ACH2023 - Segundo Exercício-Programa
Tries

Prof. Luciano Antonio Digiampietri

Prazo máximo para a entrega: 24/11/2024

Neste segundo EP, você deve desenvolver, de forma individual, um sistema básico para gerenciar uma estrutura de dados chamada de *trie* (também conhecida como árvore digital ou árvore de prefixos).

1 O que é uma Trie

"*Em ciência da computação, uma trie, ou árvore de prefixos, é uma estrutura de dados do tipo árvore ordenada, que pode ser usada para armazenar um array associativo em que as chaves são normalmente cadeias de caracteres. Edward Fredkin, o inventor, usa o termo trie (do inglês 're**trie**val' - recuperação), porque essa estrutura é basicamente usada na recuperação de dados.*

Ao contrário de uma árvore de busca binária, nenhum nó nessa árvore armazena a chave associada a ele; ao invés disso, ela é determinada pela sua posição na árvore. Todos os descendentes de qualquer nó têm um prefixo comum com a cadeia associada com aquele nó, e a raiz é associada com a cadeia vazia. Normalmente, valores não são associados com todos os nós, apenas com as folhas e alguns nós internos que correspondem a chaves de interesse.

Não é necessário que as chaves sejam explicitamente armazenadas nos nós."[1]

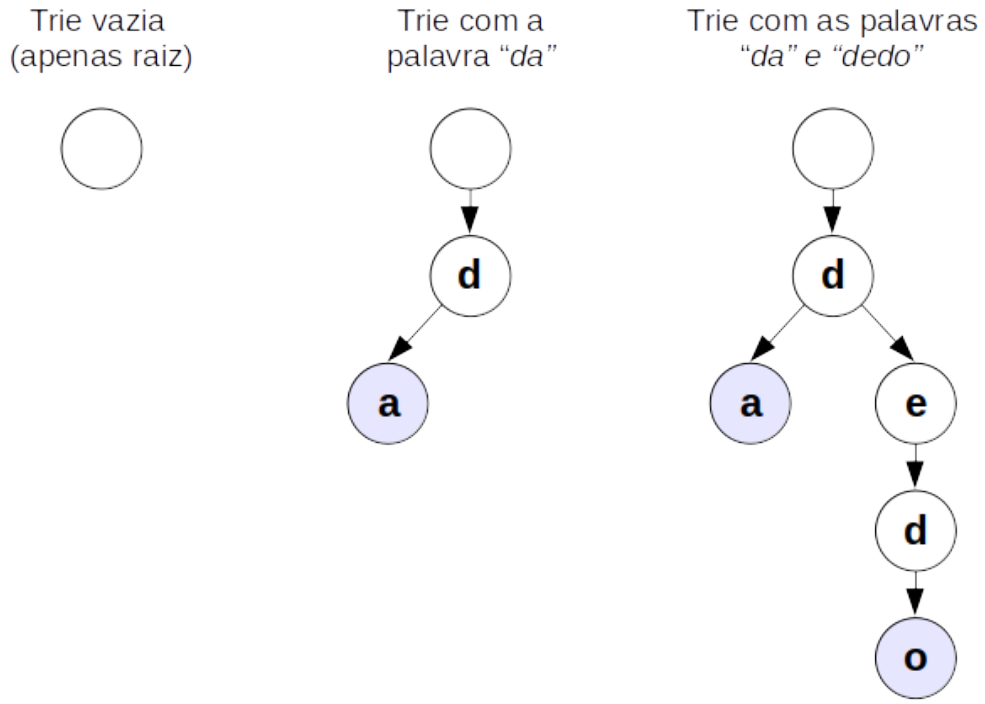
Tries são estruturas que possuem boa eficiência para gerenciar elementos representados como palavras (ou qualquer conjunto de caracteres/dígitos), podendo ser usadas no lugar de *hashs* ou para diferentes aplicações relacionadas ao Processamento de Línguas Naturais (PLN).

A figura 1 mostra um exemplo abstrato de três tries: uma sem palavras (apenas com o nó raiz), uma com a palavra "da" e outra com as palavras "da" e "dedo".

Videoaula sobre tries da disciplina Estrutura de Dados da UNIVESP (disciplina ministrada pelos professores Norton Roman e Luciano Digiampietri): `https://www.youtube.com/watch?v=zNNKeMHIq14&list=PLxI8Can9yAHf8k8LrUePyj0y3lLpigGcl&index=9`

[1] `https://pt.wikipedia.org/wiki/Trie`, acessado em 04/09/2024

Figura 1: Representação abstrata de *tries* (vazia, com uma e com duas palavras)



2 O Problema

Seu objetivo neste EP será gerenciar a inclusão, exclusão e busca por palavras (bem como gerenciar quantas vezes cada palavra está presente na estrutura).

As palavras neste EP serão formadas por apenas letras minúsculas de a até z e nenhuma palavra possuirá mais do que 1024 caracteres.

Há apenas uma estrutura (*struct*) envolvida neste EP: *NO*.

A estrutura **NO** foi projetada para armazenar os nós de nossa *trie*. Ela é composta por dois campos:

- *contador* – valor numérico que indica quantas cópias da respectiva palavra estão armazenadas;
- *filhos* – campo do tipo ponteiro para ponteiros de `NO`s que tem por objetivo armazenar o endereço do arranjo de filhos do respectivo nó ou *NULL* se o nó não possuir filhos.

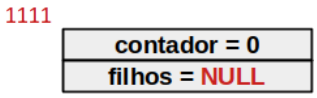
```
typedef struct aux {
 int contador;
 struct aux** filhos;
} NO, * PONT;
```

Observações: (a) Na estrutura `NO` não há um campo para indicar a que letra ele corresponde, pois isso é dado pela posição em que ele está no arranjo de filhos de seu pai. (b) Um nó que não corresponde ao fim (letra final) de uma palavra terá o valor de seu *contador* igual a zero. Por

exemplo, na figura 1 estão destacados os nós que indicam o final das palavras “da” e “dedo”. Se o usuário desejar inserir a palavra “de” na trie da direita dessa figura, nenhum nó será adicionado, mas o nó correspondente à letra “e” seria destacado (no nosso caso, o valor de seu contador seria incrementado em 1 [um]). (c) Os arranjos de filhos (ponteiros para filhos) terão sempre tamanho igual a 26 (correspondendo às 26 letras minúsculas do alfabeto, de forma que o índice zero corresponde à letra a e o índice 25 corresponde à letra z).

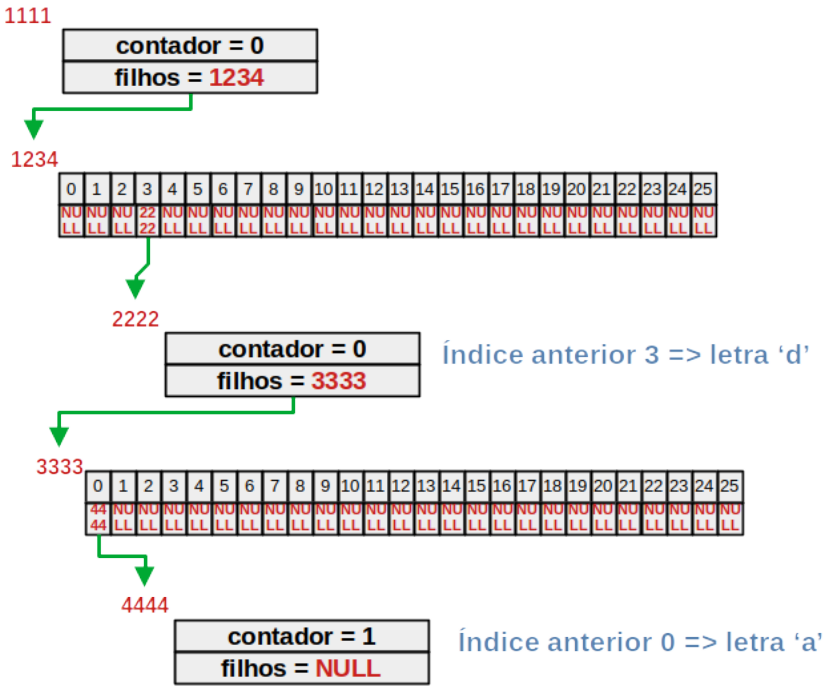
Ao criarmos um novo nó em nossa trie, não alocaremos memória para o arranjo de *filhos*, exceto se este nó tiver efetivamente filhos. A figura 2 ilustra como ficará a memória de uma trie recém inicializada (contendo apenas a raiz). O valor do campo *contador* da raiz sempre valerá 0 (zero).

Figura 2: Representação em memória de uma trie sem palavras (apenas o nó raiz). A cor vermelha indica endereços de memória



Ao inserirmos a palavra “da”, será necessário alocar memória para o arranjo de filhos da raiz, bem como criar o nó correspondente a “d”, o qual também terá um arranjo de filhos, e criar o nó correspondente à “a”, o qual não terá um arranjo de filhos e o valor de seu *contador* será igual a 1, conforme ilustrado na figura 3.

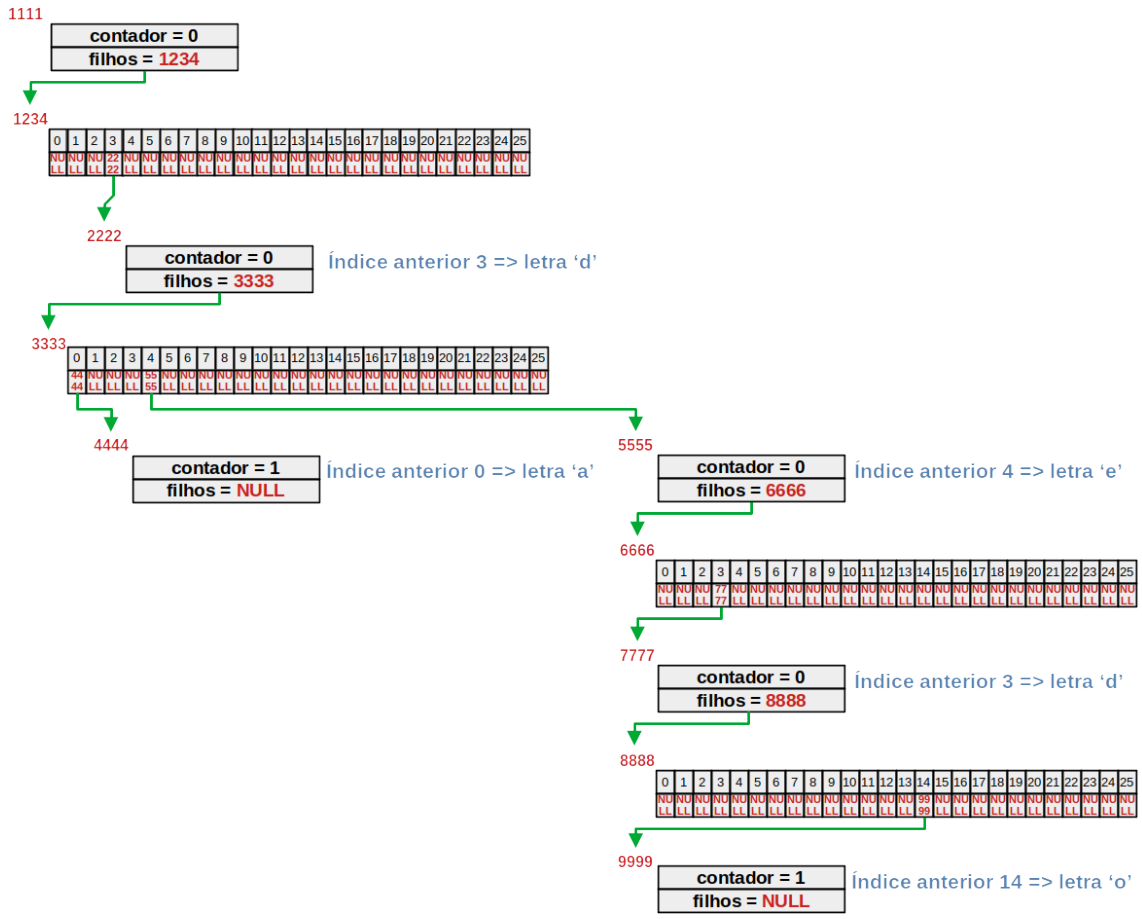
Figura 3: Representação em memória da trie contendo a palavra ‘da’. A cor vermelha indica endereços de memória, enquanto a cor azul indica valores que são deduzidos a partir da estrutura da trie



Ao inserirmos a palavra “dedo”, o nó correspondente ao primeiro “d” da palavra já existe,

será necessário criar os nós para as demais letras (e os arranjos de filhos para o nó “e” e para o nó correspondente ao segundo “d”). O nó correspondente à “o” não terá um arranjo de filhos e o valor de seu *contador* será igual a 1, conforme ilustrado na figura 4.

Figura 4: Representação em memória da trie contendo as palavras ‘da’ e ‘dedo’. A cor vermelha indica endereços de memória, enquanto a cor azul indica valores que são deduzidos a partir da estrutura da trie



O gerenciamento da Trie exige a implementação de diversas funções. Várias delas já estão implementadas no código fornecido do EP, você deve completar a implementação daquelas que estão em azul.

void inicializar(PONT raiz): função que recebe como parâmetro o endereço da raiz de uma trie e realiza a inicialização da estrutura. Isto é, atribui o valor 0 (zero) para o campo *contador* e o valor *NULL* para o campo *filhos*.

PONT criarNo(): função que aloca memória para um novo nó, inicializa seus campos (valor zero para *contador* e *NULL* para *filhos*) e retorna o endereço do novo nó.

int contarNos(PONT atual): função recursiva que recebe o endereço de um nó e retorna o número de nós da trie a partir do nó atual. Se a função for chamada a partir da raiz, retornará o número total de nós na trie.

int contarArranjos(PONT atual): função recursiva que recebe o endereço de um nó e retorna o número de arranjos de ponteiros para filhos da trie a partir do nó atual, ou seja, retorna a quantidade de nós internos (nós que não são folha) da trie. Se a função for chamada a partir da raiz, retornará o número total de arranjos na trie.

int contarPalavrasDiferentes(PONT atual): função recursiva que recebe o endereço de um nó e retorna o número de palavras diferentes da trie a partir do nó atual. Palavras diferentes significa que cada palavra será contada uma única vez. Se a função for chamada a partir da raiz, retornará o número total de palavras diferentes na trie.

int contarPalavras(PONT atual): função recursiva que recebe o endereço de um nó e retorna o número de palavras da trie a partir do nó atual. Uma palavra será contada mais de uma vez, caso o campo *contador* seja maior do que 1. Se a função for chamada a partir da raiz, retornará o número total de palavras na trie.

void exibir(PONT raiz, char* palavra): função que recebe o endereço da raiz de uma trie e o endereço de um arranjo de caracteres e imprime na tela, em ordem alfabética, todas as palavras da trie (utilizando a função *exibirAux*). O arranjo de caracteres, chamado *palavra*, é usado para compor cada palavra a partir da raiz e então imprimir essas palavras.

exibirAux(PONT atual, char* palavra, int pos): função recursiva que recebe o endereço de um nó, o endereço de um arranjo de caracteres e a posição da letra atual da palavra. Caso o nó atual corresponda à letra final de uma palavra (*contador* maior que zero) imprime a palavra atual. Se o nó atual possuir filhos, realiza uma chamada recursiva para cada um de seus filhos adicionando a letra correspondente ao seu filho no arranjo *palavra*.

int buscarPalavra(PONT raiz, char* palavra, int n): função que recebe o endereço do nó raiz de uma trie (*raiz*), o endereço de um arranjo de caracteres (*palavra*) e o tamanho da palavra presente no arranjo de caracteres (n) e retorna o número de cópias dessa palavra na trie. Provavelmente você desejará realizar a busca chamando uma função auxiliar recursiva (desenvolvida por você) que tenha, ao menos, um parâmetro adicional para indicar qual a letra atual da busca. Caso a palavra não exista na trie, sua função deverá retornar 0 (zero), caso contrário, deverá retornar o valor do campo *contador* do nó correspondente à última letra da palavra.

void inserir(PONT raiz, char* palavra, int n): função que recebe o endereço do nó raiz de uma trie (*raiz*), o endereço de um arranjo de caracteres (*palavra*) e o tamanho da palavra presente no arranjo de caracteres (n) e insere essa palavra na trie. Provavelmente você desejará realizar a inserção chamando uma função auxiliar recursiva (desenvolvida por você) que tenha, ao menos, um parâmetro adicional para indicar qual a letra atual da chamada recursiva (assumindo que a cada chamada recursiva uma letra será inserida/processada). Você pode assumir que todos os parâmetros desta função terão valores válidos e a palavra a ser inserida (presente no arranjo de caracteres) possuirá apenas letras minúsculas (sem acentos ou caracteres especiais). A inserção funciona da seguinte forma (potencialmente recursiva):

(i) se ainda falta inserir caractere(s) e o nó atual **não possui** arranjo de filhos, é necessário:

   (a) criar esse arranjo, preencher suas posições com valores *NULL* e atribuir o endereço dele no campo *filhos* do nó atual;

   (b) criar o nó correspondente ao próximo caractere, acertar seus campos (se você usar a função *criarNo* ela já acerta esses valores) e colocar o endereço dele na posição correspondente do arranjo *filhos*; e

   (c) continuar recursivamente com o processo sendo que, se você acabou de criar o último caractere da palavra, o campo *contador* deve receber o valor 1 e a inserção foi concluída.

(ii) se ainda falta inserir caractere(s) e o nó atual **possui** o arranjo de filhos: caso o nó correspondente ao próximo caractere já exista no arranjo de filhos, basta prosseguir a inserção (recursivamente) a partir dele, caso contrário é necessário prosseguir de acordo com o processo (i.b).

(iii) se você chegou ao último caractere da palavra e ele já existe na trie, é necessário incrementar seu campo *contador* em uma unidade.

void excluirTodas(PONT raiz, char* palavra, int n): função que recebe o endereço do nó raiz de uma trie (*raiz*), o endereço de um arranjo de caracteres (*palavra*) e o tamanho da palavra presente no arranjo de caracteres (n) e exclui todas as cópias dessa palavra da trie. Provavelmente você desejará realizar a exclusão chamando uma função auxiliar recursiva (desenvolvida por você) que tenha, ao menos, um parâmetro adicional para indicar qual a letra

atual da chamada recursiva (assumindo que cada chamada recursiva avançará uma letra na árvore). Adicionalmente, se a operação de exclusão resultar na eliminação de um nó, pode ser necessário, recursivamente, apagar alguns dos nós anteriores, neste caso, você pode precisar de mais um parâmetro adicional na sua função auxiliar recursiva ou ela pode ter um retorno (potencialmente booleano) para indicar que é necessário excluir nós na volta da chamada recursiva. Você pode assumir que todos os parâmetros desta função terão valores válidos e a palavra a ser excluída (presente no arranjo de caracteres) possuirá apenas letras minúsculas (sem acentos ou caracteres especiais). É possível que a palavra não exista na sua trie, então a função de exclusão não causará nenhuma mudança na trie. A exclusão funciona da seguinte forma (potencialmente recursiva):

- Você navegará na trie, caractere a caractere, até passar por toda a palavra, isto é, chegar no caractere final da palavra.

(i) Se isto não for possível, isto é, se a sequência de caracteres que forma a palavra não está presente na trie, sua função deverá encerrar sem realizar nenhuma mudança na estrutura; caso contrário, há duas possibilidades principais:

(ii) A palavra não existe na trie (isto é, apesar de suas letras estarem lá, o campo *contador* da última letra é igual a zero), neste caso, sua função deve encerrar sem modificar a estrutura;

(iii) A palavra existe na trie, neste caso ela dever ser excluída, considerando duas situações:

(a) O nó correspondente à última letra possui filhos, neste caso o *contador* dele deverá ser zerado e não há mais nada a ser feito pela função;

(b) Caso o nó correspondente à última letra não possua filhos ele deverá ser excluído, e o ponteiro para ele no arranjo de filhos de seu pai deve ser atualizado para *NULL*. Se após essa exclusão, o pai desse nó não possuir mais filhos, seu arranjo de filhos deve ser excluído (memória liberada) e seu campo *filhos* deve receber o valor *NULL*. Adicionalmente, se o campo *contador* do nó pai valer zero este também deve ser apagado (e o processo iii.b deve ser repetido enquanto cada nó [na volta da recursão] não possuir mais filhos e não for um nó final de uma palavra).

- Observação: o nó raiz nunca deverá ser excluído, porém seu arranjo de filhos poderá ser excluído caso este nó não possua filhos (trie sem nenhuma palavra) e, neste caso, seu campo *filhos* deverá ser atualizado para *NULL*.

void excluir(PONT raiz, char* palavra, int n): função que recebe o endereço do nó raiz de uma trie (*raiz*), o endereço de um arranjo de caracteres (*palavra*) e o tamanho da palavra presente no arranjo de caracteres (n) e **exclui uma cópia** dessa palavra da trie (isto é, caso a palavra exista na trie, diminui o *contador* correspondente a sua última letra em uma unidade). Observações: se a palavra não existir na trie, não há nada a ser feito pela função; se a palavra existir e o *contador* valer 1 (um) antes da exclusão, então a exclusão terá o mesmo comportamento da função *excluirTodas*.

2.1 Material a Ser Entregue

Um arquivo, denominado `NUSP.c` (sendo NUSP o seu número USP, por exemplo: 123456789.c), contendo seu código, incluindo todas as funções solicitadas e qualquer outra função adicional que ache necessário. Para sua conveniência, `completeERenomeie.c` será fornecido, cabendo a você então completá-lo e renomeá-lo para a submissão.

Atenção!

1. Não modifique as assinaturas das funções já implementadas e/ou que você deverá completar!
2. Para avaliação, as diferentes funções serão invocadas diretamente (individualmente ou em conjunto com outras funções). Em especial, qualquer código dentro da função `main()` será ignorado.

3 Entrega

A entrega será feita única e exclusivamente via sistema e-Disciplinas, até a data final marcada. Deverá ser postado no sistema um arquivo .c, tendo como nome seu número USP:

```
seuNumeroUSP.c   (por exemplo, 12345678.c)
```

Não esqueça de preencher o cabeçalho constante do arquivo .c, com seu nome, número USP, turma etc.

A responsabilidade da postagem é exclusivamente sua. Por isso, submeta e certifique-se de que o arquivo submetido é o correto (fazendo seu download, por exemplo). Problemas referentes ao uso do sistema devem ser resolvidos com antecedência.

4 Avaliação

A nota atribuída ao EP será baseada nas funcionalidades solicitadas, porém não esqueça de se atentar aos seguintes aspectos:

1. Documentação: se há comentários explicando o que se faz nos passos mais importantes e para que serve o programa (tanto a função quanto o programa em que está inserida);
2. Apresentação visual: se o código está legível, identado etc;
3. Corretude: se o programa funciona.

Além disso, algumas observações pertinentes ao trabalho, que influenciam em sua nota, são:

- Este exercício-programa deve ser elaborado individualmente;
- Não será tolerado plágio;
- Exercícios com erro de sintaxe (ou seja, erros de compilação), receberão nota ZERO.