



SERVIÇO PÚBLICO FEDERAL · MINISTÉRIO DA EDUCAÇÃO  
UNIVERSIDADE FEDERAL DE VIÇOSA · UFV  
CAMPUS FLORESTAL

## ***Trabalho 1 - AEDS 2***

***Estudo comparativo entre Árvore PATRICIA e Tabela HASH  
como estruturas para implementar arquivo invertido***

Artur Gil Luiz - 5924

Fabício Henrique Viana Albino - 5925

Erich Pinheiro Amaral - 5915

Pedro Quintão - 5916

Vinícius Arruda Melo Miranda - 5930

## ***Sumário***

1. Introdução.....	3
2. Organização.....	3
3. Desenvolvimento.....	3
3.1. “Hash”.....	4
3.2. “ListaOcorrencia”.....	5
3.3 “patricia”.....	6
3.4 “entrada”.....	7
3.5 “main”.....	9
4. Compilação e Execução.....	10
5. Desempenho.....	10
6. Conclusão.....	11
7. Referências.....	12

## 1. Introdução

O objetivo do trabalho é apresentar o desenvolvimento de um sistema de busca para os Projetos de Conclusão de Curso (POCs) do curso de Ciência da Computação da UFV - Campus Florestal, realizando um estudo comparativo entre a Árvore PATRICIA e a Tabela HASH.

Para isso, temos que construir um índice invertido, uma estrutura de dados fundamental para o funcionamento de máquinas de busca. Um índice invertido mapeia cada palavra-chave aos documentos em que ela aparece, facilitando a rápida recuperação de informações.

O estudo comparativo irá focar na implementação do índice invertido utilizando duas estruturas de dados distintas: uma Árvore PATRICIA, adaptada para o armazenamento de palavras, e uma Tabela HASH com tratamento de colisões por encadeamento. O desempenho de ambas as estruturas será avaliado em operações de inserção e pesquisa, fornecendo uma base teórica e prática para a escolha da estrutura mais adequada para esta aplicação.

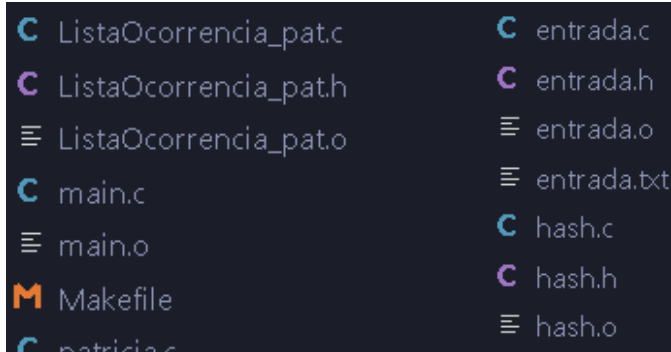
## 2. Organização

Nosso grupo se organizou da seguinte forma, 1 integrante responsável pela implementação da patricia, 1 integrante responsável pela implementação da hash, 3 integrantes responsáveis pela implementação da main.

Os arquivos foram compartilhados no github, permitindo então que cada membro do grupo acesse e contribuía com o projeto.

## 3. Desenvolvimento

Para desenvolver o algoritmo proposto, criamos os seguintes arquivos .h e .c:

A screenshot of a file explorer window with a dark background. It displays a list of files and folders in two columns. The files are color-coded: blue for C files, purple for H files, orange for Makefile, and grey for object files. The files listed are: ListaOcorrencia\_pat.c, ListaOcorrencia\_path, ListaOcorrencia\_pato, main.c, main.o, Makefile, patricia.c, entrada.c, entrada.h, entrada.o, entrada.txt, hash.c, hash.h, and hash.o.

ListaOcorrencia_pat.c	entrada.c
ListaOcorrencia_path	entrada.h
ListaOcorrencia_pato	entrada.o
main.c	entrada.txt
main.o	hash.c
Makefile	hash.h
patricia.c	hash.o

### 3.1. “Hash”

No TAD **hash.h** mostrada na figura \* tem as funções:

```
void FlVaziahash(TipoLista *Lista);
short Vaziahash(TipoLista Lista);
void InsHash(TipoItem x, TipoLista *Lista);
void GeraPesos(TipoPesos p);
TipoIndice h(Palavras Chave, TipoPesos p, int M);
void Inicializahash(TipoLista* Tabela, int M);
int PesquisaHash(Palavras Ch, TipoPesos p, TipoLista* T, int M, TipoApontador *Resultado);
void InsereHash(const char *palavra, int idDoc, TipoPesos p, TipoLista* T, int M);
void Imprime(TipoLista* Tabela, int M);
int Compara (const void *a, const void *b);
void ImprimeOrdenadohash(TipoLista* Tabela, int M);
void buscar_por_relevancia_hash(const char* consulta, struct ListaArquivos *docs, TipoLista* T, int M, TipoPesos p);
void ImprimirTotalCompInsercaohash();
```

O TAD Hash foi responsável por armazenar e recuperar palavras processadas dos documentos POCs, associando a cada uma uma lista de ocorrências com os documentos em que ela aparece e a frequência correspondente.

Além das funções básicas de inserção e busca, o TAD foi **modificado e estendido** com novas funcionalidades, com foco na análise de desempenho e busca por relevância textual.

#### Principais Funcionalidades Implementadas:

A função **InsereHash** foi adaptada para que, além de inserir uma palavra na tabela hash, ela também contabilize o número de comparações feitas durante a inserção. Caso a palavra já estivesse presente, a frequência no documento correspondente era incrementada; caso contrário, uma nova estrutura de ocorrência era alocada e inicializada.

A função **GeraPesos** é responsável por gerar uma matriz de pesos pseudo aleatórios, que são utilizados na função de dispersão (hash). Cada letra de cada posição da palavra recebe um peso específico, o que torna a função de hash mais eficiente e reduz colisões. A semente utilizada para a geração dos pesos é baseada no tempo atual, garantindo variedade a cada execução.

A função **PesquisaHash** tem como objetivo localizar uma palavra específica na tabela hash. Inicialmente, ela calcula o índice da palavra utilizando a função de dispersão **h**. Em seguida, verifica se a lista ligada naquele índice está vazia. Caso não esteja, percorre a lista comparando cada palavra armazenada com a buscada, utilizando **strcmp**. A cada tentativa de comparação, um contador é incrementado — esse contador é retornado para auxiliar na análise de desempenho. Se a palavra for encontrada, o ponteiro **Resultado** é atualizado com a posição na lista. Imagem da implementação:

A função **buscar\_por\_relevancia\_hash** recebe uma consulta textual, processa os termos e calcula a relevância de cada documento com base nos termos encontrados. Utiliza a métrica TF-IDF para atribuir um peso a cada termo e soma esses pesos para calcular a relevância final do documento. Ao final, os documentos são ordenados por relevância e exibidos. A função também contabiliza o número total de comparações feitas durante a busca.

### 3.2. “ListaOcorrencia”

Existem 2 TAD's **ListaOcorrencia.h**, um para patrícia e outro para hash, ambos tem seu funcionamento igual, separados apenas para melhor organização do projeto.

```
void FLOVaziaHash(ListaOcorrenciasHash *lista);
int insereOuAtuOcorrHash(ListaOcorrenciasHash *lista, int idDoc);
int obter_fji(ListaOcorrenciasHash* lista, int idDoc);
int obter_dj(ListaOcorrenciasHash* lista);
#endif
```

O TAD **ListaOcorrencia** foi responsável por gerenciar, para cada palavra, uma lista encadeada de documentos em que ela aparece, armazenando também a frequência de ocorrência em cada um.

Além das operações básicas, o TAD foi modificado e estendido com funções para cálculo de relevância textual (TF-IDF) e análise de desempenho, como a contagem de comparações durante a inserção e busca.

#### Principais Funcionalidades Implementadas:

A função **insereOuAtuOcorrHash** recebe uma lista de ocorrências e o identificador de um documento. Ela percorre a lista em busca do documento informado. Caso o documento já exista, sua frequência é incrementada. Caso contrário, uma nova ocorrência é criada e inserida na posição correta, mantendo a ordem crescente pelo **idDoc**. A função também contabiliza o número de comparações realizadas durante a operação, valor esse retornado para auxiliar na análise de desempenho.

### 3.3 “patricia”

No TAD **patricia.h** mostrada na figura \* tem as funções:

```
void ImprimirEmOrdem_Patricia(TipoArvore a);
void DestruirArvore(TipoArvore a);
TipoDib Bit(TipoIndexAmp i, char* k);
TipoArvore Insere(char* k, TipoArvore *t, int id_doc);
TipoArvore InsereEntre(char* k, TipoArvore *t, int i, int id_doc);
void Pesquisa(char* k, TipoArvore t);
TipoArvore CriaNoExt(char* k, int id_doc);
TipoArvore CriaNoInt(int i, TipoArvore *Esq, TipoArvore *Dir);
short EExterno(TipoArvore p);
```

A função **Bit** recebe um índice de bit e uma string, e retorna o valor do bit correspondente na posição da palavra. Esse bit é usado para decidir por qual caminho (esquerda ou direita) a busca ou inserção na árvore PATRICIA deve continuar. Trata corretamente casos em que a string é menor que o índice desejado, retornando 0.

A função **EExterno** verifica se um nó da árvore PATRICIA é **externo**, ou seja, se contém diretamente uma chave (palavra). Essa verificação é essencial para identificar o momento em que a árvore chegou em uma folha durante a busca ou inserção.

A função **CriaNoExt** cria um nó **externo** da árvore PATRICIA, armazenando uma nova palavra (**chave**) e inicializando sua lista de ocorrências com o **id\_doc** do documento onde foi encontrada. É usada tanto na inserção inicial quanto em divisões de nós.

A função **CriaNoInt** cria um nó **interno**, que armazena o índice de bit utilizado para decidir o caminho da árvore. Ele possui dois ponteiros (esquerda e

direita), conectando subárvores ou nós externos. É usado para inserir uma nova distinção entre palavras que diferem a partir de um certo bit.

A função **Inserere** adiciona uma nova palavra (**k**) à árvore PATRICIA. Se a árvore estiver vazia, insere como nó externo. Caso a palavra já exista, apenas atualiza sua lista de ocorrências com o novo **id\_doc**. Caso contrário, a função encontra o primeiro bit em que a nova palavra difere de uma já existente e chama **InserereEntre** para criar um novo nó interno e organizar a árvore.

A função **InserereEntre** é responsável por inserir uma nova palavra em uma posição intermediária da árvore, criando um nó interno com o bit de distinção e organizando os ponteiros esquerdo e direito corretamente. É fundamental para manter a estrutura compacta e balanceada da PATRICIA.

A função **Pesquisa** percorre a árvore PATRICIA buscando uma palavra (**k**). A cada passo, verifica o valor de um bit para decidir o próximo nó. Quando chega a um nó externo, compara a chave armazenada com a palavra buscada e informa se a encontrou ou não.

A função **ImprimirEmOrdem\_Patricia** percorre a árvore recursivamente e imprime cada palavra armazenada (nós externos), seguida de sua lista de ocorrências, no formato **<qtde, idDoc>**. A impressão segue a ordem de navegação da árvore, útil para análise e visualização do índice invertido armazenado na estrutura.

A função **DestruirArvore** libera recursivamente toda a memória alocada para a árvore PATRICIA, tanto nós internos quanto externos. É fundamental para evitar vazamentos de memória ao final da execução do sistema.

### 3.4 “entrada”

No TAD **entrada.h** mostrada na figura \* tem as funções:

```
void IniciaPalavrapat(PalavraInd *p);
void IniciaPalavrahash(PalavraInd *p);
ListaArquivos leitura_arq(char *arq);
void InsererePalavraIndice(const char *palavra_texto, int idDoc);
void token_palavras(PalavraInd *pal);
void ler_pocs(ListaArquivos *lista, TipoLista* Tabela, int M, TipoPesos p, TipoArvore *a);
void ImprimeIndiceInvertido();
int contar_palavras_unicas(ListaArquivos *lista, TipoPesos p_temp);
int achar_primo_inferior(int n);
int eh_primo(int n);
#endif
```

O TAD **EntradaEProcessamentoTexto** foi responsável por interpretar os arquivos de entrada e processar os documentos utilizados na construção do índice invertido.

A função **leitura\_arq** identifica os documentos a serem processados, enquanto **ler\_pocs** percorre cada um deles, lendo e normalizando palavra por palavra por meio da função **token\_palavras**, que remove caracteres especiais e padroniza o texto.

Após o tratamento, as palavras são inseridas nas estruturas de dados (PATRICIA e Tabela Hash), sendo associadas ao seu respectivo **idDoc**.

O TAD também foi estendido para calcular o número de termos distintos por documento, informação fundamental para o cálculo de relevância textual (TF-IDF).

### Principais Funcionalidades Implementadas:

A função **leitura\_arq** recebe como parâmetro o nome do arquivo de entrada (entrada.txt) e é responsável por interpretar o seu conteúdo. Esse arquivo especifica a quantidade de documentos a serem processados, bem como seus nomes. A função armazena essas informações em uma estrutura **ListaArquivos**, que será utilizada por outras funções do sistema para abrir e ler os arquivos indicados.

A função **ler\_pocs** é a principal responsável pelo processamento dos documentos. Ela percorre todos os arquivos listados na estrutura **ListaArquivos**, lendo palavra por palavra. Cada palavra é normalizada, enviada para inserção nas estruturas de dados (Hash, PATRICIA e vetor **v**), e é contabilizada para calcular o número de termos distintos do documento ( $n_i$ ). Também utiliza uma tabela hash temporária para garantir que cada palavra seja contada apenas uma vez por documento no cálculo de  $n_i$ .

A função **token\_palavras** é responsável por normalizar cada palavra lida de um documento. Ela percorre o texto da palavra, converte todas as letras para



minúsculas e remove quaisquer caracteres que não sejam letras ou números. Isso garante que todas as palavras sejam tratadas de forma padronizada, evitando duplicidades causadas por variações de formato

A função **InsererPalavraIndice** recebe uma palavra e o identificador do documento onde ela foi encontrada. Ela busca essa palavra no vetor **v** (onde ficam armazenadas todas as palavras distintas do corpus). Caso já exista, apenas atualiza suas listas de ocorrências (Hash e PATRICIA). Caso contrário, cria uma nova entrada no vetor, inicializando suas listas, e realiza a primeira inserção. Essa função centraliza o controle de quais palavras fazem parte do índice invertido.

A função **contar\_palavras\_unicas** percorre todos os documentos do corpus e conta quantas palavras únicas existem, sem considerar repetições. Para isso, utiliza uma tabela hash temporária apenas para controle das palavras já vistas. A função é útil para análises estatísticas do vocabulário total e pode ajudar no dimensionamento das estruturas de dados e na compreensão da dispersão lexical.

### 3.5 “main”

O módulo **main.c** foi responsável por coordenar a execução do sistema, integrando os TADs de leitura, indexação e busca por relevância textual.

Inicialmente, o programa realiza a leitura dos arquivos de entrada, calcula a quantidade de palavras únicas para definir o tamanho da tabela hash e constrói os índices invertidos utilizando as estruturas Hash e PATRICIA.

O sistema também oferece um menu interativo com opções para ler os arquivos, construir e imprimir os índices, além de realizar buscas por relevância textual (TF-IDF) utilizando a Hash.

Esse módulo centraliza o controle da aplicação, permitindo comparar os resultados das duas estruturas e facilitar o uso e teste do sistema.

## 4. Compilação e Execução

O algoritmo pode ser executado no terminal com comando “*mingw32-make*”, e então “.app.exe” abrirá o seguinte menu de comandos no terminal:

```
Escolha uma opcao:
1. Receber arquivos de entrada
2. Ler e construir os indices invertidos
3. Imprimir os indices invertidos
4. Buscar termos (Hash)
5. Buscar termos (Patricia)
6. Comparacoes insercao
0. Sair
```

- E então, aperte “1” e “enter” para receber os arquivos com os resumos dos POC’s.
- Após isso, “2” para registrar as palavras de cada POC e construir os indices invertidos.
- Para imprimir o indice invertido de cada palavra, “3”.
- Para buscar os termos na hash, “4”.
- Para buscar os termos na patricia, “5”.
- Para exibir o numero de comparações das inserções, “6”.

## 5. Desempenho

Para medir o desempenho de busca foi necessário adaptar um ciclo de 1000 repetições, retornando então o valor médio de busca para cada estrutura.

```
case 4:
    printf("Digite os termos para a busca: ");
    if (fgets(termo_busca, sizeof(termo_busca), stdin)) {
        termo_busca[strcspn(termo_busca, "\n")] = '\0';
        clock_t inicio = clock();
        for (int i = 0; i < REPETICOES; i++) {
            buscar_por_relevancia_hash(termo_busca, &entrada, Tabela_h, M, p_p
        }
        clock_t fim = clock();
        double tempo = (double)(fim - inicio) / CLOCKS_PER_SEC / REPETICOES;
        printf("Tempo médio de busca (Hash): %.9f segundos\n", tempo);
    }
    break;
```

Ao buscar pelo termo “zap” presente no arquivo 60, obtemos os seguintes resultados:

- Tempo médio de busca (Hash): 0.000594000 segundos
- Tempo médio de busca (Patricia): 0.000127000 segundos

Ao buscar pelo termo “as” presente diversas vezes em múltiplos arquivos, obtemos os seguintes resultados:

- Tempo médio de busca (Hash): 0.006007000 segundos
- Tempo médio de busca (Patricia): 0.000127000 segundos

Ao buscar pelo termo “you” que não está presente em nenhum dos arquivos, obtemos os seguintes resultados:

- Tempo médio de busca (Hash): 0.000647000 segundos
- Tempo médio de busca (Patricia): 0.000130000 segundos

Além disso, foi implementado uma função que compara a quantidade de comparações na inserção de cada estrutura.

Total de comparações em todas as inserções hash: 1181017

Total de comparações em todas as inserções patricia: 190521

Portanto, concluímos que a hash, em múltiplos cenários, se apresenta mais ineficiente em questão de tempo de busca e quantidade de comparações para inserção em relação a Patricia.

## 6. Conclusão

Em relação ao desempenho, os testes comparativos em múltiplos cenários (busca por termo presente uma vez, termo presente diversas vezes e termo ausente) demonstraram consistentemente que a Árvore PATRICIA apresentou um tempo médio de busca significativamente menor em comparação com a Tabela HASH, além de apresentar 10x menos comparações para inserção. Isso sugere que, para a aplicação de índice invertido estudada, a Árvore PATRICIA se mostrou mais eficiente em termos de tempo de busca e inserção.

Conclui-se, portanto, que embora ambas as estruturas sejam viáveis para a implementação de um arquivo invertido, a Árvore PATRICIA demonstrou superioridade em desempenho de busca neste estudo comparativo, fornecendo

uma base prática para a escolha da estrutura mais adequada para aplicações similares.

## 7. Referências

- <http://www2.dcc.ufmg.br/>
- <https://ccp.caf.ufv.br/tccs/>