Trabalho da M2 de Estruturas de Dados

Ivan dos Santos, André Luiz

Centro de Ciências Tecnológicas da Terra e do Mar
Universidade do Vale do Itajaí (UNIVALI)
Itajaí – SC – Brasil
{silva.andre,ivancsantos}@edu.univali.br

Resumo. *Através deste trabalho, desenvolvemos estratégias para solução de conflitos de tabela Hash (ou tabela de dispersão) em dois programas, sendo (1) utilizado algoritmo de endereçamento aberto e (2) utilizado algoritmo de listas encadeadas para tratamento dos conflitos. Em ambos os programas avaliamos a performance através de métrica de "log" de quantidade de operações realizadas para inserção e leitura de uma tabela de mil posições, estando preenchida em 10%, 25%, 75%, 100%, e 200%.*

1. Introdução

Este trabalho aborda estratégias para tratamento de conflitos de tabelas de dispersão (tabela hash), visto que o mecanismo de tratamento de colisões é muito importante ao se utilizar tabelas hash, uma vez que é natural que ocorram conflitos na função de espalhamento, a não ser em algoritmos muito complexos que produzem chaves únicas na saída como os utilizados para criptografia, por exemplo.

Nas próximas seções abordaremos individualmente ambas as estratégias utilizadas, apresentando o código utilizado bem como o relatório com log gerado em nossos testes. Na seção 2.1 apresentamos na íntegra o algoritmo utilizado para função de espalhamento bem como função de tratamento de colisões utilizando o método de endereçamento aberto. O método de listas encadeadas é da mesma forma apresentado na íntegra na seção 2.2.

2. Definições do Trabalho

Nesta seção são apresentadas as operações utilizadas para a Tabela Hash desenvolvidas para o Trabalho da M2 da disciplina de Estruturas de Dados.

2.1. Estrutura e Operações do Endereçamento Aberto

Abaixo na íntegra o código do programa:

Função inicializa – inicializa a tabela hash com elementos com valores “zerados” de uma estrutura TElemento;

```
void inicializa(tabelaHash &t){
    for(int i = 0; i < TAM; i++){
        TElemento novoElemento;
        novoElemento.chave = NULL;
        novoElemento.dado = "";
        t.vet[i] = novoElemento;
    }
}
```

Função dispersão – gera a chave para a tabela hash;

```
int dispersao(tabelaHash &tabela, int chave){
    return chave % TAM;
}
```

Função inserir – executa a inserção do novo elemento na tabela, chamando primeiramente a função dispersão para que seja gerada a chave com a posição da tabela hash a ser inserido o elemento. Caso a posição esteja ocupada, o elemento será inserido na próxima posição livre. Caso a chave já exista na tabela hash, ela não será inserida e retornará valor false. São realizadas operações adicionais para fins de log de desempenho;

```
bool inserir(tabelaHash &tabela, TElemento elemento, int &op){
    int indice = dispersao(tabela, elemento.chave);

        TElemento* novo = new TElemento;
        if (novo == NULL) {
                return false;
        }

        *novo = elemento;

        for(int i = indice; i < TAM; i++){
         if(tabela.vet[i].chave == novo->chave){
             return false;
         }
         if (tabela.vet[i].chave == NULL) {
             tabela.vet[i] = *novo;
             op++;
             return true;
         }
         op++;
        }
    for(int i = 0; i < indice; i++){
         if(tabela.vet[i].chave == novo->chave){
             return false;
         }
         if (tabela.vet[i].chave == NULL){
             tabela.vet[i] = *novo;
             op++;
             return true;
         }
         op++;
    }
        return false;
}
```

Função pesquisar – executa a pesquisa de um elemento aleatório na tabela Hash, caso o mesmo seja encontrado retorna true, caso contrário retorna false. São realizadas operações adicionais para fins de log de desempenho;

```
bool pesquisar(tabelaHash &tabela, int chave, int &op){
    int indice = dispersao(tabela, chave);

        if (tabela.vet[indice].chave == NULL) {
         op++;
                return false;
        }

        if (tabela.vet[indice].chave == chave){
         primeiro++;
         op++;
         return true;
        }

        for(int i = indice; i < TAM; i++){
         if(tabela.vet[i].chave == chave){
             encontrou++;
             op++;
             return true;
         }
         op++;
        }

        for(int i = 0; i < indice; i++){
         if(tabela.vet[i].chave == chave){
             encontrou++;
             op++;
             return true;
         }
         op++;
        }
        naoencontrou++;
        return false;

}
```

2.2. Estrutura e Operações do Lista Encadeada

Abaixo na íntegra o código do programa hash com lista encadeada:

Função inicializa - inicializa a tabela hash com estruturas de listas encadeadas vazias;

```
void inicializa(tabelaHash t){
    for(int i = 0; i < TAM; i++){
        TLista novaLista;
        novaLista.qtd = 0;
        novaLista.inicio = NULL;
        t.vet[i] = novaLista;
    }

}
```

Função dispersão - gera a chave para a tabela hash;

```
int dispersao(tabelaHash &tabela, int chave){
    return chave % TAM;
}
```

Função inserir - executa a inserção do novo elemento na tabela, chamando primeiramente a função dispersão para que seja gerada a chave com a posição da tabela hash a ser inserido o elemento. Cada posição da tabela hash possui uma lista encadeada, portanto é executada uma função insere no final para que o novo elemento ocupe a posição “próximo” do último elemento anterior. Caso a chave já exista na tabela hash, ela não será inserida e retornará valor false. São realizadas operações adicionais para fins de log de desempenho;

```
bool inserir(tabelaHash &tabela, TElemento elemento, int &op){
    int indice = dispersao(tabela, elemento.chave);

        TElemento* novo = new TElemento;
        if (novo == NULL) {
                return false;
        }
        *novo = elemento;
        if (tabela.vet[indice].inicio == NULL) {
                tabela.vet[indice].inicio = novo;
                op++;
        }
        else {
                TElemento* nav = tabela.vet[indice].inicio;
                op++;
                while (nav->proximo != NULL) {
             if (novo->chave == nav->chave){
                 return false;
             }
                        nav = nav->proximo;
                        op++;
                }
                nav->proximo = novo;
                op++;
        }
        novo->proximo = NULL;
        tabela.vet[indice].qtd++;
        return true;
}
```

Função pesquisar - executa a pesquisa de um elemento aleatório na tabela Hash, caso o mesmo seja encontrado retorna true, caso contrário retorna false. São realizadas operações adicionais para fins de log de desempenho;

```
bool pesquisar(tabelaHash &tabela, int chave, int &op){
    int indice = dispersao(tabela, chave);

        if (tabela.vet[indice].inicio == NULL) {
               return false;
        }
        TElemento* nav = tabela.vet[indice].inicio;
        op++;
        if(nav->chave == chave){
         primeiro ++;
        }
    while (nav != NULL) {
         if(nav->chave == chave){
             encontrou++;
             return true;
         }
                nav = nav->proximo;
                op++;
        }
        naoencontrou++;
        return false;

}
```

3. Comparativo

Na tabela 1 abaixo, comparamos o custo de complexidade do pior caso dos métodos de tratamento de colisões na tabela hash, através de endereçamento aberto e lista encadeada.

Table 1. Comparativo entre o pior caso

	Endereçamento Aberto	**Lista Encadeada**
Inicializa	O(n)	O(n)
Inserir	O(n)	O(n)
Pesquisar	O(n)	O(n)

Na Tabela 2, vemos as operações de inserção nos métodos de tratamento de colisão por endereçamento aberto e lista encadeada. Notamos o ganho de desempenho da lista encadeada sobre o método de endereçamento aberto conforme a tabela hash vai sendo ocupada.

Table 2. Média de operações na inserção por taxa de ocupação

	Endereçamento Aberto	Lista Encadeada
10%	1,00	1,30
25%	1,40	1,50
75%	6,50	1,50
100%	357,40	1,70
200%	1000,00	2,70

Na Tabela 3, vemos as operações de pesquisa nos métodos de tratamento de colisão por endereçamento aberto e lista encadeada. Notamos o ganho de desempenho da lista encadeada sobre o método de endereçamento aberto conforme a tabela hash vai sendo ocupada.

Table 3. Média de operações na pesquisa por taxa de ocupação

	Endereçamento Aberto	Lista Encadeada
10%	300,70	1,10
25%	400,60	1,20
75%	800,20	1,30
100%	1000,00	1,80
200%	1000,00	3,10

Notamos através das tabelas anteriores que o método de tratamento de colisões por lista encadeada permite manter a performance mesmo com a tabela hash estando super lotada, além de permitir a continuação da tabela mesmo após 100% de utilização, ela ainda permite inserções e pesquisar relativamente rápidas em comparação com o método por endereçamento aberto.

Na Figura 1, observamos os em forma de gráfico o comparativo da tendência de ambos os métodos na função de inserção.

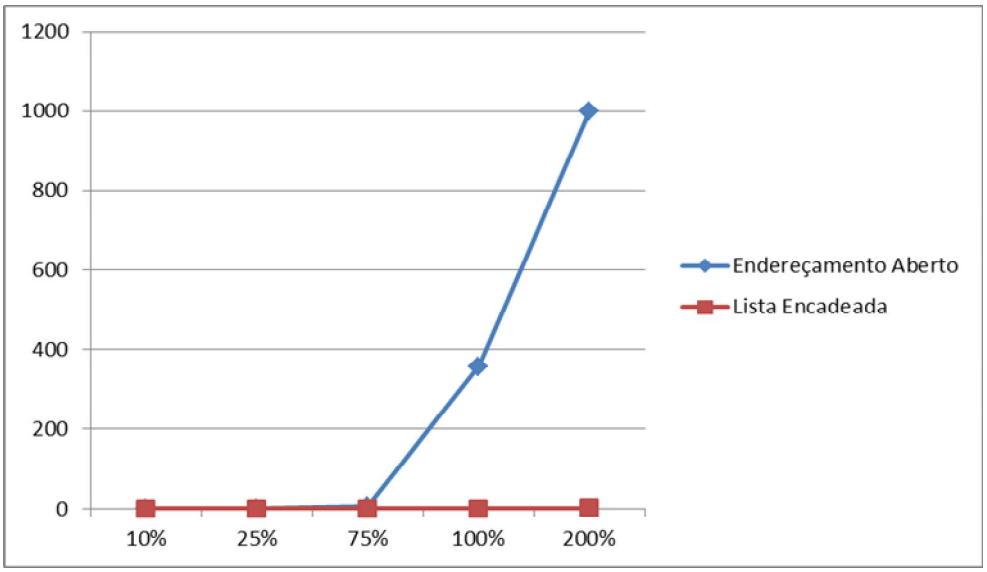

Figura 1. Gráfico da média operações na inserção por taxa de ocupação

Na Figura 2, observamos os em forma de gráfico o comparativo da tendência de ambos os métodos na função de pesquisa.

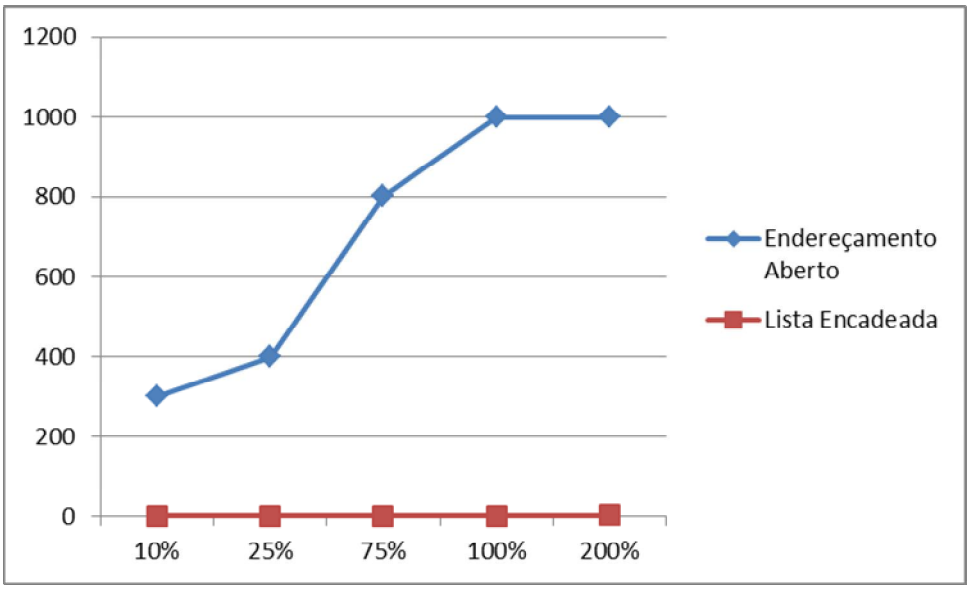

Figura 2. Gráfico da média de operações na pesquisa por taxa de ocupação

3. Conclusões

Concluímos dessa forma que dentre os métodos abordados para tratamento de colisões, o que apresenta maiores vantagens é através de lista encadeada. O mesmo permite a inserção de dados além da capacidade da tabela hash (n), pois passa a conectar elementos de uma lista encadeada à cada posição da tabela hash. Além disso, mantém um desempenho aceitável para razoáveis quantidades de dados, enquanto que o método de endereçamento aberto perde muito desempenho conforme a tabela hash cresce.