Universidade Federal De Viçosa (UFV) - Campus Florestal
Disciplina: Algoritmos e Estrutura de Dados II (AEDs II) - CCF 212
Professor(a): Glaucia Braga e Silva

Trabalho prático 1
APLICAÇÃO COM ÁRVORES DIGITAIS E TABELAS HASH

Miguel Ribeiro [EF04680]
Alan Gabriel [EF04663]
Vinicius Gontijo [EF04708]
Gabriel Ryan Dos Santos Oliveira [EF04688]

miguel.a.silva@ufv.br, *alan.g.silva@ufv.br* , *vinicius.a.gontijo@ufv.br*,
gabriel.oliveira1@ufv.br

Introdução

O trabalho tem como principal problema a construção de índice invertido para máquinas de busca.

Máquinas de busca, tais como Google, trabalham com a busca de palavras-chave em textos armazenados na Web. Nesse contexto, são utilizadas estruturas de dados que facilitem a recuperação das informações, como é o caso do uso de arquivos invertidos.

O trabalho prático de Estrutura de Dados II consiste na aplicação de árvores digitais e tabelas hash, para armazenar dados e posteriormente criar um mecanismo de busca para recuperar e localizar essas informações. Os dados serão palavras contidas em vários arquivos diferentes, e armazenadas usando a técnica de ponderação TF-IDF (Frequência de Termo – Frequência de documentos inversa). Mostrando a relevância da palavra em cada um dos documentos.

https://github.com/FurlanV/Maquina-de-Busca-com-Patricia (Git Hub de Vinicius Furlan
https://github.com/topics/patricia-tree (Diversas implementações da árvore Patricia, várias delas auxiliaram na nossa implementação)
http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes-05.php (Implementações do livro do Ziviani, Projeto de Algoritmos com Implementações em Pascal e C)

Metodologia

Após a formação do grupo, destacamos as principais tarefas a serem cumpridas para a realização do trabalho prático. O mesmo foi dividido da seguinte forma:

-Implementação da árvore Patricia e Menu - Miguel Ribeiro

-Documentação, caso de testes e outras funções - Gabriel Ryan

-Implementação da Tabela Hash - Vinicius e Alan

O código fonte foi desenvolvido em C e versionado no GitHub, visando que seria a melhor forma de compartilhamento do mesmo entre os integrantes do grupo. Para uma melhor organização e visualização do projeto, este foi dividido em subpastas.

-**./arquivos -** contém todos os arquivos de testes usados

-**./cores -** funções para personalizar os outputs

-**./remove -** script em python usado para converter caracteres especiais em um texto

-./tads - implementação dos tads Patricia, Hash, Bst e LinkedList

Os cenários de teste planejados para comparar as estruturas de dados patricia e hash, na criação do índice invertido e consultas aos termos de busca foram …

Detalhes Técnicos de Implementação

Para o continuar do desenvolvimento do trabalho prático foram feitas alterações, tanto na patricia quanto na hash, estas que foram de suma importância para que podermos dar seguimento no projeto, tais como:

PATRICIA

O algoritmo fornecido em sala de aula foi adaptado para a inserção de caracteres nos nós da árvore.

A começar pela criação dos estruturas necessárias para o funcionamento correto da árvore:

```
typedef struct No_Patricia{
  TipoNo TipoNo;
  Miguel_Ribeiro, 2 weeks ago | 1 author (Miguel_Ribeiro)
  union {
    Miguel_Ribeiro, 2 weeks ago | 1 author (Miguel_Ribeiro)
    struct {
      char caractere;
      int posicao;
      struct No_Patricia *esq;
      struct No_Patricia *dir;
    } interno;
    Miguel_Ribeiro, 2 weeks ago | 1 author (Miguel_Ribeiro)
    struct {
      char *palavra;
      Lista_Encadeada *lista;
    } externo;
  }No;
} No_Patricia;
```

Lista encadeada na estrutura do nó, para armazenar os índices invertidos e campos para armazenar variáveis do tipo char

Algumas funções também foram adaptadas e outras, criadas, como podemos ver abaixo:

```c
int Inicializa_Patricia(No_Patricia **no);
int Externo(No_Patricia **no);
int Confere(No_Patricia **no, char *palavra);
char Retornar_Caractere(No_Patricia **no, int posicao);
int Retornar_Posicao(No_Patricia **no);
int Cria_No_Externo(No_Patricia **no, char *palavra, int arquivo);
int Cria_No_Interno(No_Patricia **no, char caractere, int posicao, No_Patricia **esq, No_Patricia **dir)
int No_Diferente(No_Patricia **no, char *palavra);
int Insere_Entre(No_Patricia **no, char *palavra, int posicao, char caractere, int filename);
int Insere_Palavra(No_Patricia **no, char *palavra, int filename);
int Printar_Palavra(No_Patricia **no);
int Printar_Ocorrencias(No_Patricia **no);
int Incrementa_Ocorrencias(No_Patricia **no, int filename);
Lista_Encadeada *Buscar_Palavra(No_Patricia **no, char *palavra);
int Palavras_Diferentes_PAT(No_Patricia **no, int idArq);
int Free_Patricia(No_Patricia **no);
```

Externo() - esta função verifica qual o tipo do nó.
Confere() - verifica se o caractere do nó interno é maior ou menor que o da posição, usando o tamanho da palavra.
Retornar_Caractere() - retorna o caractere de uma posição específica
Retornar_Posicao() - retorna a posição de um nó interno
No_Diferente() - retorna a posição do i-ésimo caractere que se difere da palavra que queremos inserir da que já está inserida
Insere_Entre() - esta função faz diversas verificações, a fim de inserir a palavra no local exato da árvore e fazer modificações nas já inseridas para ficarem em seus exatos lugares. A **primeira** e verifica se o nó é externo, se for, cria um novo nó externo com a palavra a ser inserida. Após isso ela chama a função Retornar_Caractere a fim de retornar o caractere da posição que foi passada como parâmetro. Após isso faz uma outra verificação, se o caractere da palavra[posicao] é maior ou menor que o que foi retornado, criando um nó interno baseado no resultado dessa comparação. A **segunda** verifica se a posição é menor que a posição do nó interno atual, se for satisfeita, cria um nó externo e retorna o caractere, como anteriormente, após isso, verifica se a palavra[posicao] é maior ou igual o caractere, passado como parâmetro e insere no lugar correto, retornando a criação de um nó interno. A **terceira** verifica, primeiramente se a posição atual e a posição do nó interno são iguais e se a palavra[posicao] é maior que a o caractere

retornado dessa mesma posição, após isso o restante é parecido com as anteriores, mudando apenas a verificação e os parâmetros da função Cria_No_Interno(). Como a função funciona recursivamente, a fim de fazer várias verificações para inserir no local exato da árvore, a **última,** simplesmente verifica se o caractere do nó interno é maior ou menor do que queremos inserir, se for maior chama recursivamente a função para a direita se não, para esquerda.

Insere_Palavra() - resumidamente, ela funciona procurando a posição exata na qual queremos inserir a palavra, ela também pesquisa o caractere onde se diferem e verifica se a palavra que queremos inserir já está na árvore, incrementando sua lista de ocorrências e montando seu índice invertido. Essa função chama a Insere_Entre, com o caractere e a posição do nó que se diferem.

Outras - as outras funções são auto explicativas, não possuem segredos e funcionam recursivamente, vale ressaltar que a *Busca_Palavra() retorna a lista encadeada de índices invertidos da palavra que buscamos e será útil para as medições e cálculos .

PS - para mais detalhes da implementação, verificar os comentários no código e as referências especificadas na introdução.

HASH

A tabela hash foi feita utilizando os seguintes structs:

```
3  typedef struct tipoItem {
4      char *key;
5      struct tipoItem *next;
6      Lista_Encadeada *lista;
7  } tipoItem;
8
9
10 typedef struct hashTable {
11     int countItens;
12     int tamanho;
13     tipoItem **entries;
14 } hashTable;
```

O struct ***tipoItem*** é utilizado para representar as palavras do texto, as quais serão inseridas na tabela. O struct é composto por um vetor de caracteres ***key*** que é a palavra em si, um ponteiro também do ***tipoItem*** que aponta para a próxima posição da lista encadeada, setado inicialmente para nulo, e um campo **Lista_Encadeada**, que é onde são guardadas as informações sobre as palavras(quantidade de vezes que ela se repete, de qual arquivo veio etc).

O struct ***hashTable*** por sua vez, não é nada mais que um vetor do tipo itens para armazenar as palavras,tem dois campos inteiros para mantermos o controle do tamanho e da quantidade de itens presente na tabela, os quais nos facilitam o uso de algumas funções. A tabela hash foi construída um pouco diferente de algumas outras versões. Tal diferença se dá pela maneira que se armazena as listas encadeadas. Em algumas implementações cotidianas, se guarda um ponteiro para palavra na posição inicial do vetor da tabela. Nesse caso, temos palavras que guardam ponteiros para a próxima palavra nesta posição inicial do vetor da tabela, essas palavras formam uma lista encadeada, porém já iniciam essa lista nelas mesmas, e não de um ponteiro.

As funções da tabela hash são:

```
16  unsigned int hash(const char *key,int tamanho);
17  tipoItem *criaItem(const char *key, int indiceDoArq);
18  hashTable *iniciaTabela(int tamanho);
19  void inserirNaTabela(hashTable *hashtable, const char *key, int idDoArquivo);
20  Lista_Encadeada **procurarNaTabela(hashTable *hashtable, const char *key);
21  void Free_Table(hashTable *hashtable);
22  void printaTabela(hashTable *tabela);
23  void printaPalavras_Hash(hashTable *tabela);
24  int calcPalavras(hashTable *tabela,int idDoArq);
```

A primeira função ***hash()*,** retorna um valor sempre positivo o qual é o valor hash, ou hashValue como está no código da palavra que foi passada como parâmetro para a função. Seu segundo parâmetro é o tamanho da tabela hash criada, utilizado para fazer o resto da divisão do valor da palavra multiplicado por 37 (valor aleatório escolhido para espalhar os resultados da função) pelo tamanho da tabela.

A função ***criaItem()*** retorna um ponteiro para um item, que será inserido na tabela hash. A função recebe como parâmetro uma palavra e o índice do arquivo do qual ela vem.

A função de inserção da tabela ***inserirNaTabela()*** começa criando o valor hash da palavra que lhe foi passada por parâmetro e o armazenando em uma variável do tipo ponteiro para item. Logo após, conferimos se a tabela na posição gerada pelo hashValue (retorno da função *hash*) é nula, caso ela seja nula, podemos inserir o item diretamente naquela posição e chamamos a função *criaItem() e retornamos*. Caso ela não seja nula, comparamos se a palavra que será inserida já está inserida em alguma das posições da lista encadeada, caso ela não esteja na posição inicial. Para fazermos isso, utilizamos um while e o método strcmp até encontrarmos uma posição nula, o que indica que não encontramos a palavra naquela lista. Caso a palavra já esteja na lista e a encontrarmos numa determinada posição, chamamos a função ***Le_Insere_No****,* que passa pela lista de pares ordenados desta palavra até encontrar o mesmo Id do arquivo passado por parâmetro, quando encontrado incrementamos em 1 a quantidade de vezes que esta palavra se repetiu. Caso não encontrarmos,criamos um novo node(nova célula) da lista encadeada para este arquivo.

Para imprimir as palavras e os índices invertidos, como a tabela hash não é ordenada, foi usado um vetor auxiliar de tipoItem para ser ordenado e percorrido no momento da impressão no terminal. Para ordenar o vetor, utilizamos o algoritmo Quick Sort, baseado no algoritmo apresentado no livro do Ziviani, e adaptado para a estrutura do item utilizado, comparando as strings armazenadas no campo key e reorganizando o vetor a partir dessas comparações. As funções utilizadas foram a printaTabela e printaPalavras, que são muito similares, diferindo-se apenas no momento da impressão, onde printaPalavras imprime somente as palavras em ordem alfabética, enquanto printaTabela imprime as palavras seguidas pelos índices invertidos.

Funcionamento da MAIN

Após compilar o projeto com o comando “make” (ver leiame.txt) e executar com ./main, será impresso no terminal o seguinte menu:

```
1 - Inserir palavras na Patricia
2 - Inserir palavras na Hash
3 - Imprimir todas as palavras
4 - Imprimir indice invertido
5 - Buscar palavra
6 - Sair
```

Opção 1 - ao selecionar esta opção, a estrutura Patricia é inicializada e o usuário deverá digitar o nome do arquivo que contém os arquivos a serem lidos. Após inserir o nome do arquivo corretamente, o programa chamará uma função que lerá arquivo por arquivo, localizados na pasta ./arquivos, pegará palavra por palavra, verificando se há alguma letra maiúscula, e convertendo para minúscula, caso necessário, e inserindo na Árvore Patricia. Depois de inserir todas as palavras um flag inteiro “patricia” recebe 1, que significa que a estrutura foi criada e as palavras inseridas.

Opção 2 - esta opção inicializará a tabela hash, e lerá o arquivo digitado pelo usuário no terminal, que contém os arquivos a serem lidos para a montagem do índice na Tabela Hash.. Após inserir o nome do arquivo corretamente, o programa chamará uma função que lerá arquivo por arquivo, localizados na pasta ./arquivos, pegará palavra por palavra, verificando se há alguma letra maiúscula, e convertendo para minúscula, caso necessário, e inserindo na Tabela Hash. Depois de inserir todas as palavras um flag inteiro “hash” recebe 1, que significa que a estrutura foi criada e as palavras inseridas.

Opção 3 - primeiramente esta opção verificará se você já abriu e leu algum arquivo e perguntará em qual estrutura você deseja visualizar as palavras inseridas. Se a opção digitada for 1, o programa chamará a função Printar_Palavra(&no) que imprimirá no terminal, todas as palavras inseridas na árvore Patricia, e ordem alfabética. Caso a opção digitada seja 2, o processo se repetirá, porém, desta vez, para a Tabela Hash, através da função printaPalavras_Hash(ht).

Opção 4 - esta opção funciona de forma parecida com a anterior, só que agora, ela imprimirá no terminal, além da palavra, em ordem alfabética, o índice invertido da mesma, através da função Printar_Ocorrências(&no) ou printaTabela(ht).

Opção 5 - ao selecionar esta opção, o usuário deverá digitar a quantidade de termos que deseja buscar e em qual estrutura. Após digitar no terminal, o programa alocará memória dinamicamente a fim de calcular a relevância. Ao fim da alocação, o usuário deverá digitar os termos a serem pesquisados. A cada termo digitado, o peso do mesmo em cada arquivo, é inserido em uma matriz termosxNumero_Arquivos, por exemplo, o peso do termo 1 no arquivo 1 é inserido na posição 0|0 na matriz, o do termo 1 arquivo 2 é inserido na posição 0|1, e assim por diante. Depois, a quantidade de palavras diferentes em cada arquivo e a relevância são calculados, retornando um vetor relevância, desordenado. Ao final, as memórias usadas são desalocadas e os arquivos são ordenados decrescentemente, usando uma árvore binária de busca BST e impressos no terminal

Opção 6 - o programa é encerrado, e a memória usada nas estruturas é desalocada.

Resultados e Discussões

Após reuniões e apuração dos testes e comparações feitas pelo grupo coletamos dados de diferentes cenários avaliados, os quais nos mostram pontos de vista referentes às comparações feitas entre a patricia e a hash, nos levando assim a certas conclusões com base nesses resultados:

Tempo e memória para a criação das estruturas:

Patricia	Hash(500)	Hash(100)	Hash(50)
0.017359s	**0.013858s**	**0.014761s**	**0.014831s**
276,608 bytes	**240,843 bytes**	**237,643 bytes**	**237,243 bytes**

Tempo e memória para a busca de 10 palavras e retorno da relevância:

Patricia	Hash(500)	Hash(100)	Hash(50)
0.002471s	**0.001846s**	**0.001673s**	**0.001550s**
278,128 bytes	**242,363 bytes**	**239,163 bytes**	**238,763 bytes**

Os tempos e gasto de memória foram calculados baseados na média de 5 testes e inseridos na tabela, num total de aproximadamente 5.000 palavras espalhadas em 20 arquivos.

Considerações sobre a criação e montagem da estrutura:

Percebe-se que na Patricia o tempo gasto para sua montagem é maior que o da tabela hash, isso se dá pela complexidade da estrutura e da quantidade de comparações feitas. Olhando apenas para a Hash, verifica-se que para Hash com tamanhos menores, o tempo gasto para a montagem é maior, o que é óbvio, visto que estamos falando de Tabelas Hash com encadeamento.

Passando a observar a quantidade de bytes alocada verifica-se que a Patricia alocou mais bytes que a tabela Hash, o que faz sentido, pois são muitos nós que são usados na árvore para fazer a inserção das chaves. Para as tabelas Hash, a alocação aumentou, para valores maiores, visto que alocamos mais espaços.

Considerações sobre a busca na estrutura:

O tempo para a busca na Patricia é maior que na Hash, na nossa implementação, faz sentido, visto que são muitas, as comparações necessárias para encontrar a palavra. Olhando apenas para as tabelas Hash os tempos ficaram bem próximos, com a Hash(500) desempenhando um pouco pior.. A quantidade de bytes alocada cresceu em todas as estruturas, visto que para o cálculo da relevância, foram alocadas algumas estruturas no HEAP, como matriz e vetores. A Hash com tamanho menor desempenhou melhor e alocou menos bytes que todas as outras estruturas.

Como foram feitos?

As medições de memória foram feitas usando Valgrind, um software livre que auxilia o trabalho de depuração de progarmas (ver referências) e para a medição do tempo, foi usado a biblioteca time.h, do C.
A IDE que utilizamos foi o VsCode e o terminal, WSL2-Ubuntu.

Máquina utilizada:

Processador	AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
RAM instalada	8,00 GB (utilizável: 5,94 GB)
ID do dispositivo	E8D54C01-21BE-4A7C-AE96-C6C1D0388F61
ID do Produto	00327-30226-15614-AAOEM
Tipo de sistema	Sistema operacional de 64 bits, processador baseado em x64

Considerações finais

Após o fim do trabalho prático podemos apontar certas dificuldades e facilidades encaradas pelo grupo durante a implementação desse projeto. Notamos uma certa facilidade na hora de entender como a árvore Patricia e Hash deveriam funcionar e como os cálculos das relevâncias poderiam ser feitos.Entretanto também tivemos notórias dificuldades na implementação, pesquisamos algumas implementações de árvores Patricia no google (ver referências) para entendermos como o algoritmo em si deveria ser feito, conversamos com colegas e amigos que auxiliaram em algumas partes da implementação de ambos TAD e no cálculo da relevância. Tivemos alguns problemas com alocação de memória na limpeza do buffer de entrada, que foram devidamente corrigidos.

Mas contudo é visível o proveito e as lições aprendidas durante esse período de desenvolvimento do trabalho prático, tais estas como o funcionamento de uma árvore de pesquisa e tabela hash, cálculo de relevâncias em arquivos, como alocar memória suficiente para não comprometer o funcionamento do programa, aprendemos também sobre limpeza de buffer e desalocação de memória. Vale ressaltar que dentre dezenas de pesquisas no google, encontramos diversas funções que ajudaram na montagem do código descritas nas referências que nem imaginaria existir, como a de cálculo de memória.

Referências:

https://github.com/FurlanV/Maquina-de-Busca-com-Patricia (Git Hub de Vinicius Furlan)

https://github.com/topics/patricia-tree (Diversas implementações da árvore Patricia, várias delas auxiliaram na nossa implementação)

http://www2.dcc.ufmg.br/livros/algoritmos/implementacoes-05.php (Implementações do livro do Ziviani, Projeto de Algoritmos com Implementações em Pascal e C)

https://programacaodescomplicada.wordpress.com/ (vídeo aulas do Paulo Bracks)

https://www.geeksforgeeks.org/dynamically-allocate-2d-array-c/ (Como alocar uma matriz dinamicamente)

https://pt.stackoverflow.com/questions/9427/limpeza-do-buffer-do-teclado-ap%C3%B3s-scanf (Limpeza do buffer do teclado)

https://www.vivaolinux.com.br/topico/C-C++/biblioteca-math.h (biblioteca Math.h no Linux)

https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks (Como calcular o consumo de memória)

https://www.javatpoint.com/convert-uppercase-to-lowercase-in-c (letras maiúsculas para minúsculas)

https://www.delftstack.com/pt/howto/python/remove-special-characters-from-string-python/ (remover caracteres especiais)

LETRAS.MUS.BR - Letras de músicas (arquivos inseridos)