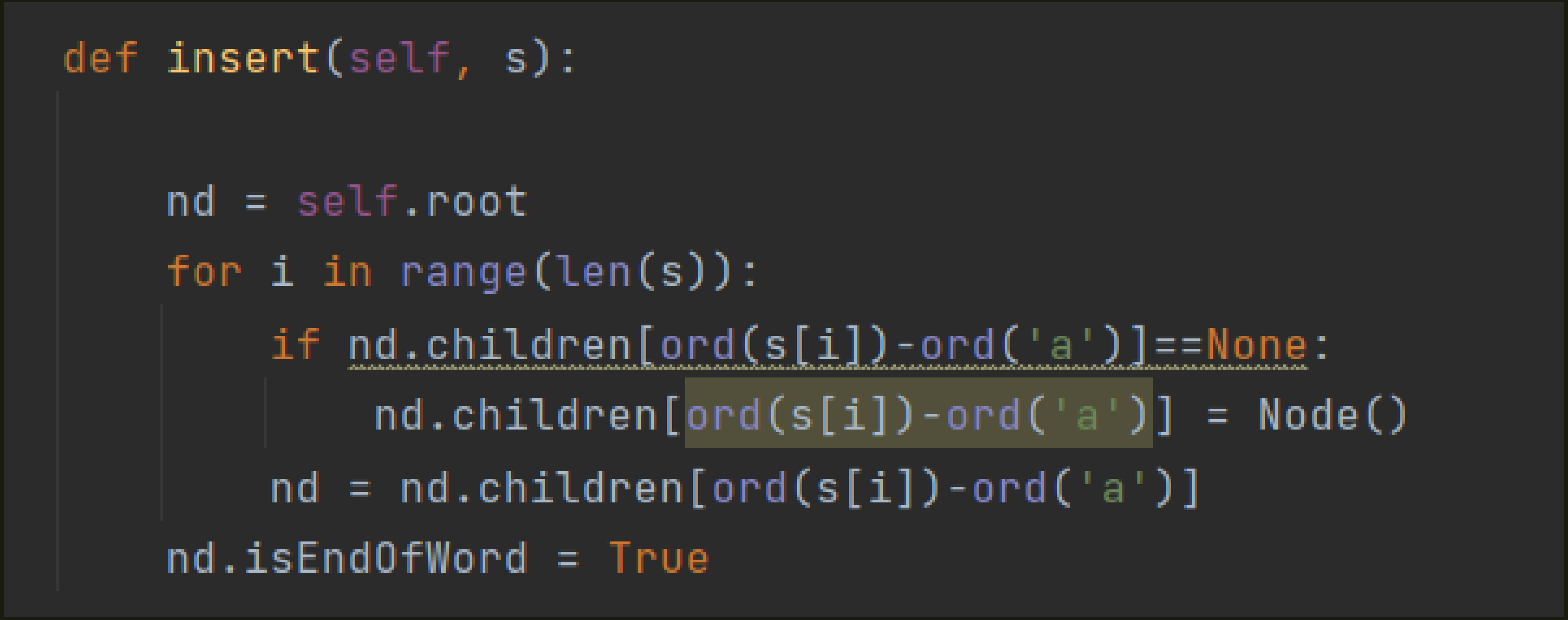
she
sells
sea
shells
by
the
sea
shore

is shown on the right.


key
value
she 0
root
value is in node corresponding to last character
sells 1
one node for each key character
sea 2
key is sequence of characters from root to value
shells 3
nodes corresponding to characters at the end of the key do not exist, so create them and set the value of the last one
key
value
by 4
the 5
sea 6
node corresponding to the last key character exists, so reset its value
shore 7

Trie construction trace for standard indexing client

PYTHON IMPLEMENTATION OF TRIE INSERT



```
def insert(self, s):


    nd = self.root
    for i in range(len(s)):
        if nd.children[ord(s[i])-ord('a')]==None:
            nd.children[ord(s[i])-ord('a')] = Node()
        nd = nd.children[ord(s[i])-ord('a')]
    nd.isEndOfWord = True
```

Time Complexity (n is the length of the input): O(n) every case

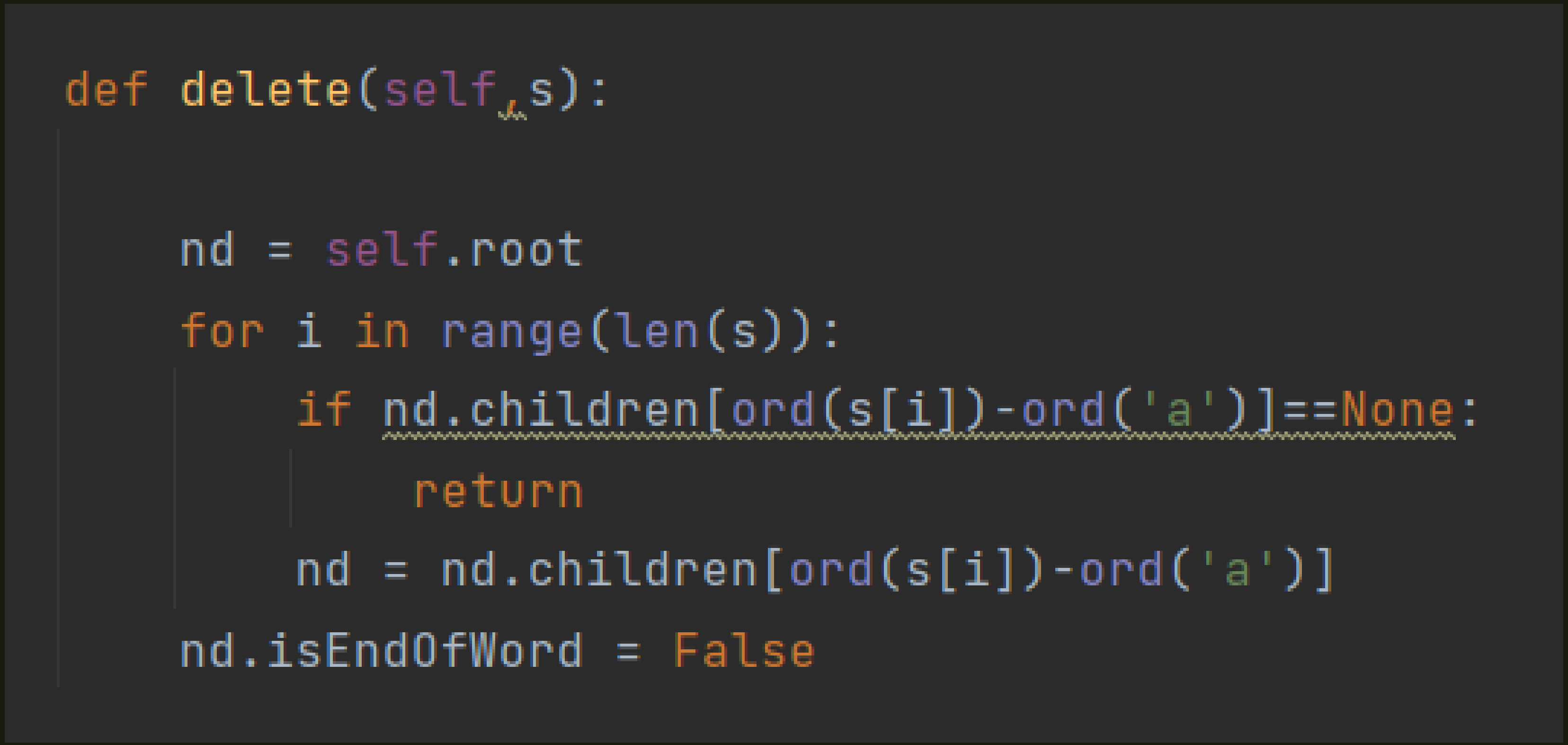
Delete

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

PYTHON IMPLEMENTATION OF TRIE DELETE

(THE EASIEST WAY)



```python
def delete(self,s):

    nd = self.root
    for i in range(len(s)):
        if nd.children[ord(s[i])-ord('a')]==None:
            return
        nd = nd.children[ord(s[i])-ord('a')]
    nd.isEndOfWord = False
```

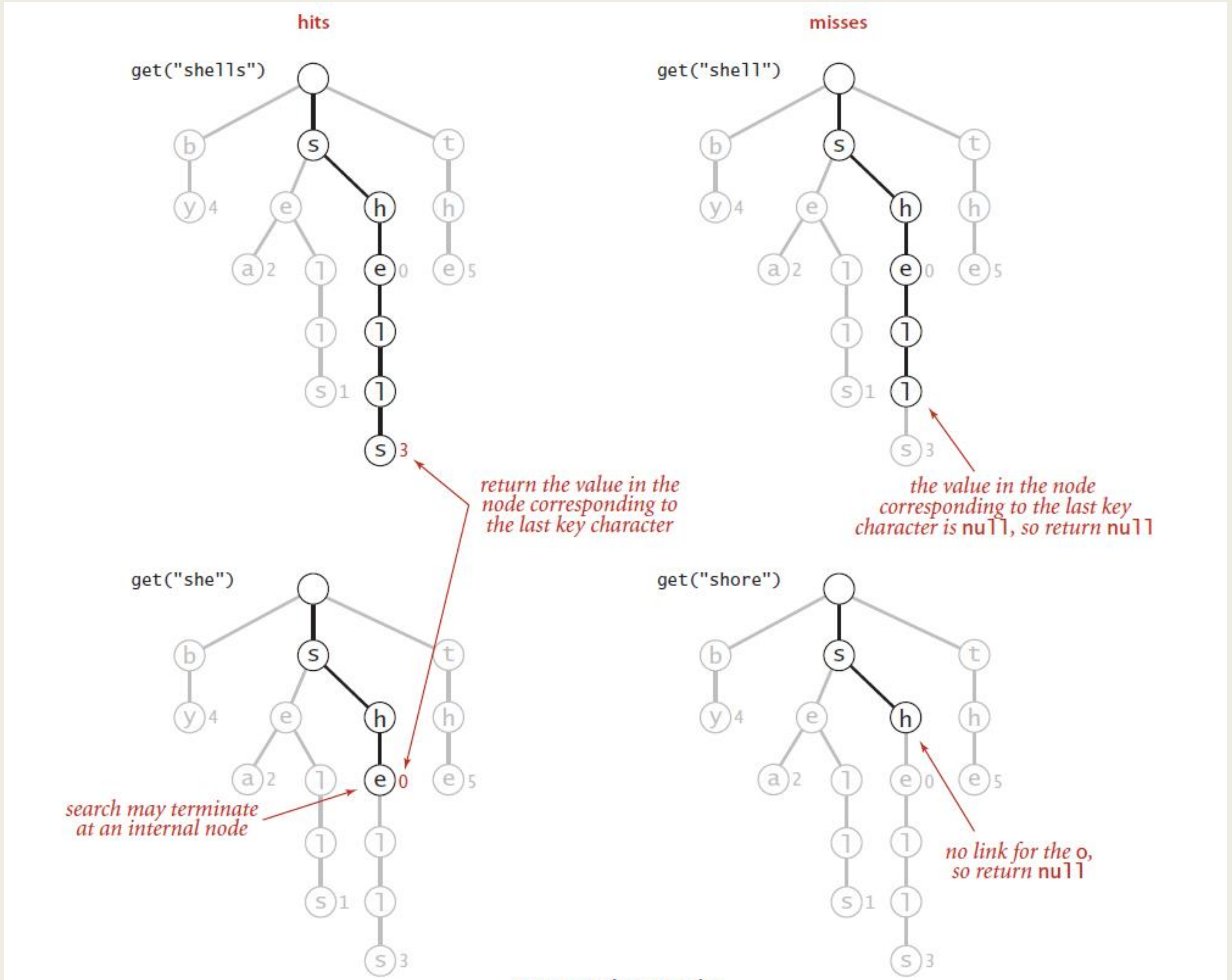
Time Complexity (n is the length of the input): bc: O(1) wc: O(n)

Search in a Trie

Finding the value associated with a given string key in a trie is a simple process, guided by the characters in the search key. Each node in the trie has a link corresponding to each possible string character. We start at the root, then follow the link associated with the first character in the key; from that node we follow the link associated with the second character in the key; from that node we follow the link associated with the third character in the key and so forth, until reaching the last character of the key or a null link. At this point, one of the following three conditions holds (refer to the figure above for examples):

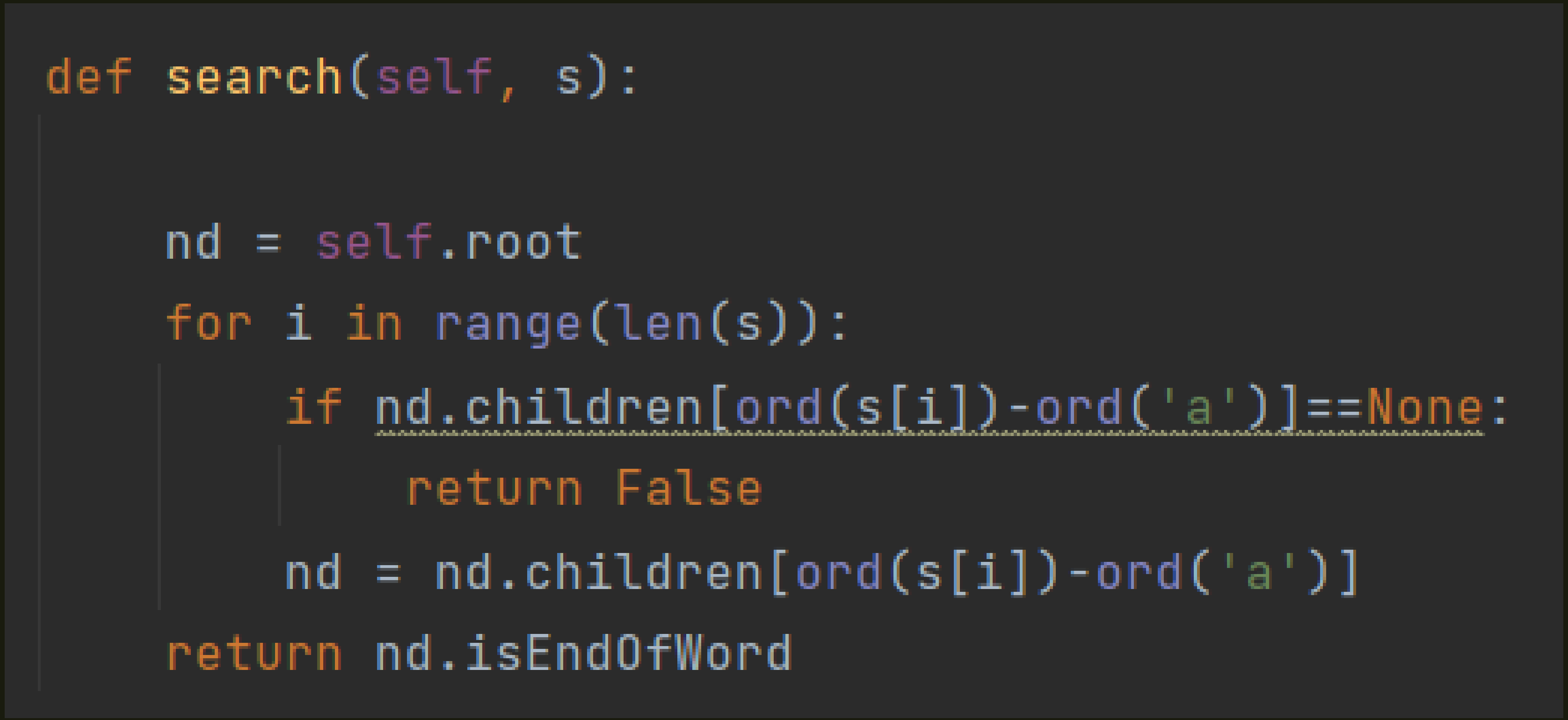
- The value at the node corresponding to the last character in the key is not null (as in the searches for shells and she depicted at left above). This result is a search hit—the value associated with the key is the value in the node corresponding to its last character.
- The value in the node corresponding to the last character in the key is null (as in the search for shell depicted at top right above). This result is a search miss: the key is not in the table.
- The search terminated with a null link (as in the search for shore depicted at bottom right above). This result is also a search miss.

In all cases, the search is accomplished just by examining nodes along a path from the root to another node in the trie.


hits
misses
get("shells")
get("shell")
get("she")
get("shore")
return the value in the node corresponding to the last key character
the value in the node corresponding to the last key character is null, so return null
search may terminate at an internal node
no link for the o, so return null

Trie search examples

PYTHON IMPLEMENTATION OF TRIE SEARCH



```python
def search(self, s):


    nd = self.root
    for i in range(len(s)):
        if nd.children[ord(s[i])-ord('a')]==None:
            return False
        nd = nd.children[ord(s[i])-ord('a')]
    return nd.isEndOfWord
```

Time Complexity (n is the length of the input): bc: O(1) wc: O(n)

Applications of Trie data structure:

It has a wide variety of applications in data compression, computational biology, longest prefix matching algorithm used for routing tables for IP addresses, implementation of the dictionary, pattern searching, storing/querying XML documents, etc.

Real-time applications of Trie data structure:

1. **Browser History:**

Web browsers keep track of the history of websites visited by the user So when the prefix of a previously visited URL is written in the address bar the user would be given suggestions of the website to visit.

Trie is used by storing the number of visits to a website as the key value and organizing this history on the Trie data structure.

2. **AutoComplete:**

It is one of the most important applications of trie data structure. This feature speeds up interactions between a user and the application and greatly enhances the user experience. Auto Complete feature is used by web browsers, email, search engines, code editors, command-line interpreters(CLI), and word processors.

Trie provides the alphabetical ordering of data by keys. Trie is used because it is the fastest for auto-complete suggestions, even in the worst case, it is O(n) (where n is the string length) times faster than the alternate imperfect hash table algorithm and also overcomes the problem of key collisions in imperfect hash tables.

3. Spell Checkers/Auto-correct:

It is a 3-step process that includes :

1. Checking for the word in the data dictionary.
2. Generating potential suggestions.
3. Sorting the suggestions with higher priority on top.

Trie stores the data dictionary and makes it easier to build an algorithm for searching the word from the dictionary and provides the list of valid words for the suggestion.

4. Longest Prefix Matching Algorithm (Maximum Prefix Length Match):

This algorithm is used in networking by the routing devices in IP networking. Optimization of network routes requires contiguous masking that bound the complexity of lookup a time to O(n), where n is the length of the URL address in bits.

To speed up the lookup process, Multiple Bit trie schemes were developed that perform the lookups of multiple bits faster.

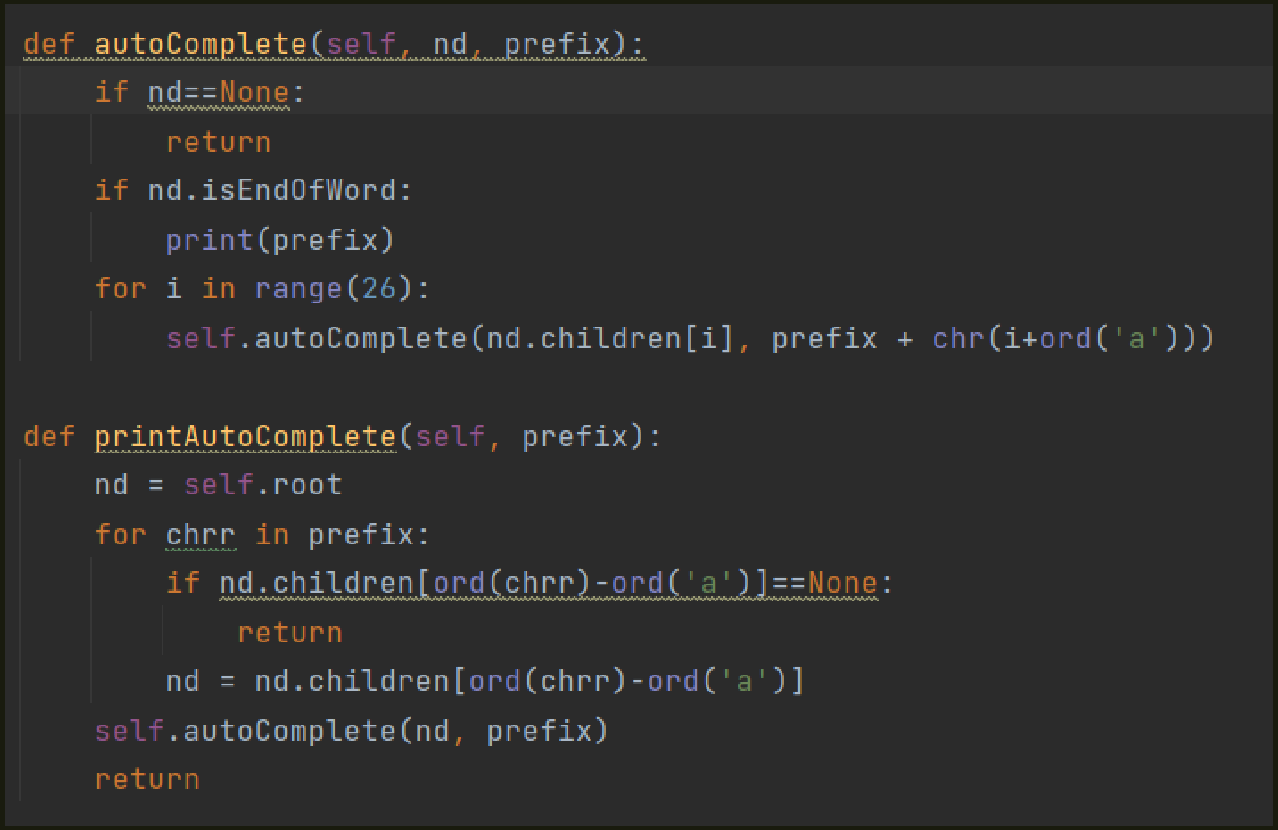
Auto-complete feature using Trie

We are given a Trie with a set of strings stored in it. Now the user types in a prefix of his search query, we need to give him all recommendations to auto-complete his query based on the strings stored in the Trie. We assume that the Trie stores past searches by the users. For example if the Trie store {“abc”, “abcd”, “aa”, “abbbaba”} and the User types in “ab” then he must be shown {“abc”, “abcd”, “abbbaba”}.

Given a query prefix, we search for all words having this query.

1. Search for the given query using the standard Trie search algorithm.
2. If the query prefix itself is not present, return -1 to indicate the same.
3. If the query is present and is the end of a word in Trie, print query. This can quickly be checked by seeing if the last matching node has **isEndWord** flag set. We use this flag in Trie to mark the end of word nodes for purpose of searching.
4. If the last matching node of the query has no children, return.
5. Else recursively print all nodes under a subtree of last matching node.

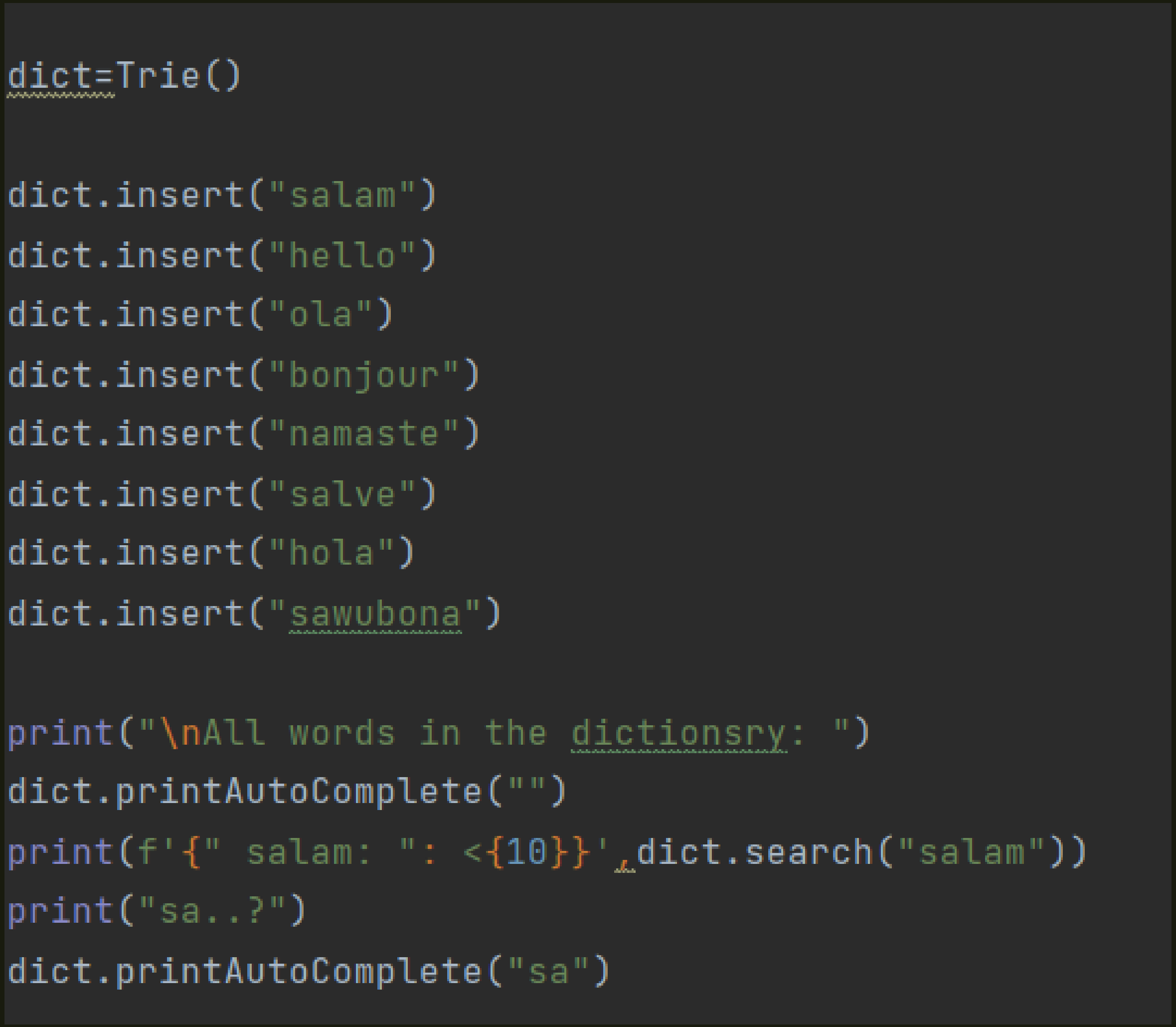
PYTHON IMPLEMENTATION OF AUTO-COMPLETE



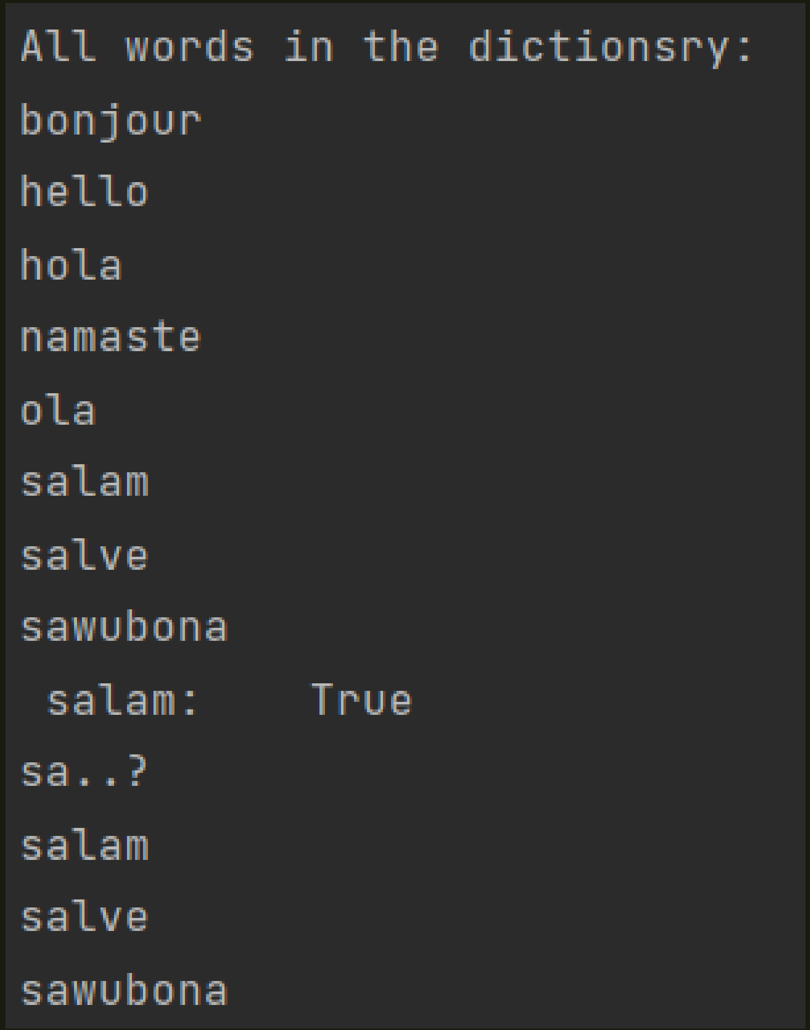
```python
def autoComplete(self, nd, prefix):
    if nd==None:
        return
    if nd.isEndOfWord:
        print(prefix)
    for i in range(26):
        self.autoComplete(nd.children[i], prefix + chr(i+ord('a')))

def printAutoComplete(self, prefix):
    nd = self.root
    for chrr in prefix:
        if nd.children[ord(chrr)-ord('a')]==None:
            return
        nd = nd.children[ord(chrr)-ord('a')]
    self.autoComplete(nd, prefix)
    return
```

Time Complexity (M is the total number of children): bc: O(1) wc: O(M)

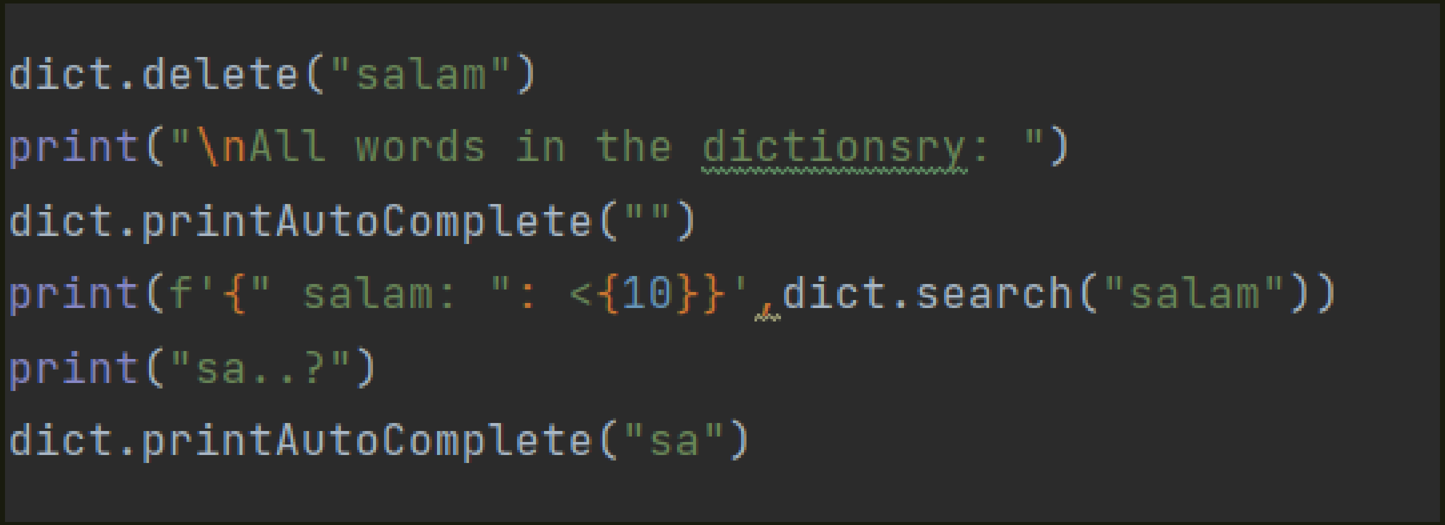
EXAMPLE



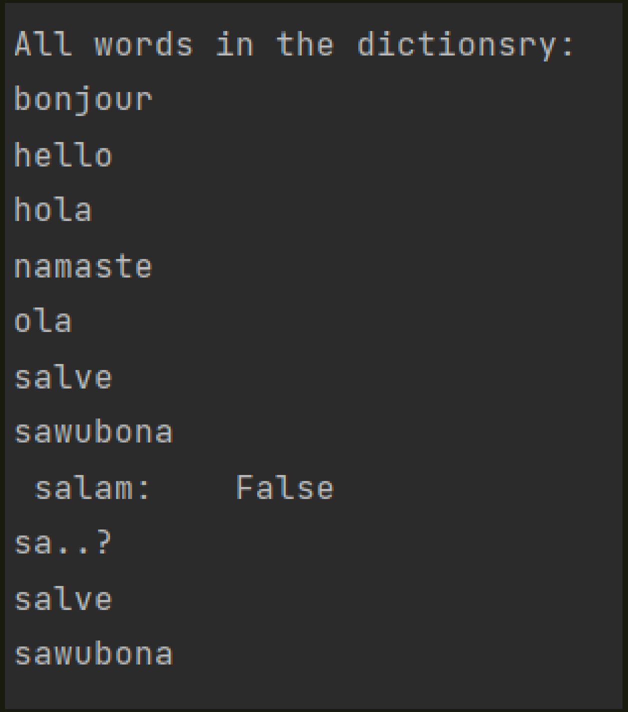
```python
dict=Trie()

dict.insert("salam")
dict.insert("hello")
dict.insert("ola")
dict.insert("bonjour")
dict.insert("namaste")
dict.insert("salve")
dict.insert("hola")
dict.insert("sawubona")

print("\nAll words in the dictionsry: ")
dict.printAutoComplete("")
print(f'{" salam: ": <{10}}',dict.search("salam"))
print("sa..?")
dict.printAutoComplete("sa")
```



```
All words in the dictionsry:
bonjour
hello
hola
namaste
ola
salam
salve
sawubona
 salam:    True
sa..?
salam
salve
sawubona
```



```
dict.delete("salam")
print("\nAll words in the dictionsry: ")
dict.printAutoComplete("")
print(f'{" salam: ": <{10}}',dict.search("salam"))
print("sa..?")
dict.printAutoComplete("sa")
```



```
All words in the dictionsry:
bonjour
hello
hola
namaste
ola
salve
sawubona
 salam:    False
sa..?
salve
sawubona
```

Advantages of Trie data structure:

- Trie allows us to input and finds strings in O(l) time, where **l** is the length of a single word. It is faster as compared to both hash tables and binary search trees.
- It provides alphabetical filtering of entries by the key of the node and hence makes it easier to print all words in alphabetical order.
- Trie takes less space when compared to BST because the keys are not explicitly saved instead each key requires just an amortized fixed amount of space to be stored.
- Prefix search/Longest prefix matching can be efficiently done with the help of trie data structure.
- Since trie doesn’t need any hash function for its implementation so they are generally faster than hash tables for small keys like integers and pointers.
- Tries support ordered iteration whereas iteration in a hash table will result in pseudorandom order given by the hash function which is usually more cumbersome.
- Deletion is also a straightforward algorithm with O(l) as its time complexity, where **l** is the length of the word to be deleted.

Disadvantages of Trie data structure:

- The main disadvantage of the trie is that it takes a lot of memory to store all the strings. For each node, we have too many node pointers which are equal to the no of characters in the worst case.
- An efficiently constructed hash table(i.e. a good hash function and a reasonable load factor) has O(1) as lookup time which is way faster than O(l) in the case of a trie, where l is the length of the string.

References

- https://en.wikipedia.org/wiki/Trie#History,_etymology,_and_pronunciation
- Algorithms, FOURTH EDITION, PART II, by Robert Sedgewick and Kevin Wayne.
- https://www.geeksforgeeks.org/auto-complete-feature-using-trie/
- https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-trie/
- https://www.geeksforgeeks.org/trie-insert-and-search/
- https://www.geeksforgeeks.org/trie-delete/