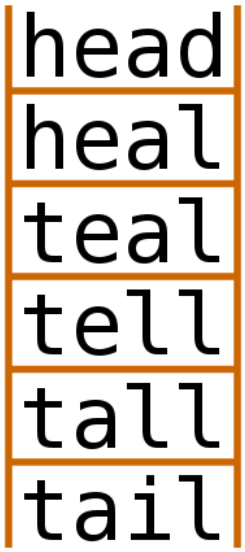
WORD LADDER



For this extra credit assignment, you will work with templates and several container data structures from the C++ standard library (at least *vector*, *map, stack/queue,* and *set*).

You will implement a word puzzle invented in the 19th century by Lewis Carroll. For this game, you will be assigned a starting word and a target word. At each step of the game, you must change a single letter of the word into another word (it is forbidden to transform a word into a non-word).

For example, if the starting word is SAME and the target word is COST, one solution would be the following:

SAME

CAME

C**A**SE

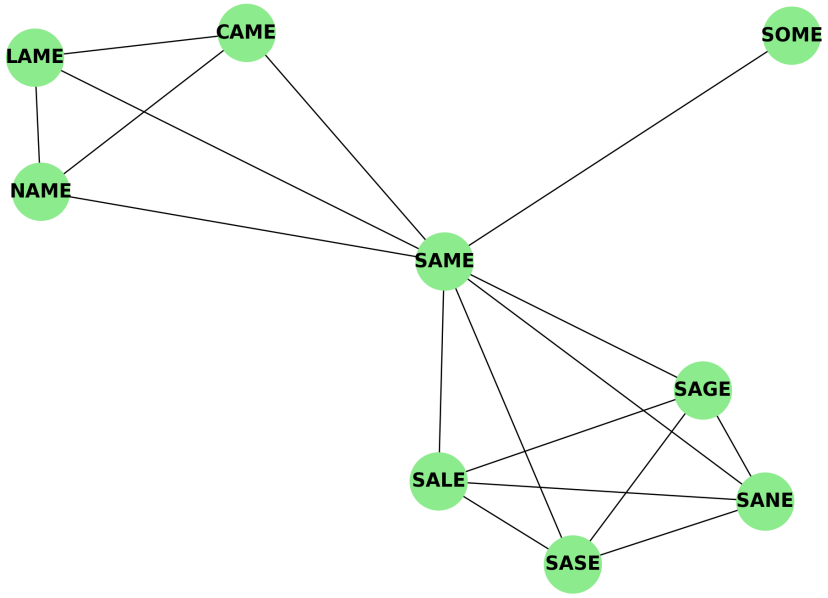
CAS**T**

C**O**ST

In this assignment, we are interested in computing the way we can transform the starting word into the target word. We will achieve this using a graph data structure.

More specifically, we will construct a graph based on a dictionary containing words with the same number of letters. The nodes in the graph represent the words from the dictionary; there is an edge between two words if you can transition from one word to the other by changing a single letter.

Assuming that the words in the dictionary are: "LAME", "NAME", "SOME", "SASE", "SAME", "SANE", "CAME", "SAGE", and "SALE", you should construct the graph below. Each node represents a word, and edges connect words that differ by exactly one letter, illustrating the possible transitions in your word ladder game.



To begin, you will need to define an abstract data type to represent a graph. **You should use templates** for this (i.e., you should be able to store any data type in a node in the graph). The graph should be represented by adjacency lists. To store the lists of neighbours for a node, you **should use an STL container** (you'll have to choose the appropriate one and argue your decision). It's up to you what private and public methods you define for this graph data structure.

Now that we have the building block data structure for our problem, we should think of how you can construct the graphs based on the words from a dictionary (by dictionary we refer to a simple text file that contains all the valid words with the same number of letters).

The straightforward approach would be to generate all the possible word pairs and compare the words that make up a pair. If they differ by a single letter, we add an edge in the graph, otherwise, we don't. However, this is not that efficient – it has $O(n^2)$ complexity – and it turns out that we can do better than this if we use the appropriate data structure.

To achieve this, we'll be using the concept of a wildcard and a balanced tree data structure (don't worry, you don't need to implement it yourself, you will use the implementation *map* from STL; this is similar to a dictionary in python).

A wildcard is a special type of character that can be replaced with one or more characters in a string. For sure you have use wildcards before when working in the command line; for example, to list all the .cpp files in a directory you might have used:

```
ls *.cpp
```

Here * is a wildcard.

Let's see how we can apply these two concepts (wildcard and map) to come up with a more efficient strategy to build the graph.

We will use an additional map data structure to store all the words that comply with a given pattern; a pattern is defined by a word in which one and only one of its letters have been replaced by a *, and therefore it can be replaced by any letter.

For example, we might have the following patterns and words that comply with that patterns:

*AME : CAME, LAME, NAME, SAME

S*ME: SAME, SOME

SA*E: SAGE, SALE, SAME, SANE, SASE

In the first line from the example above: we have the pattern *AME. This will be a key in the map. In the bucket corresponding to this key, we add all the words from the dictionary that correspond to this pattern: CAME, LAME, NAME, SAME. Of course, this example is not exhaustive.

This map will have as key a string (the pattern) and as values a list of words that comply with that pattern. To construct the map, we iterate through all the words in the dictionary and add them to all the corresponding patterns in the map.

Then we can start building the graph: for each key in the map, we take all the words that comply with that pattern and we add an edge between all the possible pairs from this list.

Now we can drop the additional map data structure and focus only on the graph.

The problem of finding the best transformation from the starting word to the target word comes to finding the shortest path between the starting word and the target word in the graph that you created.

I leave it up to you to choose the algorithm you prefer to achieve this, but I suggest that you use a container from STL in this implementation. For example, you could use breadth-first search (BFS) to traverse the graph from the starting node and then use the *parent* array to track back the words you passed through to reach from the source (starting word) to the destination (target word). In the BFS implementation, you can use the *queue* from STL.

To sum up, for this assignment you have to:

- Define a class for a graph data structure; the graph is represented with adjacency lists. This class should use templates.
- Implement an efficient algorithm for building up the graph for this game;
- Implement an algorithm to find the shortest path between two nodes in the graph. Use this algorithm to find the transformations that you need to apply to reach from the starting word to the target word.
- Create a Qt widgets application for this game (you can design the UI as you wish, but should use widgets, signals and slots and layouts). This application should have the following options:

- Automatic mode: you select the starting word and the target word and the application displays the transformations you need to perform.
- Playing mode:
    - the user is prompted for his/her name;
    - the user first selects the number of letters for the word, and then the starting and the target word are randomly selected from the appropriate dictionary (based on the number of letters of the word).
    - the user then plays this game and the application needs to ensure that the user respects the rules: i.e., when transforming a word to another one, a single letter must be changed and that word should exist in the dictionary.
    - during the game the user can ask for hints: in this case, you need to suggest to the user which letter he/she should change to reach faster to the solution. When displaying the hint, you should colour differently the letter that should be changed.

    - when the user is done, you should compute the following:
        - the starting word and the target word;
        - date and time of the game;
        - how many hints did the user use;
        - which were the words that the user used;
        - how many moves did the user do to reach from the source word to the target word, and what is the optimal number of moves that he could have used.
    - and save this data into a file *username.csv* (or you can choose a different format). if that file exists, you should append data to that file and not re-create it.
- Analytics module.
    - This will compute some statistics about a user's game.
    - You first type the name of the user, and the application loads the data from the corresponding file (*username.csv*)
    - Then you should display the number of unique words the user used when playing the game (you should use a *std::set* to achieve this)
    - … and optionally, any other statistics that you think would be useful.