**Como estão estruturados os dados**

Os dados estão armazenados em uma tabela de dispersão, um vetor onde cada posição guarda um ponteiro para uma lista encadeada de nós. Cada nó representa uma palavra – *struct word*. O primeiro nó de cada lista corresponde a palavra principal da lista, os demais são nós sinônimos dessa palavra.

**Justificativa para o tamanho da tabela e para a escolha da função de dispersão**

A eficácia da função *hash* depende do tamanho da tabela, dependendo do tamanho da tabela, algumas funções são melhores que outras. Podemos usar números primos para definir o tamanho da tabela, pois esses números só têm dois divisores e dessa forma aumentamos a aleatoriedade de resultados que a função *hash* pode gerar dado entradas distintas. Se a tabela tiver tamanho múltiplo de 2, é mais fácil acontecer colisões, pois nesse caso o que determina o *hash*, são os últimos bits menos significativos, o que é ruim pois os bits mais significativos que tem mais peso, são ignorados.

O tamanho da tabela escolhido foi 977, número que eu considero razoável pois, não é pequeno demais para causar colisões frequentes, e não é muito grande para impactar na performance, além de ser um número primo e não múltiplo de 2.

Foi utilizado uma função de dispersão que dado uma cadeia, percorre ela inteira e realiza um somatório da multiplicação do código ASCII do caractere vezes a potência de *i* de uma constante, sendo *i* o número da iteração.

Essa multiplicação str[i] \* k^i tem o objetivo de encontrar um número que seja o mais próximo de único possível para a cadeia, pois leva em consideração a posição do caractere na cadeia, dado por k^i, reduzindo assim a possibilidade de palavras diferentes chegarem ao mesmo número.

E no final obtém-se o resto da divisão do somatório por 977.

**Partes desenvolvidas com sucesso**

O programa executa a função insere, busca e remove da forma que foi pedido na descrição.

**Partes que não funcionam**

Se o usuário inserir uma palavra e a função de dispersão retornar um índice que esteja ocupado e que a palavra-chave guardada nessa posição for diferente da informada pelo usuário, o programa não vai fazer nada, e seguirá a execução esperando o próximo comando do usuário.

OBS: Se ao inserir um par de palavras (str1,str2) acontecer uma colisão com a str1, não será criada uma lista para a str1 e seus sinónimos, mas se no próximo passo da função, ao inserir o outro par (str2, str1) não acontecer nenhuma colisão, a lista será criada normalmente e a str1 estará presente como um sinônimo da str2.

**Principais dificuldades**

A maior dificuldade foi em elaborar uma forma de tratar as colisões, onde não tive sucesso. Tentei implementar a dispersão dupla, mas no fim das contas só acrescentou mais complexidade no meu programa.

A forma que eu planejei, o encadeamento externo não trata as colisões, ele é armazena os sinônimos de uma palavra.

Encontrei dificuldades em alocar memoria dinamicamente para *strings*. Primeiramente nem me dei conta de que deveria alocar memória para uma cadeia, depois fiquei travado no código abaixo. O código não funcionava, pois, eu colocava a função *sizeof* e esquecia de informar uma posição a mais para o carácter nulo:

Texto

Descrição gerada automaticamente

Esse código é usado para inserir uma palavra na lista, por causa desse erro pensei que a lógica da função de inserção estava errada, ou os ponteiros e perdi um tempo considerável, mas acredito que foi válido, pois eu realmente não sabia trabalhar dessa forma com *strings*.

Também enrolei-me para liberar a memória de uma lista, eu esquecia de salvar o próximo nó da lista, e quando eu tentava reatribuir o nó atual ele não funcionava por que eu liberei ele, logo o programa não ia conseguir acessar para fazer a retribuição. Tive dificuldade em como tratar a função remove, quando recebe uma entrada e quando recebe duas entradas, resolvi essa questão pesquisando na internet, onde encontrei sobre a função *fgets* e *sscanf*. créditos: *ChatGPT*;