Projeto 2

Nome: Maykon Marcos Junior
Matrícula: 22102199

Algoritmos Principais

Estrutura Trie

Organizada como uma árvore 26-ária, para as letras minúsculas do alfabeto.

Nodos: Uma struct privada, com char (letra), 26 ponteiros nodos (filhos), outro para o pai e 5 inteiros, posição (distância da raiz), prefixos (quantas palavras válidas o sucedem na árvore), início (primeira aparição no arquivo), N (comprimento da linha em que apareceu) e um is_word (na verdade é um booleano (é declarado como int para ser usado como chave de vetor na função main, evitando a necessidade de ifs e quebras de execução) para indicar se é válido como palavra ou não ⇒ só a última letra de uma palavra inserida recebe)

Métodos:

Trie(): A root é automaticamente instanciada, com dado ‘ ’ e pai nullptr, fazendo com que estrutura nunca esteja completamente vazia (evita vários erros), mas o size continua sendo 1

~Trie(): Implementado recursivamente, chamando a função privada destructor para a raiz, que checa se o nodo enviado é nullptr, chama o destrutor para cada filho e deleta o nodo que recebeu.

Insert(): Recebe a palavra, a posição no arquivo e tamanho da linha, verifica iterativamente quais letras já estão inseridas, cria os nodos para inserir as letras que faltam até a última (como a entrada é um dicionário e apenas as palavras definidas serão inseridas, não precisa checar se uma palavra se repete, isso iria mascarar um erro no arquivo), então seta as variáveis inicio, N e is_word do último nodo para os parâmetros da função e 1 (respectivamente) e chama a função pref->update() dele para incrementar todas as letras antes dele (que agora são prefixo de uma palavra a mais)

Contains(): Recebe uma palavra, calcula o tamanho dela (SIZE) e manda ambos como parâmetro para função contains da raiz, que checa se a variável posição do nodo é menor que o comprimento da palavra (se não for, significa que chegou-se ao fim da busca e retorna-se true), cria um temporário para armazenar o resultado de find(palavra[posição]), ou seja, buscando a letra correspondente ao nodo atual em um nodo filho (já que a raiz não tem uma letra) e, se for nullptr, retorna false, se não, chama a função para o temporário, mandando a palavra e SIZE como parâmetro.

Prefix(): Recebe a palavra que se quer saber quantos prefixos tem e um vetor de 4 inteiros (nomeado saida), calcula o tamanho da palavra (SIZE) e envia, junto à palavra e o vetor, para a função prefix da raiz. Lá, assim como em contains é checado se a posição do nodo é maior que SIZE, (se não for, seta o vetor para os valores prefixos, início, N e is_word do nodo atual, respectivamente), busca-se palavra[posicao] entre os filhos, se achar chama a função para o filho com a letra, se não seta o vetor para 0, 0, 0, 0.

Função lê arquivo

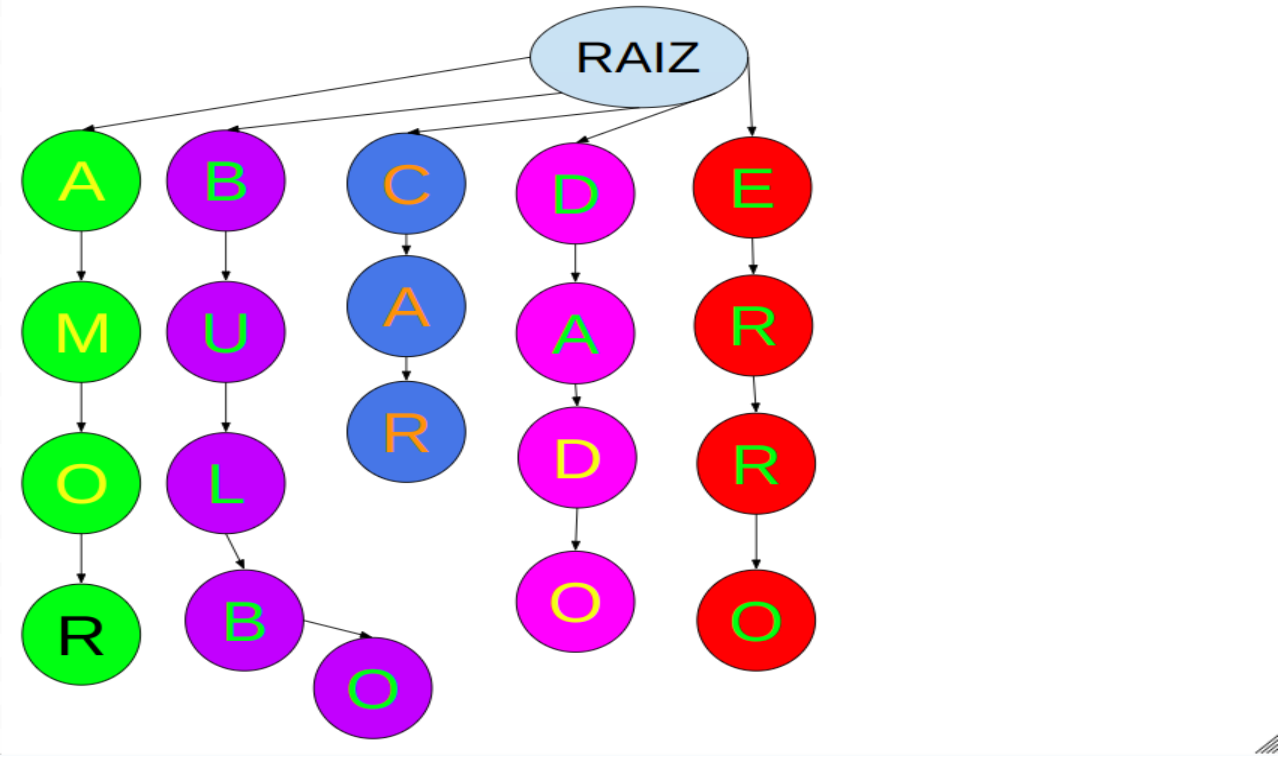
Recebe o filename e o endereço do buscador (Trie), abre o arquivo e o busca linha por linha.
A cada iteração de while(getline(arquivo, linha)) é realizado um for para identificar a palavra entre colchetes daquela linha.
A palavra é armazenando letra por letra em uma string temporária para depois ser inserida no buscador, junto aos dados de posição do arquivo e tamanho da linha em que se encontra.

Função main

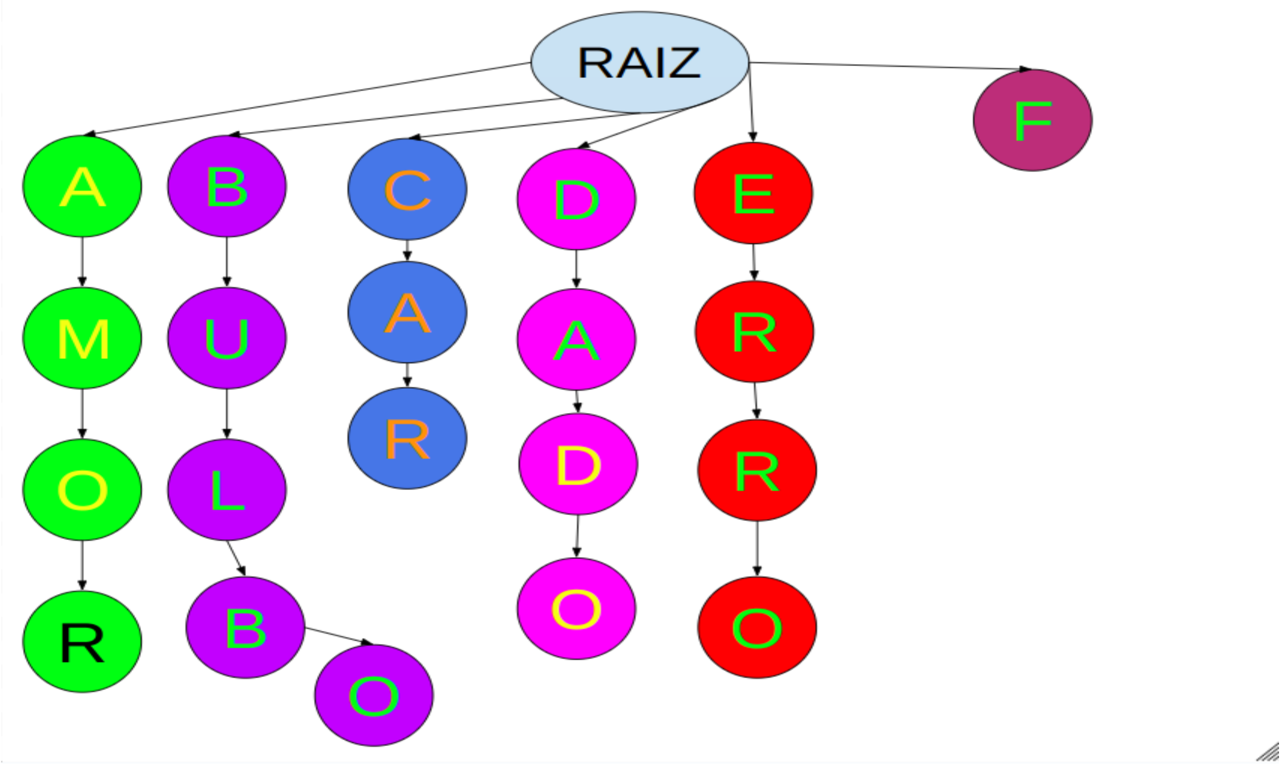
Instancia a Trie, recebe o nome do arquivo, envia ambos para a função lê arquivo e executa a lógica de receber as palavras, buscá-las na Trie e retornar os dados delas no arquivo (que já estão armazenados na Trie), preparando a saída de acordo com cada palavra (e a cada entrada também, sem recorrer a armazenar a entrada completa em um vetor e só depois preparar a saída correspondente a cada uma

Imagens

Estado Inicial:

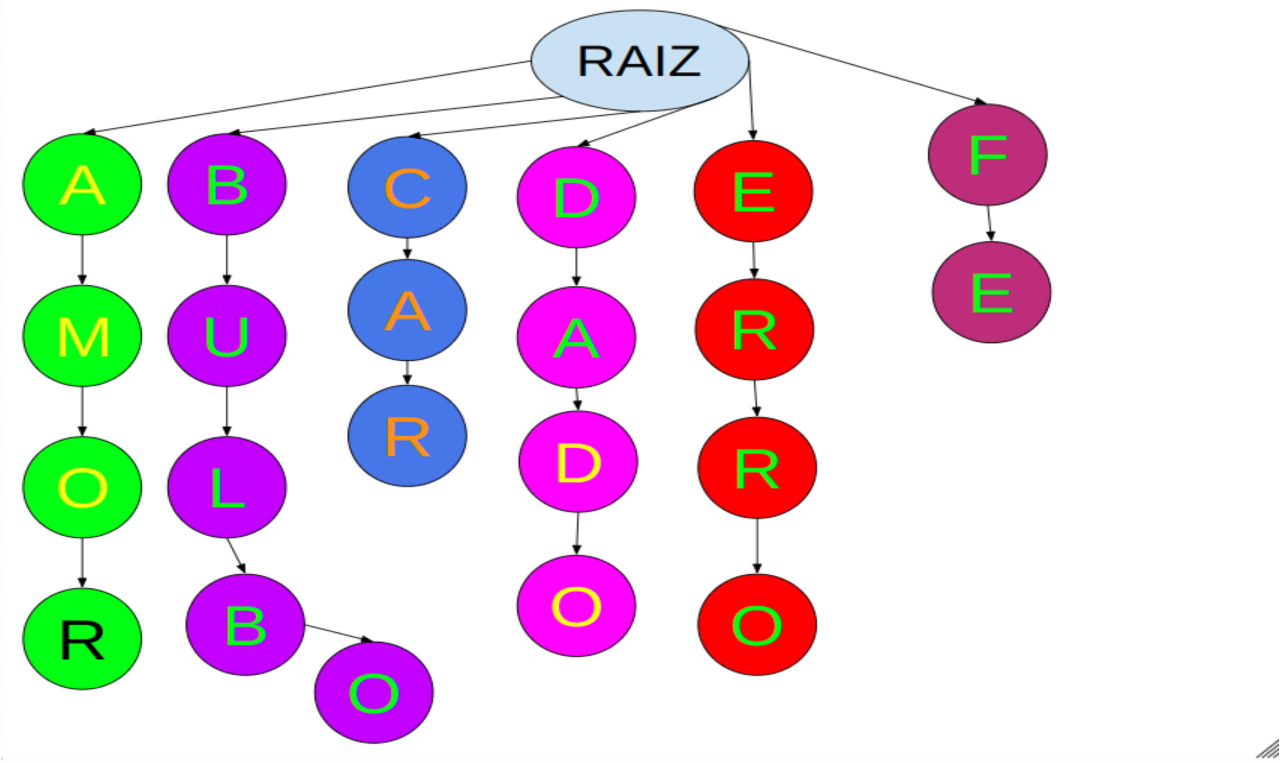


Inserindo “FEZ” desde o princípio:



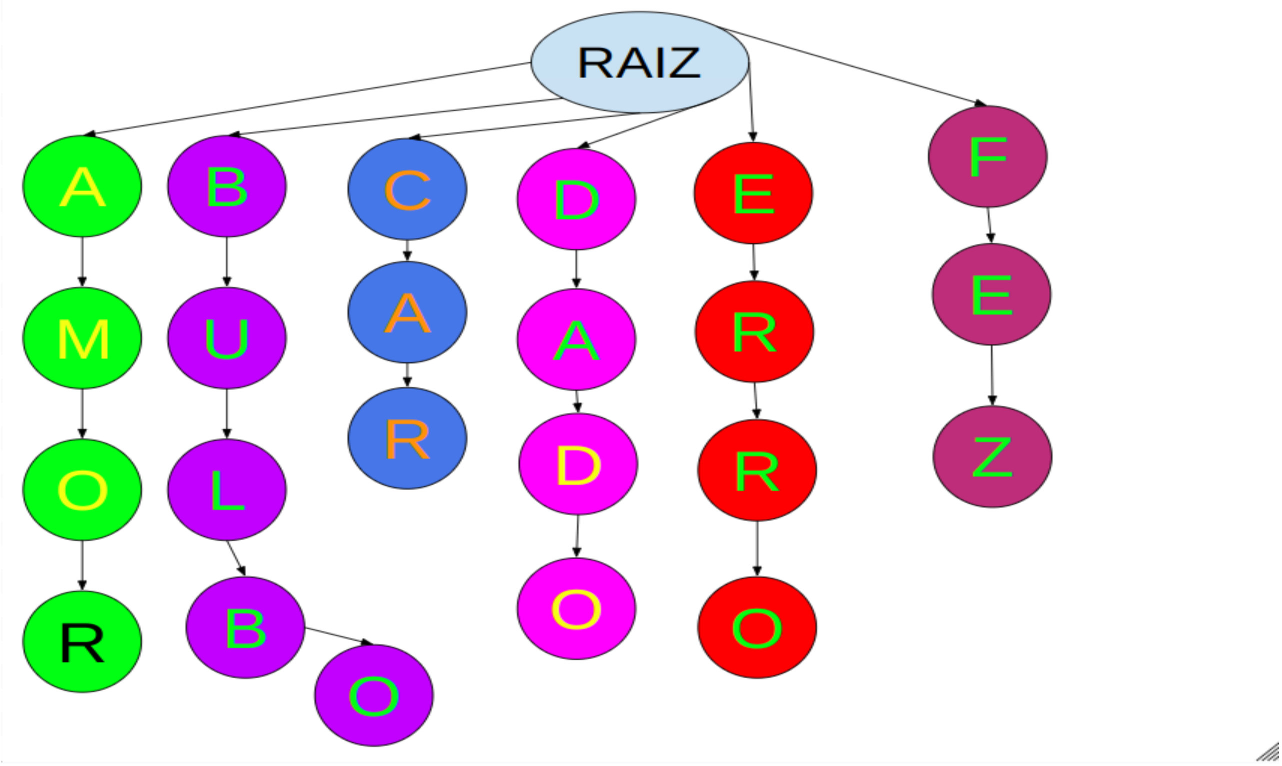
Raiz não encontra F entre os seus filhos, então o primeiro loop não é executado. Um novo nodo é criado, com a letra F e a raiz como pai.

↓↓
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓



ainda há mais duas letras na palavra, então o loop continua a criar nodos e adicioná-los um como pai do outro

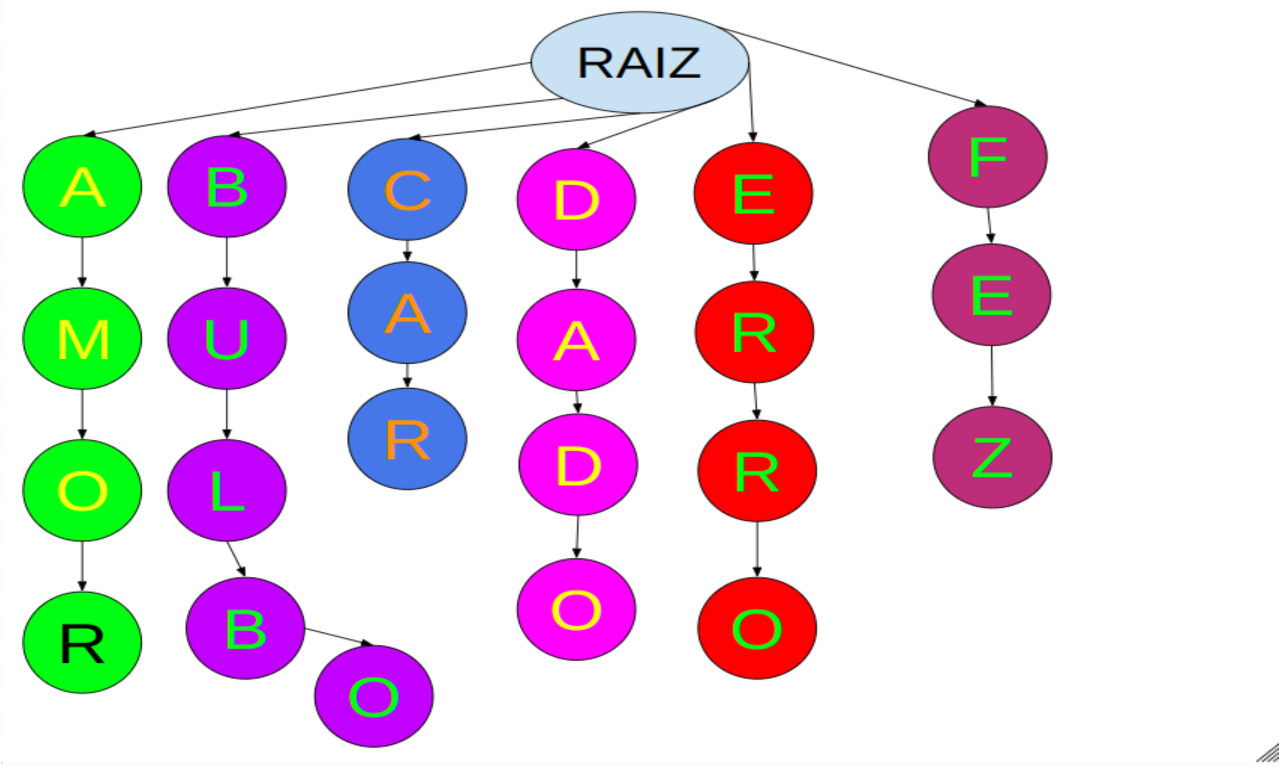
↓↓
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓



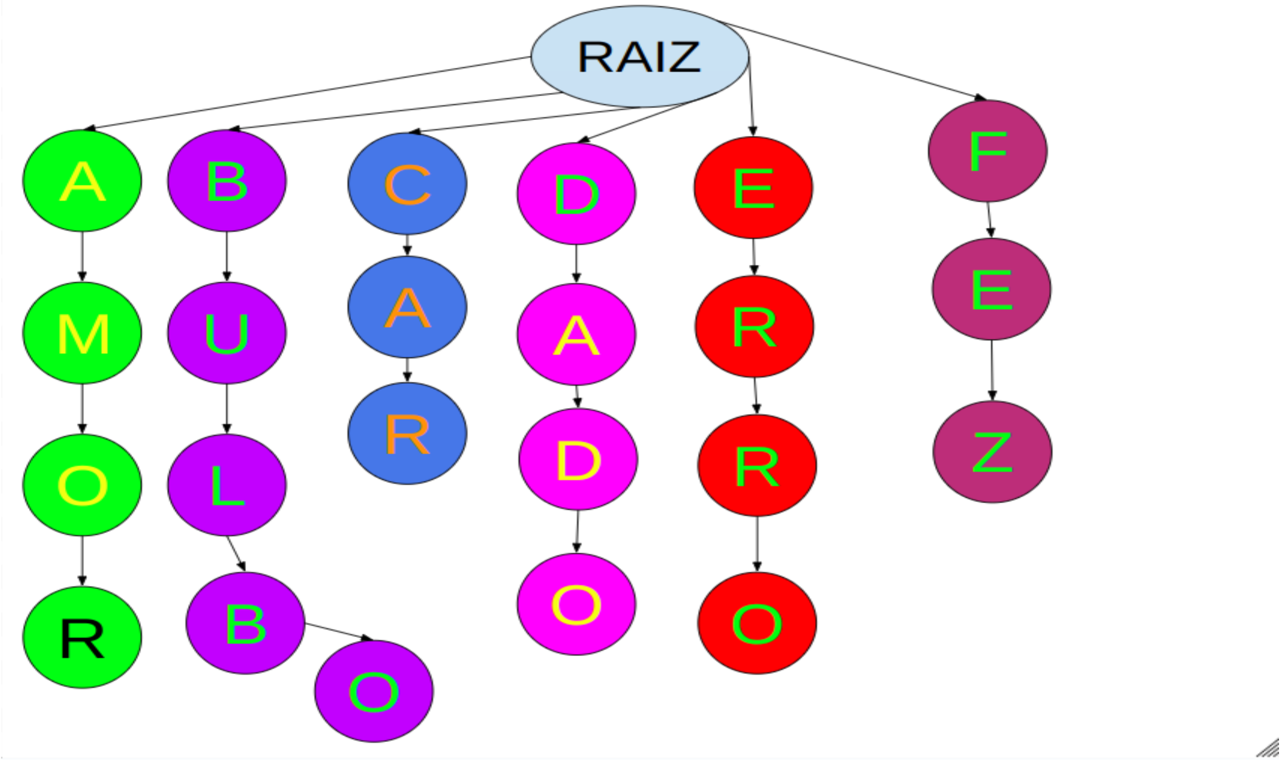
No final, o Nodo com a letra Z recebe as coordenadas de FEZ no arquivo e seu is_word é qualificado em 1. A função pref_update é chamada e Z, E e F passam a ter prefixos = 1 (são todos prefixos da palavra FEZ), assim como o número de prefixos da raiz também é incrementado (passa a ser 7)

↓↓
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

Inserindo a palavra DATA:

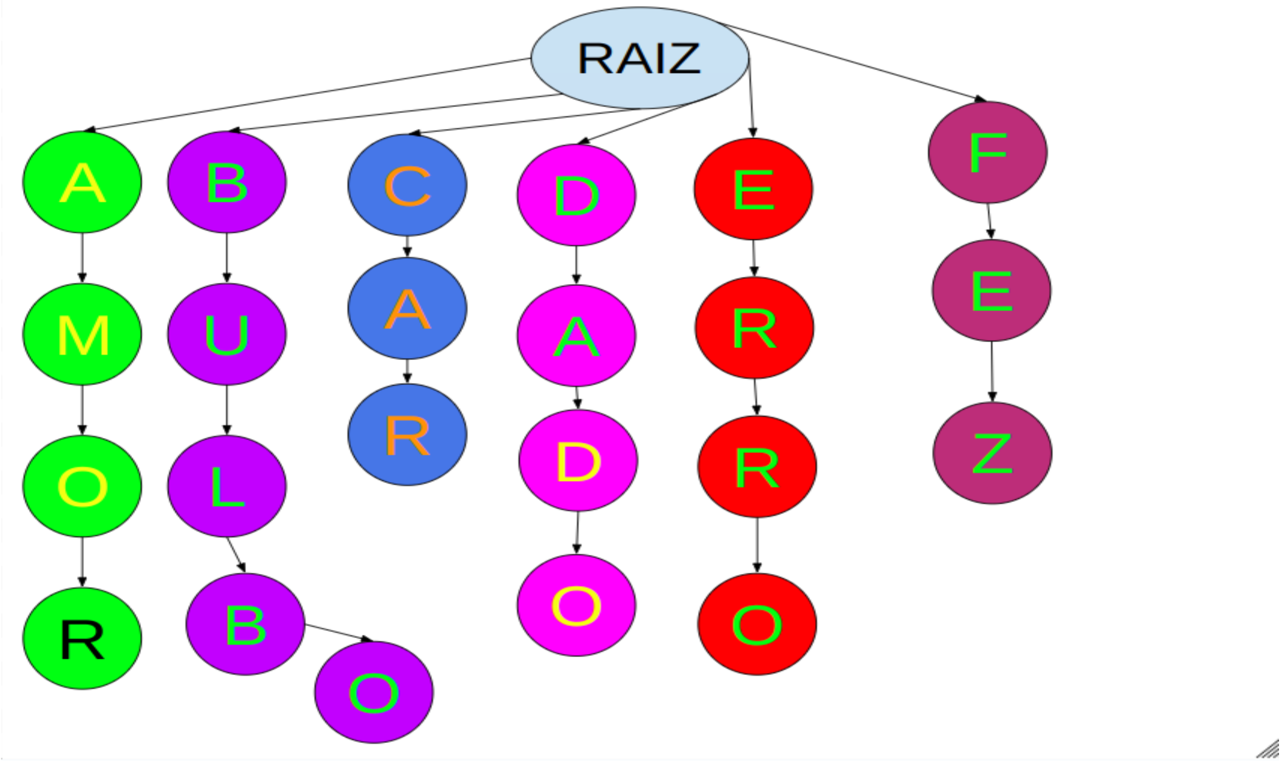


↓↓↓
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓



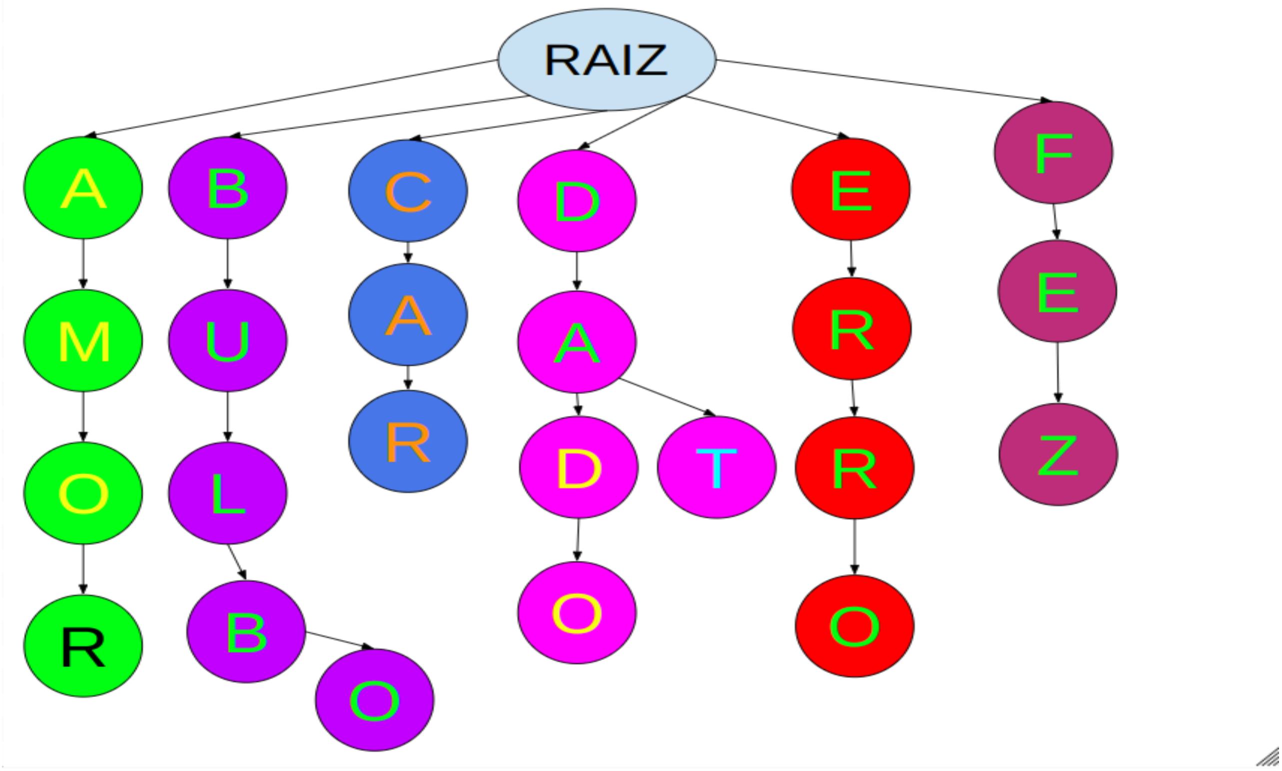
D é encontrado como filho de raiz, então o primeiro loop continua.

↓↓↓
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓



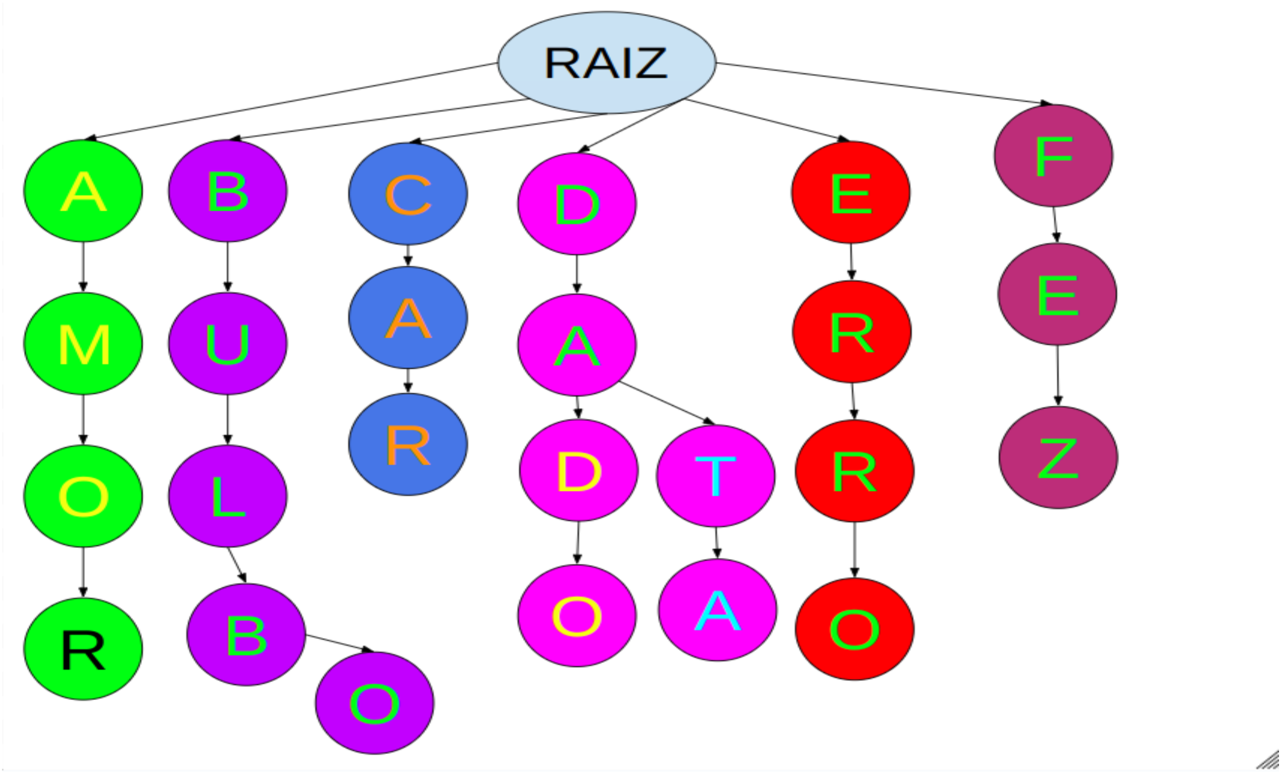
A também é encontrado entre os filhos de D, mas T não é encontrado como neto, encerrando o primeiro loop

↓↓↓
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓



Um novo nodo contendo T é criado e inserido como filho de A, mas ainda não chegou-se na última letra.

↓↓↓



Finalmente a última letra é criada e inserida, recebendo as coordenadas de DATA no arquivo e tendo o número de prefixos de A, T, A, D e Raiz incrementados (para 1, 1, 2, 2 e 8, respectivamente)

Conclusão

A principal dificuldade encontrada, ou pelo menos a última a ser resolvida, foi no método de inserção da Trie. Não foi encontrado uma maneira de reorganizar o código para incrementar i após a busca pela letra. Isso faz com que, caso uma palavra já tenha sido inserida, i seja incrementado até o comprimento da palavra e, portanto ela seja acessada em uma posição além de sua última letra (o que, testando em um compilador online, leva a string a retornar 0) e causar um acesso indevido no find da última letra (-97). Entretanto, tentativas de reorganizar a lógica para incrementar o contador após a busca da letra resultam em falha. No caso, como find irá buscar um valor em um espaço não reservado de memória, a probabilidade é que será retornado um valor inválido, mas que ainda se desviaria do previsto e poderia gerar erros em outras circunstâncias.

Referências

https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html
https://cplusplus.com/reference/string/string/