Universidade Federal de Viçosa
Campus Florestal

TRABALHO PRÁTICO 01

Ana Clara Antunes Rocha - 4218
Tássia Martins Almeida Gomes - 4247
Thulio Marcus Santos Silva - 4223

Florestal
2022

SUMÁRIO

1. INTRODUÇÃO

Um índice invertido pode ser definido como uma estrutura que contém uma entrada para cada palavra que aparece em, pelo menos, um documento. Neste trabalho, foi pedido que o grupo implementasse um índice invertido através de uma Árvore Patrícia e uma Tabela Hash. O trabalho, e o índice invertido, têm como objetivo permitir pesquisas rápidas de um texto, a um custo relativamente mais baixo.

Para realizar o projeto, o grupo realizou, primeiramente, discussões acerca do entendimento do trabalho, além de buscar referências juntamente com outro grupo (Gabriel Bonfim, Douglas Diones e Patrick de Oliveira). Em seguida, foram utilizados alguns códigos como referência, sendo eles: o código disponibilizado em sala, do professor Nivio Ziviani; o código de Árvore Patrícia do aluno Kristtopher Kayo de TP's passados; e o código de Árvore Patrícia e Tabela Hash do aluno Lázaro Bodevan, também de TP's passados.

Para finalizar, o grupo utilizou também como referência algumas funções encontradas na internet, sendo elas: tolower() e ispunct(), referenciadas ao final deste documento.

2. METODOLOGIA E DETALHES TÉCNICOS

Inicialmente, o grupo realizou encontros de forma presencial e online a fim de entender o trabalho e dividir tarefas.

Quanto à implementação, primeiramente foram criadas funções para a leitura do arquivo: lerArquivoPrincipal e lerArquivosDentroDoArqPrincipal. O primeiro é responsável por ler o arquivo que contém os nomes dos arquivos existentes para teste. Já o segundo, é responsável por abrir o arquivo que de fato contém frases para teste.

Quando os arquivos que contém as frases é aberto, cada caracter é passado para a forma minúscula usando a função tolower() e todas as pontuações são retiradas utilizando a função ispunct().

Em seguida, o usuário escolhe qual estrutura deseja usar para inserir as palavras. Independentemente da estrutura escolhida, o índice invertido é criado no momento da inserção.

O índice invertido foi criado a partir de uma lista encadeada, na qual as células são compostas pela quantidade de vezes que a palavra se repete e o id do documento em que ela se encontra, podendo este ser mais de um. Dessa forma, no código, no momento da inserção é primeiramente avaliado qual documento está sendo lido. Caso o documento já tenha sido lido, e a palavra já foi inserida uma vez, apenas incrementa-se um na variável “qntd_repete”. Se o documento ainda não tiver sido lido, cria-se uma nova célula com o novo id e é feito todo o processo de análise novamente.

. Dividimos o trabalho em arquivos .h e .c, sendo eles: NossaPaty.h e NossaPaty.c, novaHash.h e novaHash.c, indiceHash.h e indiceHash.c, IndicePaty.h e IndicePaty.c, e a main.c.

2.1) NovaHash.h e NovaHash.c

Como pedido, o grupo implementou a tabela Hash a partir do endereçamento fechado, ou seja, implementando lista encadeada dentro da tabela. Por isso, definimos no arquivo .h, as seguintes structs: TipoItem, TipoCelula e TipoLista.

A struct TipoItem contém a Chave do TipoPalavra, sendo definida do tipo char. A struct TipoCelula possui um ItemH, uma lista de ocorrências para que o índice invertido seja feito e um Apontador para o próximo Item. Já a struct TipoLista, contém um ponteiro para a primeira e um ponteiro para a última posição na lista.

Nestes arquivos também foram declaradas e desenvolvidas todas as funções necessárias para a Tabela Hash. Sendo as mais importantes:

void InicializaListaVazia(TipoLista *Lista): Esta função é responsável por alocar memória para a primeira posição da lista e fazer com que a primeira posição seja igual a segunda, fazendo com que as duas sejam iguais a NULL.

void InsereHash(int idDoc,char *chave, TipoLista *Lista): Nesta função, insere-se a palavra na Tabela Hash. A estratégia utilizada é que cada palavra é inserida em uma posição aleatória da lista encadeada, posição esta gerada a partir da função GeraPesos() e h().

É feita uma alocação dinâmica de memória para a próxima posição vazia da Lista e esta passa a ser a última posição da Lista. Dessa forma, esta posição recebe uma nova chave.

É nessa função, também, que se insere a palavra no índice invertido.

TipoApontador buscarNaLista(char *palavra, TipoDicionario table): Inicialmente, a variável aux é atribuída como a primeira posição da lista. Se a chave analisada for igual a palavra buscada, a função retorna esta palavra. Caso contrário, a variável aux recebe a próxima posição. Esta comparação é feita enquanto a variável aux for diferente de NULL.

void VerificaInsere(int num_arquivo, char *chave ,TipoPesos peso, TipoDicionario dicionario): Esta função é responsável por verificar em cada linha da tabela Hash, que não seja igual a NULL, se a palavra já existe na lista. Se não existir, é atribuído o valor 1 na variável “qntd_repete” na lista de ocorrências.

void BuscaPesquisa(TipoDicionario dicionario, char *buscaPalavra): Esta função verifica se cada linha da tabela Hash é vazia e faz a busca pela palavra a partir da função buscarNaLista. Caso a função retorne a palavra, imprimimos a mensagem que achamos a chave e imprimimos o índice invertido.

2.2) NossaPaty.h e NossaPaty.c

A árvore Patrícia foi implementada a partir de algumas structs chaves, denominadas de TipoNo, TipoPatNo *TipoArvoreApontador e TipoPatNo.

A struct TipoNo, é um enum, e possui a função de verificar se o nó é do tipo interno ou externo. Logo após, criamos um Apontador que aponta para o TipoPatNo, que é a struct que representa os nós da árvore, podendo assumir dois valores com o union: o tipo de cada nó (externo ou interno) e uma struct No, que possui um índice, uma letra a ser comparada e um apontador para esquerda e para a direita.

Para a Árvore Patrícia, foram criadas algumas funções importantes, sendo elas:

void InicializaArvorePatricia(TipoArvoreApontador *arvore): Nessa função, um ponteiro para a árvore recebe NULL.

TipoArvoreApontador InicializaNoInterno(TipoArvoreApontador *esq, TipoArvoreApontador *dir, char letra, int indice): Inicialmente, cria-se uma variável do tipo Apontador, denominada ArvoreAux e aloca-se memória para ela, ou seja, cria-se um nó. Logo após, define-se que ele é do tipo Interno e são atribuídos os ponteiros para a direita e para a esquerda do nó, além de atribuir seu índice e uma letra.

TipoArvoreApontador InicializaNoExterno(TipoArvoreApontador *esq, TipoArvoreApontador *dir, char letra, int indice): Para a criação do nó externo,

segue o mesmo padrão do nó interno, porém a diferença é que também aloca-se memória para a palavra que pertence ao nó externo. Além disso, o nó externo não possui apontadores para a esquerda e para a direita, como o nó interno.

É nessa função, também, que se insere no índice invertido a partir da Árvore Patricia.

TipoArvoreApontador InserePatricia(TipoArvoreApontador *arvore, char *palavraInserir, int idDoc): Nesta função, analisamos duas condições. Caso a árvore seja nula, criamos um nó externo e inserimos a palavra. Caso a árvore exista, verificamos se o nó é interno e, caso seja, verificamos se o índice do nó é maior que o tamanho da palavra, se sim, a palavra é adicionada. Caso não seja, a letra recebe a letra que difere as duas palavras e, se a letra for maior, olhamos para a árvore da direita. Se for menor, olhamos para a árvore da esquerda.

Se a palavra já existir e se não encontrar o arquivo na lista de ocorrências, criamos uma nova célula no índice invertido para adicionar uma nova lista da palavra que está sendo repetida e, assim, inserimos no índice invertido.

Se a palavra não existir na árvore, inserimos na árvore e atribuímos valor “1” ao índice, que verifica se as letras são iguais. Caso seja, incrementa-se o valor do índice. Se as letras forem diferentes, comparamos para ver qual é maior, caso a letra da palavra que queremos inserir seja maior, a letra que difere receberá a letra da palavra que queremos inserir. Se for menor ou igual, a letra que difere vai receber a letra da palavra que está no nó.

TipoArvoreApontador InsereEntrePatricia(char*palavra, TipoArvoreApontador *arvore, int indice, char letraQueDifere, int idDoc): Inicialmente, verifica-se se o nó analisado é externo. Se sim, condicionamos se a letra da palavra a ser inserida é maior que a letra da palavra que já está no nó. Se sim, criamos um nó interno com uma árvore à esquerda e uma auxiliar à direita. Caso contrário, criamos um nó interno com uma árvore auxiliar a esquerda e uma árvore à direita.

Se o nó não for externo e o índice for menor que o índice analisado, caso o índice da letra que difere for menor que o índice do nó interno, criamos um externo. Se a letra for maior ou igual, criamos um nó interno com a árvore à esquerda e a árvore auxiliar à direita. Se for menor, criamos um nó interno com uma árvore auxiliar à esquerda e a árvore à direita.

Se o nó não for externo e não atender a condição acima, e se o índice for maior ou igual ao índice do nó interno, analisamos se a letra da palavra analisada é maior ou igual a letra comparada e chamamos recursivamente o "InsereEntrePatricia" para o nó direito. Caso contrário, chamamos recursivamente o "InsereEntrePatricia" para o nó esquerdo.

int BuscaPatiElndice(TipoArvoreApontador arvore, char *palavraBusca): Primeiramente, verificamos se a árvore existe. Caso ela seja diferente de NULL, analisamos se o nó é externo. Caso seja, comparamos com a palavra buscada, e se for igual, printamos a chave. Caso o nó seja interno, percorremos recursivamente tanto para a árvore à esquerda, quanto para a árvore à direita.

2.3) indiceHash.h e indiceHash.c

Como já foi mencionado, o Indice Invertido foi implementado a partir de uma lista encadeada. No seu arquivo .h, temos a struct TipoItemQntd, que possui uma variável para a quantidade de vezes que a palavra repete e o id do documento em que ela se encontra. Também foi criada a struct TipoCelulaQntd, que possui um Item e um Apontador do TipoCelulaQntd para a próxima célula. E, por último, temos a struct TipoListaQntd, que possui um Apontador para a primeira e a última posição.

Como funções importantes, temos:

void FLVaziaQntd(TipoListaQntd *Lista): Esta função é responsável por alocar memória para a primeira posição da lista e fazer com que a primeira posição seja igual a última, fazendo com que as duas sejam iguais a NULL.

void InsereQntd(int idDoc, int qntd_repete, TipoListaQntd *Lista): Nesta função, alocamos memória para a próxima célula vazia da lista e atribuímos os valores do qntd_repete e o id do documento à célula.

int confereIdDocHash(TipoListaQntd Lista, int idDoc): Nesta função, declaramos um Apontador ApHash e atribuímos a ele o valor da próxima posição da lista. Se o id do Documento já foi analisado, incrementamos a variável qntd_repete.

2.4) indicePatricia.h e indicePatricia.c

Da mesma forma, como foi implementado na Tabela Hash, o índice invertido da Árvore Patrícia também é uma lista encadeada. No seu arquivo .h, temos a struct ItemIndicePat, que possui uma variável para a quantidade de vezes que a palavra repete e o id do documento em que ela se encontra. Também foi criada a struct CelIndicePat, que possui um Item e um Apontador do ApIndicePat para a próxima

célula. E, por último, temos a struct ListaIndicePat, que possui um Apontador para a primeira e a última posição.

As funções em destaque para o índice invertido da Árvore Patrícia, são:

void FazListaIndiceVazia_Pat(ListaIndicePat *Lista): Nesta função, alocamos memória para a primeira célula da lista, e a iguala a última posição, sendo igual a NULL.

void InsereIndice_Pat(int idDoc, int qntd_repete, ListaIndicePat *Lista): Inicialmente, alocamos memória para a próxima posição vazia da lista, e fazemos a atual última posição da lista apontar para a próxima célula. Dessa forma, inserimos a variável qntd_repete e o id do documento. Por último, tornamos a próxima célula, a última da lista.

int confereIdDoc(ListaIndicePat Lista, int recebe_iDoc): Nesta função, declaramos um Apontador Ap e atribuímos a ele o valor da próxima posição da lista. Se o id do Documento já foi analisado, incrementamos a variável qntd_repete e continuamos percorrendo a lista.

void BuscaIndice(ListaIndicePat Lista, char *valor_busca): Esta função percorre a lista para encontrar quantas vezes a palavra escolhida se repete e em qual arquivo a palavra repete.

2.5) Medição de Tempo

A partir da biblioteca <time.h>, utilizamos a função clock e obtivemos o tempo de execução antes das operações. Após a execução, obtivemos novamente o tempo de execução e subtraímos pelo tempo registrado ao início da execução. Dessa forma, obtemos o tempo de execução total do código.

3. RESULTADOS E DISCUSSÕES

O grupo reuniu as informações medidas acerca do tempo de execução dos códigos e chegamos ao seguinte resultado:

Para o tamanho da Tabela Hash sendo igual a 13:

Operação	Tempo
Inserção	0,71 milissegundos
Impressão	0,2 milissegundos

Para o tamanho da Tabela Hash sendo igual a 33:

Operação	Tempo
Inserção	1,7 milissegundos
Impressão	0,12 milissegundos

Para a Árvore Patrícia:

Operação	Tempo
Inserção	0,014 milissegundos
Impressão	0,075 milissegundos

Desse modo, podemos perceber que as operações de inserção e impressão se deram de forma mais rápida na Árvore Patrícia, comparado aos dois tamanhos de Tabela Hash.

4. CONCLUSÃO

Ao final do trabalho, o grupo concluiu que é possível afirmar o número de linhas foi maior na Tabela Hash, no código desenvolvido, a execução se torna mais lenta, uma vez que é necessário percorrer todas as linhas na função de Busca. Porém, importante lembrar que o tempo pode variar de acordo com o tamanho do texto a ser inserido, já que haveria um maior espalhamento, evitando colisões.

Também concluímos que a Árvore Patrícia possui a inserção e a impressão mais rápida em comparação às tabelas Hash analisadas, de acordo com a medição.

O trabalho foi de grande aprendizado, já que foi possível ver uma aplicação real dos conceitos aprendidos em sala de aula, acerca das Tabelas Hash, Árvore Patrícia e Índice Invertido. Porém, o grupo não conseguiu concluir a implementação da impressão por ordem alfabética da Tabela Hash e o código de relevância de

documentos (Term frequency – Inverse Document Frequency), por mais que tenha sido tentado.

5. REFERÊNCIAS

GeeksForGeeks. ispunct() function in C. Disponível em: https://www.geeksforgeeks.org/ispunct-function-c/. Acesso em 20 de junho de 2022.

Gaspar, Wagner. Como Converter Qualquer String em Maiusculo ou Minusculo SO com a linguagem C. Disponível em: https://wagnergaspar.com/como-converter-uma-string-em-maiusculo-ou-minusculo-em-qualquer-so-com-a-linguagem-c/#:~:text=Para%20isso%20podemos%20utilizar%20as,retorna%20sua%20vers%C3%A3o%20em%20min%C3%BAsculo. Acesso em 16 de junho de 2022.

Ziviani, Nizio. Projeto de Algoritmos com Implementações em Pascal e C. Disponível em: https://www2.dcc.ufmg.br/livros/algoritmos/cap5/codigo/c/5.16a5.21-patricia.c. Acesso em 16 de junho 2022.

Ziviani, Nizio. Projeto de Algoritmos com Implementações em Pascal e C. Disponível em: https://www2.dcc.ufmg.br/livros/algoritmos/cap5/codigo/c/5.22a5.27-hash-listas.c. Acesso em 16 de junho de 2022.