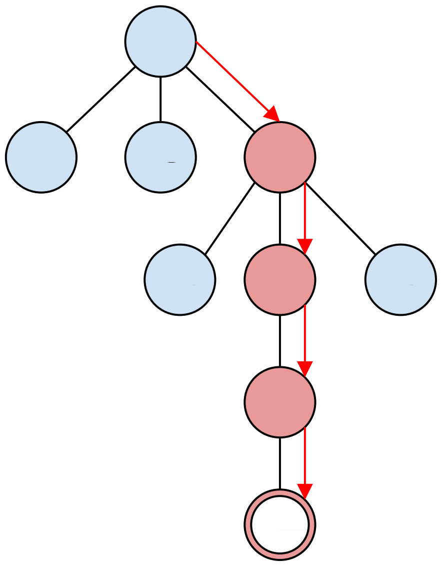
Advanced Data Structures - Tries

Learning Objectives:

- Understanding a new data structure Trie and its applications.

1. Trie

Recall how you look up for a word in an English dictionary (of course the paperback). Say you want to find a word “trie”. You first go to the page where the first letter, “t”, starts. Among those words beginning with “t”, you look for those who have a letter “r” following “t”, and so forth. If we visualize this process, we can present this as a “tree” structure as follows:



We notice some properties of this tree --- the root is empty, and all the other nodes are represented by a letter. Thus, looking for a word is simply a search problem of the tree. Starting from the root of the tree, we keep going down until we reach the end of the word. Then, all the nodes along the path from root to the destination becomes the spelling of the word.

1.1 Structure

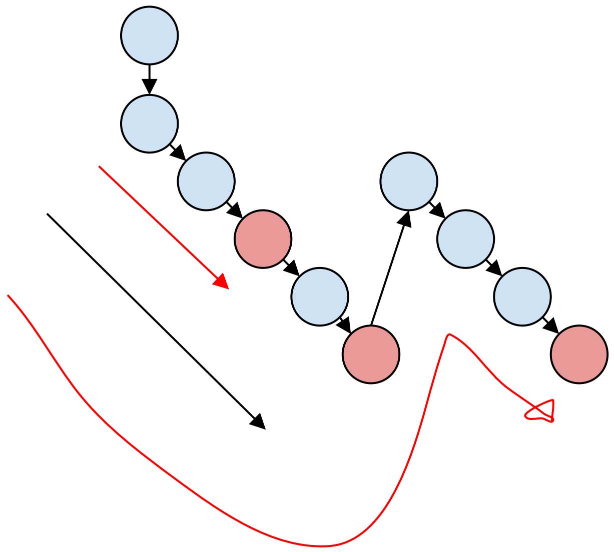
As usual, we first define the structure of each node in this tree. Recall that in BST, we have a field called value, and pointers to its left and right children. For Trie, there are more than two possible children (26 at most in English). So, we arrange them into an array, and define the struct as follows:

```
struct TrieNode
{
        struct TrieNode *children[ALPHABET_SIZE];
        bool isEndOfWord;
};
```

Note that in the structure, `ALPHABET_SIZE` can be different. For English, we set it to 26. Another field we need in this structure is a boolean called `isEndOfWord` (not "isEndOfWorld"). Here, we present a simple example of the usage of this variable. Consider that we have three words:

```
the, there, therefore
```

All these three words are in the path from the root to the leaf which ends with "therefore". If we want to find the word "the", we can simply stop at the first "e" along the path. At that point, all the letters from the root to this node become a word "the". Thus, the node "e" could mark the end of the word, and that's why we need a boolean variable.



Except the basic definition above, we can also associate each word with a count, indicating how many times it appears in a given context.

1.2 Inserting a word

Before we read in any word, the trie has only one empty node, which is also the root. As we read in more words, we will probably need to add new nodes. If the word is already in the trie, we should record the time this word appears in the context.

```
Coding Time!!... read from the notes that has the commented code.
```

1.3 Searching for a word

Searching for a word is similar to inserting a word. In fact, the common part in both searching and inserting is that at each node `i`, we need to look at the next letter `j` from node `i`'s `children`, to see if a node `j` exists in that pointer list to form a bi-gram "`ij`".

```
Coding Time!!...again...look at the commented code.
```

2. Tasks that you can do to improve your concepts

You need to finish the following tasks to have a solid understanding of this Data Structure:

Programming Exercise:

1. Modify the structure of trie nodes so that we can record the number of times a word appears.
2. Read in a text file and create a trie. Then when you search for a word, it should output the number of times that word appeared in the text file. Otherwise, output `-1`.

Analysis:

1. Implement a simple solution where we use array to store words. Then compare the difference of running time of searching a word using a trie and an array.