Hash de Índice Remissivo
IFMG - Campus Formiga

Guilherme Cardoso Silva

[1] Algoritmos e Estrutura de Dados II – Instituto Federal de Minas Gerais (IFMG)
Caixa Postal 35570-000 – Formiga – MG – Brasil

godgcs@gmail.com.br

1. Introdução

O objetivo geral deste trabalho é o projeto e a implementação de um sistema que construa um índice remissivo. Sendo que para implementação deste método deve ser utilizado Tabelas Hash tanto no formato OpenHash quando no ExternHash.

Com a implementação de ambos os metodos foi possivel comparar ambas as estruturas por meio de medidas de: tempo, complexidade computacional, facilidade de implementação e numero de colisões na inserção das palavras.

Levando em consideração o período em que foi desenvolvido o metodo OpenHash, ele apresenta um desepenho razoavel, porém ainda não é o metodo mais indicado principalmente para um grande texto junto a um grande numero de palavras chaves.

O metodo HashExtern apresentou um desempenho muito melhor a OpenHash devido ao seu melhor tratamento com relação a colisões.

2. Implementação

Para compilar é necessario apenas utilizar *gcc *.c -DTAM="TamanhoHashExterna" -o "NomeExecutavel"* para gerar o executavel.

Para executar são necessarios passar como parametro o nome do arquivo que contém as palavras chaves, o nome do arquivo que contém o texto a ser consultado e o nome do arquivo onde serão gravados os indices das palavras lidas.

A implementação deste projeto foi divido em duas TADs, sendo uma da estrutura *OpenHash* e outra *ExternHash*. O motivo para esta separação entre os codigos é que futuramente caso seja necessario reaproveitar alguma das duas, preferencialmente a que apresentar melhor desempenho, para utilizar para construção de algum indice se tornaria mais facil a separação entre as funções.

2.1. OpenHash

A estrutura de dados implementada para este metodo foi a seguinte:

```
struct linhas{
    int Linha;
    struct linhas *Prox;
};
struct nodo{
    char Palavra[50];
```

```
    struct linhas *IndiceI;
    struct linhas *IndiceF;
    int Vazio;
    int Colisoes;
};
typedef struct nodo *Nodo;
typedef struct linhas *Linha;
typedef struct nodo *TabelaOpen;
```

A estrutura seria definida por um *array* de *struct nodo*, Nodo (definição usada dentro da TAD) ou TabelaOpen (Definição publica), onde cad aindice do vetro seriam compostos pelas seguintes propriedades:

- **Palavra:** Vetor contendo a palavra chave a ser buscada no arquivo;
- **IndiceI:** Ponteiro para o primeiro elemento de uma lista de ocorrencia da palavra em um texto;
- **IndiceF:** Ponteiro para o ultimo elemento de ocorrencia da palavra em um texto;
- **Vazio:** Identifica se esta posição da Hash já foi preenchida ou não em uma leitura de palavras chaves;
- **Colisoes:** Contabiliza a quantidade de colisões existiram neste indice na inserção de palavras chaves.

A TAD *openHash.c* foi implementada de acordo que pudesse armazenar e consultar esta lista de palavras, e para isto foi disponibilizado as seguintes funções:

- **TabelaOpen OLeChaves(char Arq[], int Op, int Tam, int* vetor:** Responsavel por alocar a memoria para a estrutura, e realizar toda a leitura de palavras chaves do arquivo e as armazenar na Hash. Ela recebe como parametro o arquivo que contém as palavras chaves, a opção Hash que pode ser 1,2 ou 3, o tamanho da estrutura que deve ser maior ou igual ao tamanho de palavras chaves e um vetor de pesos aleatorios que é utilizado na função Hash 3. Ela irá retornar um ponteiro para a estrutura que armazena a Hash.
  Primeiramente ela aloca o espaço de memoria para o tamanho de palavras chaves que é passado por parametro pela variavel *Tam*, Em seguida ele abre o arquivo passado por parametro pela varivel *Arq[]* e começa uma leitura das palavras contidas nele. A cada palavra que é lida, ele verifica se é considerada uma palavra valida, caso seja ele a insere na Hash. Para inseção na Hash é selecionada uma opção Hash que é passada por parametro pela variavel *Op*, sendo que caso exista colisão a propria função irá gerenciar isto e buscar uma nova posição para a inserção.
  Este processo vai se repetir até que todo o arquivo seja lido.
- **void OBuscaPalavras(TabelaOpen OHash, int Tam, char ArqFrases[], int Op, int* vetor):** Responsavel por realizar a busca das palavras chaves no arquivo texto, a cada palavra encontrada ele armazena em uma lista quais as linhas que elas foram encontradas. Ela recebe como parametro o ponteiro para a estrtura Hash, o tamanho da estrutura Hash, o arquivo que contém o texto a ser lido e um vetor de pesos aleatorios que é utilizado na função Hash 3, caso não ela não va ser utilizada é possivel passar NULL como parametro.
  Através do arquivo passado como parametro que é referente ao texto a ser lido, a função começa lendo linha por linha do arquivo, e faz uma busca por todas as

palavras em cada linha, quando um intervalo de posições forma uma palavra valida a função utiliza da mesma função Hash utilizada para inserção e começa a buscar pela palavra, comparando a encontrada com as palavras gravadas na estrutura de dados, caso ela encontre a palavra na Hash é criado um *nodo linha* que é inserido o numero da linha lida no final dos indices remissivos da palavra buscada. Este processo se repete para todas as palavras em uma linha, e por todas as linhas do arquivo texto.

- **void OResultadoBusca(TabelaOpen OHash, int Tam, FILE *arquivo):** Responsavel por gerar os resultados de saida da Hash, onde irá ordenar utilizando o metodo QuickSort todas as palavras chaves e irá escreve-las em um arquivo junto do indice onde elas são localizadas no texto. Ele recebe como parametro um ponteiro para a Hash, o tamanho da estrutura e um pontei para o arquivo onde será escrito os resultados.
  Após a leitura do texto onde foi identificado o indice remissivo de todas as palavras chaves, esta função foi implementada para que escrevesse um arquivo de saida com os resultados.
  Ela ordena todas as palavras em ordem alfabetica com o metodo de ordenação QuickSort e em seguida escreve em um arquivo de saida, informado pelo usuario, todas as palavras junto do indice de onde elas ocorrem no texto.
- **void OLiberaHash(TabelaOpen OHash, int Tam):** Responsavel por desalocar a memoria utilizada para armazenar os dados em uma Hash.
- **int OColisoes(TabelaOpen OHash, int Tam):** Responsavel por contabilizar a quantidade de colisões na estrutura.
  No nodo de cada palavra é armazenado a quantidade de vezes em que ocorreu uma colisão com aquela posição na tentativa de uma inserção, está função faz um somatorio de todos estes valores e retorna para que seja contabilizada.
- **int OTamanhoHash(char Arq[]):** Responsavel por contar a quantidade de palavras chaves em um arquivo. Ela retorna o tamanho que deve ser utilizado para criação da Hash.
- **int* OPreencheVetorPesos(void):** Para utilizar a função Hash 3 é necessario utilizar um vetor de pesos para calcular uma posição onde se deve inserir, então esta função é responsavel por criar este vetor.
  Como uma palavra pode ter no maximo até 32 caracteres, está função cria um vetor de 50 posições contendo inteiros aleatorios que serão usados para ponderar os valores multiplicados aos caracteres quando for utilizar a função Hash 3 referente a bibliografia do [Ziviani 2011].
  As funções de leitura de palavras chaves e de busca no texto utilizam estas funções.

2.2. HashExtern

A estrutura de dados implementada para este metodo foi a seguinte:

```
struct linhas{
        int Linha;
        struct linhas *Prox;
};
struct nodo{
        char Palavra[50];
```

```
        struct linhas *IndiceI;
        struct linhas *IndiceF;
        struct nodo *Prox;
};
struct tabela{
        struct nodo *Inicio;
        struct nodo *Fim;
        int Vazio;
        int Colisoes;
};
typedef struct nodo *Nodo;
typedef struct linhas *Linha;
typedef struct tabela *TabelaExtern;
```

A estrutura seria definida por um ponteiro um array de tamanho N onde cada um apresentaria as seguintes propriedades:

- **Inicio:** Ponteiro para o primeiro elemento de uma lista palavras chaves;
- **Fim:** Ponteiro para o ultimo elemento de uma lista palavras chaves;
- **Vazio:** Identifica se esta posição da Hash já foi preenchida ou não em uma leitura de palavras chaves;
- **Colisoes:** Contabiliza a quantidade de colisões existiram neste indice na inserção de palavras chaves.

Os ponteiros para Inicio e Fim são do tipo *struct nodo* e contém as mesmas propriedades que a struct do *OpenHash*, a diferença é que aqui são indexadas externamente.

A TAD *externHash.c* foi implementada de acordo que pudesse armazenar e consultar esta lista de palavras com indereçamento externo e para isto foi disponibilizado as seguintes funções:

- **TabelaExtern ELeChaves(char Arq[], int Op, int* vetor)**;
- **int ETamanhoHash(void)**;
- **void EBuscaPalavras(TabelaExtern EHash, char Arq[], int Op, FILE *arquivo, int* vetor)**;
- **void EResultadoBusca(TabelaExtern EHash, FILE *arquivo)**;
- **void ELiberaHash(TabelaExtern EHash)**;
- **int EColisoes(TabelaExtern EHash)**;
- **int* EPreencheVetorPesos(void)**.

Estas funções são similares as detalhadas na TAD *openhash.c*, logicamente a forma como são implementadas é diferente, por se tratar de um endereçamento externo de palavras. Estas funções também não necessiram que o tamanho da hash seje passado por parametro, pois ele já é definido na compilação do executavel.

3. Complexidade

Complexidade das principais funções das TADs

3.1. OpenHash

- **OLeChaves:** O calculo do custo seria: n(numero de palavras chaves) * 32(tamanho de caracteres das palavras chaves) * n(pior caso de colisão na inserção de uma palavra).
  Então o custo é O(n^2).
- **OBuscaPalavras:** O custo é m(numero de palavras no arquivo texto) * 1(verificar palavra valida) * 1, logn ou n(consultar palavra na Hash pode ser 1 ou n no pior caso) * 1(armazenar indice na estrutura).
  Tendo como resultado O(m) como melhor caso, O($mlogn$) como caso medio ou O($mnlogn$) como pior caso.
  É importante destacar que a maioria das palavras não é encontrada nas hashs, então o custo de uma busca assim será sempre O(n), resultando em uma forma ineficiente de implementar a extrutura.
- **OResultadoBusca:** O custo é n(criar vetor com todas as palavras para ordenar) + nlog(n)(Ordenação das palavras) + n(escrever um arquivo de indices remissivos).
  Resultando em um custo de O($nlogn$).
- **OLiberaHash:** Para liberar a memoria utilizada pela estrutura é necessario liberar cada indice de linha contido nos nodos de palavras, para em seguida liberar a tabela hash que é um *array* de nodos palavras.
  O custo seria de 1(custo de liberar toda a tabela hash) * n(percorrer toda a tabela hash para liberar os indices) * m(numero de indices de cada palavra), resultando em O(nm).
- **OPreencheVetorPesos:** 1(alocar memoria) + 50(tamanho do vetor) * 1(atribuir valor a posição).
  Resultando em um custo de O(1).
- **OColisoes:** Como é necessario contabilizar a colisão de todas as palavras o custo é O(n).
- **OTamanhoHash:** Como é necessario ler todo o arquivo para poder saber o tamaho o custo é O(n).

3.2. ExternHash

- **ELeChaves:** O calculo do custo seria: n(numero de palavras chaves) * 32(tamanho de caracteres das palavras chaves) * 1(custo de colisões).
  Então o custo é O(n).
- **EBuscaPalavras:** O custo é m(numero de palavras no arquivo texto) * 1(verificar palavra valida) * 1 ou logn(consultar palavra na Hash pode ser 1 ou logn nos piores casos) * 1(armazenar indice na estrutura).
  Tendo como resultado O(m) como melhor caso, O($mlogn$) como caso medio, já o pior caso tem uma probabilidade muito pequena de acontecer levando em consideração a forma que as funções foram implementadas.
- **EResultadoBusca:** O custo é n(criar vetor com todas as palavras para ordenar) + nlog(n)(Ordenação das palavras) + n(escrever um arquivo de indices remissivos).
  Resultando em um custo de O($nlogn$).
- **ELiberaHash:** Para liberar a estrutura externa o custo é um pouco diferente, pois que é necessario liberar todos os indices de cada nodo, em seguida o nodo correspondente, e repetir este processo até que todos tenham sido liberados, para só então liberar a tabela.

De uma forma geral o custo acaba se mostrando o mesmo da OpenHash por se tratar que ambas terão que liberar todos os indices e os nodos, a unica diferença é a forma como isto é implementado, sendo que pequenas constantes vão não irão influenciar no resultado final
Com o custo de m(numero de indices de cada nodo) * n(numero de palavras chaves). O(mn).

- **EPreencheVetorPesos:** 1(alocar memoria) + 50(tamanho do vetor) * 1(atribuir valor a posição).
  Resultando em um custo de O(1).
- **EColisoes:** Como é necessario contabilizar a colisão de todas as palavras o custo é O(n).
- **ETamanhoHash:** O tamanho da Hash é definido por uma constante, então o custo é O(1).

4. Descrição dos Testes

Para que fosse possivel avaliar os metodos implementados, foram realizados uma bateria de testes, variando as funções Hash de busca e inserção, os metodos de colisão e o tamanho das estruturas.

Foram implementadas três funções Hash para os testes:

- **Hash1:**
  for(i=0; isalpha(Palavra[i]); i++){
  h = h + Palavra[i];
  }
  h = h + Peso;
  return(h%Tam);
- **Hash2:**
  for(i=0; isalpha(Palavra[i]); i++){
  *h = ((h*2 + Palavra[i])+1932)%Tam;*
  }
  h = h + Peso;
  return(h%Tam);
- **Hash3:** Esta função é implementada no livro Projeto de Algoritmos e foi proposta para implementação no trabalho, ela utiliza de um vetor de pesos aleatorios para variar o numero resultante da Hash.
  for(i=0; isalpha(Palavra[i]); i++){
  *h = h + Palavra[i] * vetor[i];*
  }
  h = h + Peso;
  return(h%Tam);

As três funções na estrutura HashExtern foram testadas em três tamanhos diferentes, para poder avaliar em diversos casos qual se sairia melhor.

5. Resultados

Os resultados de todos os testes podem ser analizados nas Tabelas 1, 2, 3 e 4.

Table 1. OpenHash

Função	Colisões	Tempo
Hash 1	1421	0.403
Hash 2	480	0.722
Hash 3	475	0.446

Table 2. HashExtern Tamanho 50

Função	Colisões	Tempo
Hash 1	69	0.029
Hash 2	84	0.032
Hash 3	70	0.025

Table 3. HashExtern Tamanho 200

Função	Colisões	Tempo
Hash 1	30	0.027
Hash 2	44	0.024
Hash 3	22	0.029

Table 4. HashExtern Tamanho 1000

Função	Colisões	Tempo
Hash 1	18	0.026
Hash 2	13	0.027
Hash 3	9	0.022

Pode-se observar uma grande diferença nos numeros da *OpenHash* e da *ExternHash*, os valores de tempo chegam a aproximadamente 15 vezes mais que o endereçamento externo de palavras, com uma grande diferença no numero de colisão, onde a *OpenHash* também apresentou um maior custo de utilização.

Para os testes com a *OpenHash* a função Hash 3 referente ao livro do Ziviane apresentou o melhor desempenho, comparado as outras duas funções.

A função Hash 2 apresentou um numero de colisão razoavelmente baixo comparado as outras duas funções, porém o tempo gasto por ela foi um valor bem alto, variando de 0.7 segundos a 1 segundo na maioria dos testes.

Para os testes com a *ExternHash* é possivel observar que quanto maior o tamanho da Hash menor se torna o numero de colisões, e o tempo também sofre uma pequena redução, como se trata de uma medida de tempo extremamente pequena, este pequeno valor tem uma grande importancia.

A função Hash 3 novamente apresentou um desempenho excelente, onde na maioria dos testes apresentou o menor tempo e o menor numero de colisões, onde em uma Hash de tamanho 1000 foi constatado apenas 9 colisões e sua execução foi feita em menos de 0.023 segundos.

Com estes resultados pode-se costatar que o metodo de endereçamento externo se mostrou mais eficiente para a implementação de um indice remissivo.

A função Hash 3 para consulta e colisão se mostrou melhor em varios testes, comparada com as outras duas, então ela também é indicada para trabalhar com este tipo de consulta. É bom destacar que como se trata de um vetor de pessos aleatorios este desempenho pode variar, chegando a existirem pesos que resultem em um baixo desempenho, porém na maioria dos casos como já descrito o desempenho foi superior.

6. Conclusão

ESte trabalho resultou em uma grande experiencia e conhecimento com relação a formas de implementar uma extrutura Hash. Além disto pode-se exercitar com um problema pratico, e comparar duas formas de armazenamento, o open e o externo, ambos apresen-

taram resultados bem proximos uns dos outros, mas ainda assim foi possivel classificar qual seria a melhor forma para tratar o problema proposto.

Neste caso o HashExterno apresentou um melhor resultado, um custo de tempo menor e com um numero de verificações inferior. O metodo OpenHash apesar de apresentar um maior custo também conseguiu tratar o problema, e isto mostra que existem varias formas de se tratar um problema real sendo algumas formas mais eficientes e outras menos eficientes.

References

Ziviani, N. (2011). *Projeto de Algoritmos: Com Implementacoes em Pascal e C*. Cengage Learning, 3 edition.