Trie vs Hash Table

This lesson highlights the differences between a Trie and a Hash Table, including their common applications.

Comparison between Trie & HashTable#

One might wonder what is the need of using Tries when we can implement dictionaries with Hash Tables as well. A simple answer to this would be, yes, you can use Hash Tables to build dictionaries, but if you need a fast lookup and have long words which share common prefixes then a `Trie` is the perfect data structure for you. It also makes storing words easier, as the implementation is very simple. Some of the key points which differentiate a `Hash Table` from Tries are given below:

1. Common Prefixes#

In `Tries`, the words having common prefixes share a common path until the end of the **prefix**. After that, they are divided into two branches. We cannot do that in `Hash Tables`; no matter how similar the words are, we would still need to store them at different locations. This would result in irrelevant iterations while searching. Here is an interesting example to explain what we just said: two words **"interest"** and **"interesting"** will be stored differently in a `HashTable`, but in a `Trie` we only need to add **3** new `nodes` for **"ing"** at the end of the **"t"** `Node`. Did you notice the space efficiency?

2. Lookup for Exact Words#

As discussed in the previous lesson, `Tries` can perform a spell-check, but in `Hashing`. We can only look up exact words, otherwise, it will not be able to identify the word.

3. No Re-hashing Required#

The most important part of a `HashTable` is the **Hash function**. It is often very difficult to build as the performance of **HashTable** is entirely dependent on it. But in Tries, we do not need to perform **re-hashing** to generate an index. It just traverses the nodes and inserts new `nodes`, that's it!

4. Collision Resolution#

One downside of using a `HashTable` is that we always need to come up with good **collision resolution** strategies to avoid collisions if the data is huge. A **collision** can **never** occur in `Trie` because all words are unique and can act like a **"key"**. This makes the implementation of Tries so much **simpler!**

5. Memory Requirements#

In **worst case** scenarios, a `Trie` will definitely perform better than a `HashTable`, but HashTables will be more convenient to use in average cases-- depending upon the efficiency of the **Hash function**. As in `Trie`, we need to allocate **26** pointers at every node even if most of them are **Null,** so a `HashTable` would be more of a wise choice here!

If your **dictionary** contains a lot of words with common prefixes or suffixes then `Tries` would be an efficient data structure to use as compared to Hash-Tables.