Juan Bustamante
Sebastián Jaramillo

Engineering method

Problematic situation:
As a way of training, the ICESI university has given its VIP Simulation team the task of developing a software tool to simulate and efficiently manage large databases of people.

Step 1. Problem identification

1.1 The software should:

- **Generate automatically** a given number of people according to the next conditions:
  - Each person has a `name`, `surname`, `gender`, `birthdate`, `height`, `nationality`, a `picture` and an auto generated `code`.
  - To generate a compound name, the program must use this first name dataset and the larger file in this surname dataset.
  - The continent wide birth dates must be generated using this age distribution for the USA.
  - The nationalities must follow this population per country data.
  - The gender could be 50% male and 50% female; the height should be taken randomly from a reasonable interval; and the picture should use this website ignoring gender, age and others.
- **Show** a progress bar in case the "generate" process takes more than 1 second.
- **Show** how long did the "generate" process take.
- **Save** the generated data to load it when opening the program again.
- **Create** a person.
- **Delete** an existing person.
- **Update** the info of an existing person.
- **Search** separately, by:
  - Name
  - Surname
  - Compound name
  - Code
- **Show** a suggestion list when searching by Name or Compound name.

1.2 The problem:

We need to look for an efficient way to develop a CRUD(Create, Read, Update, Delete) system to manage a huge database of people as a training.

Step 2. Information gathering

From now on, we are focusing on solving the item:

- **Show** a suggestion list when searching by Name or Compound name.

2.1 Some definitions

- **prefix:** a string of characters that many words share at their beginnings. For example a prefix for [Color, Colombia, Colmenares, Colision] would be Col.

2.2.1 How has this been done before?

In this gist by floralvikings, we can see a class that provides suggestions for a given prefix, based on a previous set of known words. The algorithm to find the suggestions is, in short:

```
known_srt = ordinated list of all the strings to search suggestions in
    getSuggestions(prefix):
        first = lightest word in know_str that starts the prefix.
        last = largest (heaviest) word in know_str that starts the prefix.
        return know_str.subSet(from: first, to: last)
```

To illustrate it, here in line one we can see the ordered list[1] of known words. Now, running the previous algorithm with BL as a parameter we get the suggestions starting with BL.

```
[BLACK, BLOCK, BLUE, BROWN, GREEN, ORANGE, RED, WHITE, YELLOW]
prefix: BL
sub set: [BLACK, BLOCK, BLUE]
```

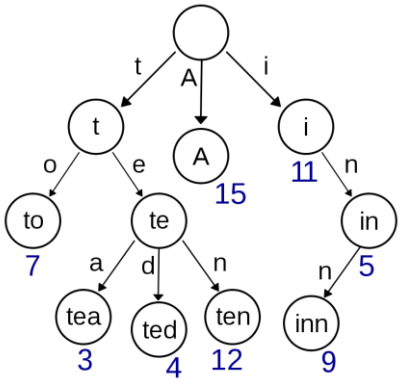
Finally, in case it is not clear, "BLACK" would be the lightest word starting with our prefix, as it is the sooner to appear in the ordered list; and BLUE would be the heaviest. Those two and all the words in between constitute the suggestions sub set.

The same approach could be seen in the MDIUtilities library (docs about auto completion), that uses a dictionary to store known words, and search suggestions with a similar strategy we saw before.

2.2.2 The *tries*

Autocompletion is one of the most used tools today. Even though many people don't notice it, we use it on the daily as we type on our phones or search something on Google. It is not a surprise that there are advanced approaches on the matter.

In the article Data structures for fast autocomplete, Antti Ajanki, introduces us to a kind of tree specialized in key searches: The Prefix tree, also known as *trie*. In short, this structure stores values in its nodes, but does not store or calculate keys. Instad, the path to a certain value defines its key. This works because each link represents a character and they 'add up' to make words. So, given a certain prefix, the tree can fastly search it and identify which known words could be suggested.



But, although fast, the Prefix trie is notably memory-exhaustive. Luckily, there is its memory-optimized sibling: Radix tree. Fundamentally, both work the same way. However, in a Radix tree, if a node is the only child of its parent, both are merged into only one node. This allows the structure to store the same number of words using less nodes, and therefore, consuming less memory.

Step 3. Ideas for a solution:

3.1 An ordered list

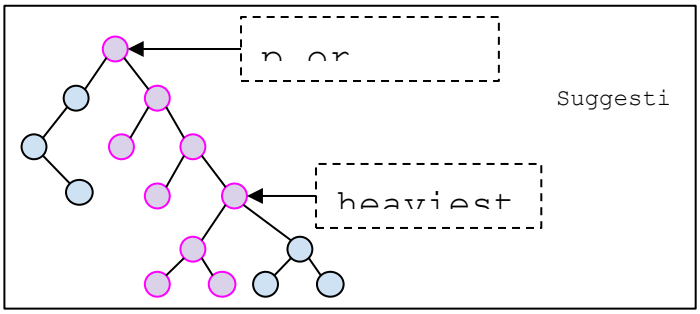
As the simplest solution, we could implement a list like the one we talked about before. As the user adds people to the database, the list must be kept updated with all the names to provide complete suggestions. Being the list ordered, binary search could be used to find the first and the last word starting with a given prefix; and return that segment.

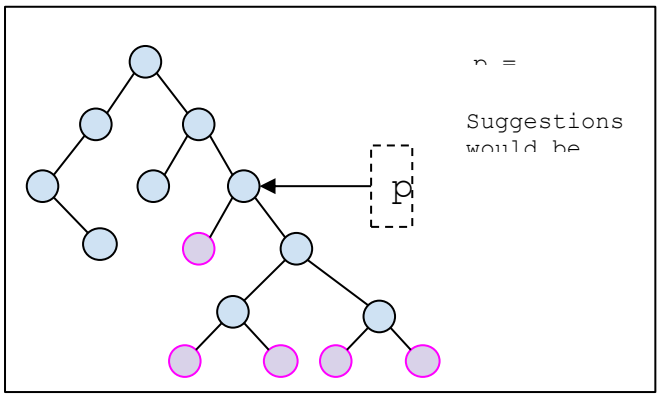
3.2 Multiple ordered lists

A list with names starting with all the characters in the abecedary could be inefficient, because, as soon as the user presses a letter to search, 1/25 of the list is discarded. This could be fixed using a matrix where every row is a letter of the alphabet. As soon as the user types a character, the program would know in which list search suggestions, reducing the search time greatly.

3.3 Segments of a tree

Orderly adding items to a tree is easier than adding them to a list. A binary tree could be used to store all the Compound names and just extract a segment when searching for suggestions. This could be done like in the image:





3.4 Prefix tree

A prefix tree would exploit all the fastness of adding in a tree, plus it would not be necessary to extract a portion of the tree to get suggestions, as every node in the tre are prefix themselves. NOTE: a prefix tree it's not binary, but due to space we could not include more nodes in the image.

3.5 Radix tree

As explained before, a radix tree brings all the benefits of a prefix tree, plus memory optimization. However, the 'add' and 'remove' action is a bit more complex in the radix tree, as it has to merge/split nodes depending on the value of the altered node added.

Step 4. Transition from ideas to preliminar design:
After some tests, it has been decided to discard both 3.1 and 3.2 solutions. Here are some of the reasons:

4.1 Discarding solutions

- Both solutions were highly memory-demanding, without bringing any speed advantage
- Both solutions could be achieved nicely using 3.3 and 3.4/5 sol. This is, if we want to search and separate a segment (3.1), this could be done by a tree; on the other hand, if we look to divide the problem into smaller ones (3.2), it is easier (read cleaner) to do with solution 3.4 or 3.5.

Step 5. Solution evaluation:
The next criteria would be applied to each solution to find out which one fits the problem better:

5.1 Aclarations:

- The only possible answers for each item is YES/NO, the solution gets a point if the answer is yes
- When the word 'relatively' is used, we are answering based on Github issues/comments, Stackoverflow threads and the team perspective.

5.2 Criteria:

a) The solution divides the problem into smaller problems.
b) The solution has a memory usage/operations speed equilibrium.
c) The solution is relatively easy and clean to implement.
d) The solution is specialized in this problem.

5.3 Selection:

Criteria \ Solution	a	b	c	d	Total
3.3 Segments of a tree			*		**1**
3.4 Prefix tree	*		*	*	**3**
3.5 Radix tree	*	*		*	**3**

There is a draw between both *tries*. We have chosen to implement the Prefix tree, as it is the simplest approach.

1: Yes, we know 'BLOCK' is not a color, but we were lacking words starting with BL ;).