Tries
Wednesday (Aug 21, 2024)

==================== Instructions====================

1. In the case of user input assume only valid values will be passed as input.
2. You can use C or C++ as the programming language. **However, you are not allowed to use any STL libraries in C++.**
3. Regarding Submission: For each question create a separate C file. -> <rollno>_Q1.c. Create a zip file of all these C files in the name <rollno>_A5_Tries.zip and submit it to Moodle. For example, if your roll number is 24CS60R15, then your file name will be 24CS60R15_Q1.c and your zip file name will be 24CS60R15_A5_Tries.zip. **All the tasks should be done in a single C file only.**
4. Inputs should be taken from a file and outputs should be printed to file as written below.
5. **You have been provided with a boilerplate.c file. You may use it accordingly.**

==

It should be able to detect if the input is present in the dictionary, find all the words in the dictionary which begin with the given input and autocorrect the input word. For each word, we will maintain a rank [Initially all words have a rank of 0]. The rank of a word gets updated by 1 every time it gets used in a query. [More details below]

Input and Output :

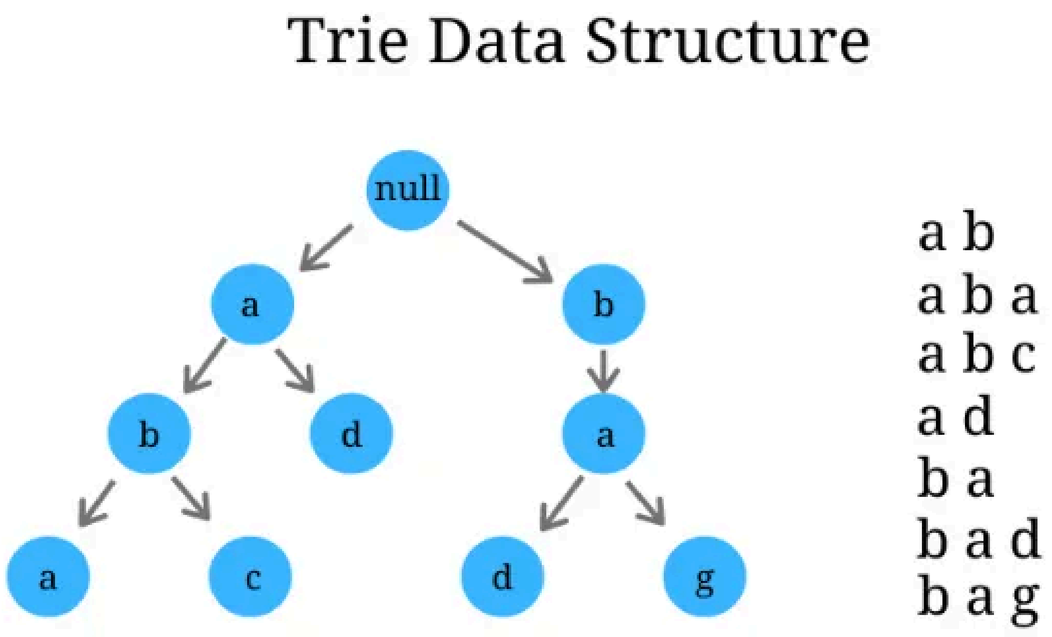
a. First Line contains 1 integer **Q**. Where **Q** is the number of queries to be processed.
b. Next **Q** lines will contain two space separated values, First one will be an integer **x** and second will be a word **w**.
c. The output format is mentioned in the shared C File.
d. There would be 2 files, dict.txt and input.txt. The `dict.txt` is used initially to fill the trie. The `input.txt` will contain the query part mentioned above.

Notes :

1. Whenever updating of rank is mentioned, increase its value by 1.
2. **Rank of each word can go to a maximum of 5**. If a rank 5 word gets called in a query again, do not update its rank.
3. All Inputs will be from a file.
4. All outputs should be in a file.
5. In information theory, linguistics, and computer science, the **Levenshtein distance** is a string metric for measuring the difference between two sequences. The Levenshtein distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

6. Given a `boilerplate.c` file. You may use it accordingly.

Introduction to tries:



```
struct TrieNode {
  struct TrieNode *children[26];  //Each node has 26 children(for the
26 letters)
  int isEndOfWord; // Flag to indicate node represents end of a word
};
```

Task 1(x = 1) : Adding an element to Trie

Given a word **w**, complete the function **Add()** , that adds the word **w** to the dictionary with the default rank of 1

```
void Add(TrieNode *root, char *w);
```

Explanation: Start from the root node, and for every character in the word **w**, check if the corresponding node for that character exist, if not, then create a new trie node and perform the same action for the following characters in the word **w.**

Task 2 (x = 2): Deletion in Trie

Delete the word w from the dictionary.(Also take care of its rank)

```
// Deletes the word w from the Trie,ensuring the rank is removed appropriately
void Delete(TrieNode *root, char *w);
```

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 the prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having at least one other key as prefix key. Delete nodes from end of key until first leaf node of longest prefix key.

Task 3 (x = 3) : Spell Check

Check word **w** for spelling, If it is correct, print 1 and update the rank of the word. If it is not, then print 0 (and do not update any rank)

```
int SpellCheck(TrieNode *root, char *w);
```

Explanation : Just traverse the trie for the given node(the code should be similar to adding a word, but whenever you encounter a character in the word for which there is no trie node associated, report that the word is not there in the trie.) If it do not exists, **call Add() form previous task** and then print **Not Found (given in the shared C file).**

Task 4 (x = 4) :(Autocomplete)

Find all the words in the dictionary that match **w** and print the top 5 in decreasing order of their rank (**w** is the prefix to all these words). If there are two or more with the same rank, print them in lexicographical order. [Update the rank of the 5 words by 1(if possible)]

```
void Autocomplete(TrieNode *root, char *prefix, FILE *outputFile);
```

Task 5 (x = 5): (Autocorrect)

Find all the words in the dictionary which are at an edit distance(Levenshtein distance) of at most 3 from w, then display the highest ranked word (If there are multiple, print the lexicographically smallest word) and the update the rank of that word(if possible)

Note: You need to make use of the existing trie to calculate the edit distance and stop if the edit distance becomes more than required, to avoid unnecessary calculations.

```
void Autocorrect(TrieNode *root, char *w, FILE *outputFile);
```

Task 6 (x = 6)

Print 1 and update the rank of w by 1 **if the word w can be formed by joining two or more words from the current dictionary.** Otherwise just print 0.

```
int CheckConcatenation(TrieNode *root, char *w, FILE *outputFile);
```