2022

DiCtionary

The new app which will change your life

PROJECT C THIRD SEMESTER

PAUL MAIRESSE & AXEL LOONES & QUENTIN CARDONA

INT-1

INTRODUCTION

As part of our course of second year of C, we had to make a dictionary in C. This project was planned for a length of 1 month and in groups of 3. The goal of the project is to search a word in a dictionary extracted from a file and generate grammatically correct sentences from it.

For this project, we had to store the word in **N-ARRY TREE**. For the structure of the **NODES,** we chose to use **STD_LIST** to optimize the space in the memory, particularly in last **NODES** with few children.

This program allows a person to either search a word in the dictionary: basic form or complex one. See all its relative information: alternate forms, basic form associated, attributes like singular, feminine, masculine, participle, imperative, …

To organize ourselves in the group, we used git and GitHub for the versioning of our code and easy access between every member. This also allowed us to save past versions of the code in case of need.

We chose to use *vs-code* with **GCC** for compilation and IntelliSence because it's a tool that we are comfortable with and highly customizable to match all our needs.

SUMMARY

FUNCTIONALITIES **3**
- Search for a word 3
- Display a word 3
- Choose a random word 3
- Generate a random sentence 3
- Extract data from files 3

OPTIONAL FUNCTIONALITIES **4**
- Add a word in the dictionary 4
- Autocompletion for a word 4

DATA STRUCTURE OF THE TREE **5**
- The problem 5
- Our solution 5

SEARCH **6**
- Basic search: 6
- Advanced search: 6
- Autocompletion: 6

RANDOMNESS **8**
- Simple Random 8
- Ponderation of the tree 8
- Traversal random 8
- The rand function 8
- Results 9

CONCLUSION **10**
- What we learned 10
- How we organized ourselves/our difficulties 10

FUNCTIONALITIES

Search for a word

You can search for a word in the dictionary file. You can decide to search specifically for a type to get a result faster or to not specify it. The function will return the first occurrence of the word in all trees. There are also two types of searches, one more efficient for only the base words, and the other for alternative forms.

Display a word

We decided that the displays in our project were very important, as it is what the user sees. Thus, we added many colors and font styles thanks to the **ANSI** codes and made the user interface friendly. The navigation through the menu is intuitive, and we arranged for example the display of the words so that, if you are searching for a specific form, you will find it easily. Indeed, instead of displaying all alternative forms in alphabetical order, they are displayed following their tags.

Choose a random word

To generate a random sentence, we first had to get a random word from the trees. There are two types of randoms: one slightly faster than the other, but not uniform at all, while the other is fast and uniform. Indeed, the first one only goes randomly into one of the children of the nodes and has a small chance to stop at each node if there are words. The second one however is a bit longer as it must build a ponderation at least once for each node of the tree following the number of words in the children of the node. It is also more uniform as each word has an equal probability to get chosen.

Generate a random sentence

The second choice in the menu allows you to generate a random sentence. You can generate a sentence with random words following three models:

- **NOUN** - **ADJECTIVE** - **VERB** – **NOUN**.
- **NOUN** - "qui" - **VERB** - **VERB** - **NOUN** – **ADJECTIVE**.
- **VERB** - **NOUN** - **PAST PARTICIPLE** - "que" - **SUBJUNCTIVE** – **NOUN**.

Extract data from files

We used the file **DICTIONNAIRE.TXT** to import all the words and were able to implement them in a tree by separating the form, the base form, and the tags the word has. We were then able to change this tag into a binary flag using **GETFLAGS()**, a function that takes a string for its input and returns the binary number corresponding to the information of the string.

OPTIONAL FUNCTIONALITIES

Add a word in the dictionary

You can also add a word in the dictionary if it doesn't already exist. This option can be used in two ways: either directly from the main menu, or after searching for a word that doesn't exist. When adding a word, information about the word is asked to the user who can thus add the word exactly as he wants it to be. The word is then added to the corresponding tree and in the file.

Autocompletion for a word

This function allows to complete the search when entering *tabulation* on keyboard. If there are less than 10 words which can be found to complete the search, they are displayed, and the user can then see the remaining possibilities for the word he wants to enter. Autocompletion only works with the base form of words for performance. Checking alternate forms would require checking all the trees of the dictionary in the worst case and wouldn't be instantaneous on some computers.

DATA STRUCTURE OF THE TREE

The problem

We chose at first to make a specific structure for every type, with an enumerator to store the specific type of every form and an array to store the links to children. This structure is not adapted at all to the project because it forced us to make different functions for each type and it weighs a lot on the memory for nothing: two to three fields for every form and minimal array size a lot bigger than the mean number of children of the nodes of the trees. Because we were going in the wrong direction, we chose to remake everything from scratch and rethink our structures entirely.

Our solution

For our final structures, we chose to only make 3 different ones versus the 19 different ones from before. First, we chose to totally abandon the use of array and use **STD_LIST** instead. **STD_LIST** is more adapted to this kind of project because we can have a different number of children at each node without reallocating memory each time. The second big difference is the use of int for tag. By translating each attribute of a form to binary bit flags, we were able to store each type with only one field which is better on every aspect.

Our tree is composed of three structures. **NODE** for each leaf of our tree which contains the data (letter and form associated). **FORMS** which is a **STD_LIST** containing a word and an associated tag which represents the list of forms of a leaf. And **CHILD** which is another **STD_LIST** containing the list of children of a **NODE**.

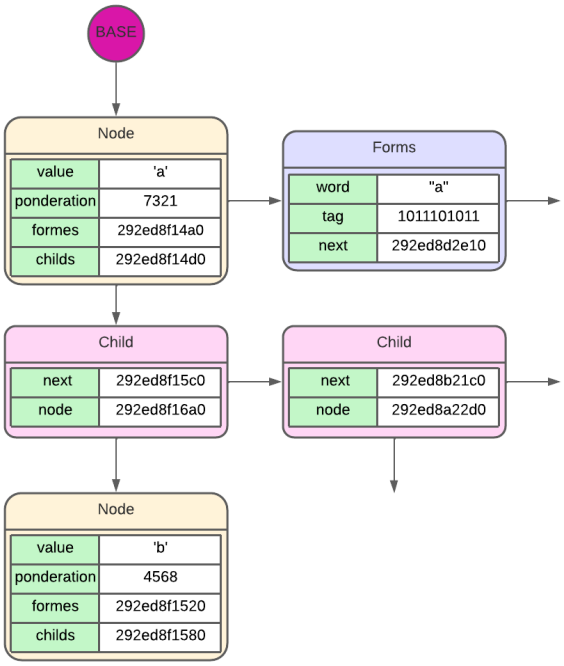

Figure 1 : Flowchart of the structure of the tree

SEARCH

Basic search:

Our basic search is a simple function. It allows to search for the basic form of a word in trees. The function takes each character of the word given and goes in the tree node by node. At each step it searches the next character in the children of the actual node. If it can't find it then it returns **NULL**, else it continues until the last character and returns the associated word (basic form + associated form).

Advanced search:

The advanced search allows to search alternate forms in addition to basic forms of a word. The task can be difficult because in most cases the letter to the word gets you to another path of the tree than the one with the basic form and every associated form. To avoid checking the whole tree each time we used recursion, memory cache and a stack. A first function explores the tree like the basic search until it loses the word or finds it and stacks each node of the path. Then a recursive function unstacks a node and explores each child. If the word is found the node is then returned, else the function unstacks another node. Using this, the function searches the word at the closest nodes to where it lost the word and goes away step by step. To avoid checking again previously explored children when unstacking a new node, we add each address of explored nodes to a hashable list. Each time the function is called and doesn't find the word, it adds an entry to a list with the address of the node. Then at each call it checks if the node is in the list. If it finds it, then it returns **NULL** without checking the forms and the children of the node. Since the root of our tree is an empty node, we can unstack all letters of the word and find the basic form of an alternate form where the first letter is different ("irons" and "Aller" for example).

Autocompletion:

We replaced the **SCANF()** of our search function with our autocompletion function. By pressing *tabulation*, we can list the basic forms of every word we can find in the dictionary after our last letter. With this, we wait for the user to enter a key and store it into a queue. If the user enters *tabulation* the autocompletion searches for words in the dictionary. If more than ten are found, it stops the search and displays an alert, else the list of word is displayed. If the user enters the return key, the input is validated, and it returns to the search function.

To make this function, we had to overcome three main problems. The first one is about the display when the user uses backspace. Every time a key is pressed, we print the char to the stdout. But since we can't remove a character from stdout, we found a solution by printing *"\b \b"*. That replaces the last character by a space and replaces the cursor just before it. The second main problem is that on the UNIX system, stdout is buffered until a *"\n"* char, so we cannot read by default was the user enters until he validates his input. To solve this, we can override the parameters of the terminal in the code. Finally, our last problem was that by default, on the **UNIX** system, accents are encoded on multibyte char, so we must read two

char and not only one. To solve this, we used a string of size 3 where we put one or two char before returning it to the autocompletion function.

RANDOMNESS

Simple Random

At first, we chose to make a simple random algorithm. The function is called at the root of a tree. If the node is a word, it has a 1 chance out of 7 to stop, else it chooses a random node in the child and calls the function again at the chosen node. The function can be called at most the same number of times than the height of the tree, like 21 for noun for example with *“radiocristallographie”.*

Ponderation of the tree

We rapidly deduced that this random wasn’t random. Indeed, the trees aren’t balanced because branches with less words in its children will have a higher probability to be selected than other branches. To solve this, we chose to build a ponderation of the trees and used them to weight our probabilities. The ponderation’s building function is a recursive function that explores all the nodes of a tree and adds one each time it found a node with a word. The root will contain the total number of words (base form only) in the tree and at each node the number of words under them (the word at the current node is considered). This function only needs to explore the tree once to calculate all the ponderations.

With those ponderations, we can choose uniformly if we stop at a node or continue to a randomly chosen children. Each branch has a probability to be taken relative to the number of words it contains.

To test the difference, we chose to make a small directory with 20 words, called our random ten million times and made percentages with the occurrences of our generation. We could clearly see a difference between the two functions. The first only gives two results with approximately *50%* each while the weighted one returns each word with *+/- 80%* of precision which was a disappointing result. We identified that the problem was the **RAND()** function. We then chose to use the **PCG-RANDOM** library to get uniform results. The difference was now of *+/- 70%*, which was still disappointing.

Traversal random

We then chose to change our approach and generated a number between 0 and the number of words in our tree, and then made a prefix traversal of the tree removing 1 from our number each time we encountered a word. If the number was negative, we then returned the word. With this algorithm, we got a near perfect result at *+/- 0.01%* with the previous test. While our random was uniform, we now had to explore in the worst-case scenario the whole tree before returning the word. To take the noun’s tree again, that’s *277877* calls of function versus the previous *21*.

The rand function

By searching a bit more, we found out that the problem with our random number generated by **RAND()** wasn’t the function itself but the use we made of it. Each node has a

different number of children or ponderation, so a different interval of generation. Even by removing the imperfection of **RAND()** with better generators like **PCG-RANDOM**, we stilled called the function with different intervals each time. For an unknown reason, this caused our generation not to be uniform. To solve this, we chose to generate a float between 0 and 1 and to convert our ponderation to a probability between 0 and 1. With this, our second function was able to match the results of our traversal random with a performance matching the one of our basic random.

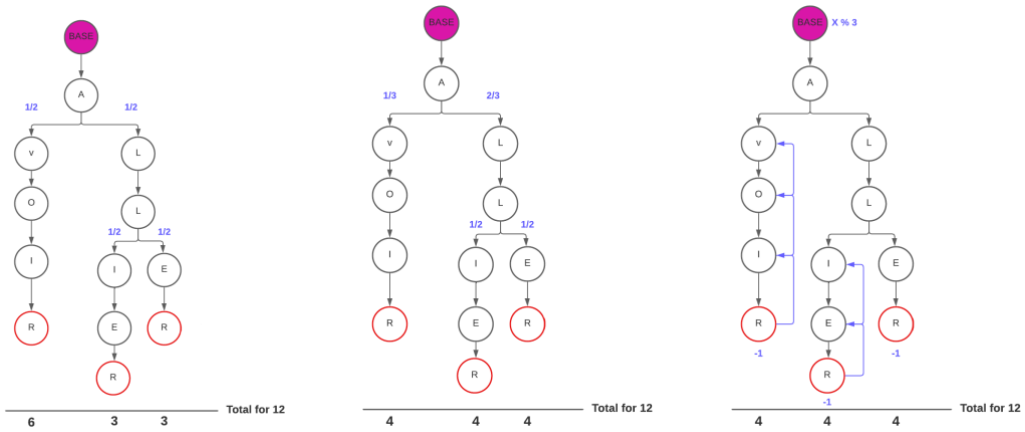

Figure 2 : chart of the 3 random functions

Results

To check the different functions, we called them each 10 million times and sent the output to a file with commands. By reading the file with a **PYTHON** script, we were able to make diagrams of the results with **MATPLOTLIB**. On the performance side, the basic and weighted randoms are equivalent with *5 second* computation time versus *4 hours* for the unoptimized traversal random. On the uniformity, the basic random returned only *10%* of all the words with a standard deviation of *1900*. On the other hand, the two others returned all the words with a difference of only *400* occurrences between the max and the min. The standard deviation for both was of only *20*.

The following diagrams represent the results of 10 million calls, words in sorted group of 100 on the abscissa and the average number of occurrences on the ordinates.

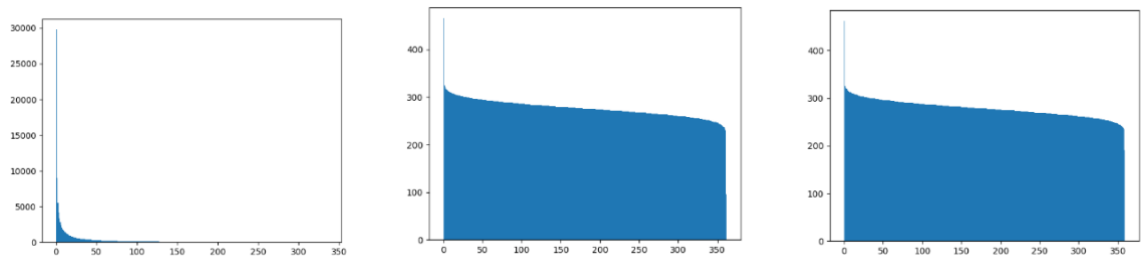

Figure 3 : Result for basic random, weighted random and traversal random (from left to right)

CONCLUSION

What we learned

We first learned how to design our structures to be the most efficient at runtime and in memory. The structures also needed to be easy to use and reusable to avoid problems and recurrences of same code in different functions.

Random was also an interesting part as we discovered that a pseudo-random function isn't always uniform and the way we use them can be impactful. Indeed, changing the interval of the modulo for each node for example is not very efficient, and we had to find other solutions. Thinking about many ways of getting what we wanted was very interesting, because not only did it need to be functional, but also efficient and uniform at the same time.

Finally, we learned the importance of **UNIT TESTS** in projects. With the 300 thousand different lines of the dictionary, testing the function once, twice, twenty times isn't enough at all. Without those tests, we would have maybe executed some functions like search for example 1000 times at most. If we had a bug that shows in only *0.01%* of the cases, we would never have noticed it. The first time we executed the test file, the search failed before even reaching *B* in the dictionary. With this complete test, we were able to find all cases where the function didn't work before moving on. Also, this allowed us with **PYTHON** to make statistics and charts of our random and improve it to represent the whole dictionary, not a small part only.

How we organized ourselves/our difficulties

We encountered many difficulties during this project, on many different subjects. The first one was thinking about the structures and, as stated before, we had to start the project over because the first ones we made were not good enough. At first, we decided to distribute the different structures between each member of the group but, with the new structures, only one person needed to work on it so the others could start the rest of the project.

We decided at the start that Quentin oversaw the random and the sentence construction, while Axel took care of the menus and all the displays. Finally, Paul made the search and autocompletion algorithms. However, the tasks we distributed were also in function of the time we had and as stated before, we made sessions to solve bugs together and see if we could help each other with the difficulties encountered. **VISUAL STUDIO CODE** was also useful for teamwork, especially thanks to the extension **LIVE SHARE** that allowed us to code on the same computer at the same time.

The biggest difficulties we had were supporting every **OS**. By the fact that many functions change from one **OS** to the other, we encountered problems very soon in the development of the project. However, for the **C** project of the previous year, we had already encountered problems like this one, so we knew how to use **MACROS**, and it made things easier. Still, the fact that accents are counted on two **BYTES** on Mac, and on only one on Windows was harder to fix because it changed the function. Thus, we encountered many

difficulties, but were able to overcome them, even if it meant going back a bit in the project or spending a long time on those issues.