DJAIR MAYKON DE NOVAES MIRANDA

HÉRSON REIS REZENDE DOS SANTOS

ISIS BEATRIZ MUTTI SILVA SANTOS

RELATÓRIO DO TRABALHO I

MATA54 - ESTRUTURAS DE DADOS E ALGORITMOS II

HASHING E COLISÕES

SALVADOR - BA

MAIO, 2023

Introdução

Este relatório irá abordar as tabelas de hashing, que são estruturas de dados eficientes e que permitem armazenar e recuperar informações com base em uma chave primária (única). As tabelas de hashing são comumente utilizadas em várias aplicações como, por exemplo, bancos de dados, sistemas de indexação e algoritmos de busca. No entanto, quando uma função hash faz com que duas chaves distintas sejam mapeadas para o mesmo local na tabela, ocorre o que é chamado de colisão.

Lidar com colisões é um desafio imprescindível no uso de tabelas hashing. Dentre as mais variadas técnicas para resolver colisões, estão as com encadeamento explícito, que utilizam ponteiros e alocação de memória, assim como os hashings de endereçamento aberto, que fazem realocação de registros. Cada abordagem possui vantagens e desvantagens específicas, e a escolha da técnica adequada depende do contexto e dos requisitos da aplicação. Gerenciar tais colisões é um passo fundamental para garantir o desempenho eficiente e a corretude das operações em uma tabela hashing.

Espera-se, ao final deste relatório, que seja possível visualizar e comparar os resultados analíticos das diferentes funções de hashing implementadas e aplicadas neste estudo. Espera-se também que, para o leitor, a descrição teórica dos métodos objetos de estudo seja satisfatória.

Descrição dos métodos objetos de estudo

Antes de começar a falar diretamente sobre os métodos e tabelas de hashing, é necessário definir e entender o que são as funções de hash, ferramental técnico que é fundamental para a implementação das tabelas. As funções de hash são métodos auxiliares que ajudam a mapear os registros a partir de uma determinada lógica. Por exemplo, a função hash mais comum e utilizada é a ***h(k) = k mod m***, onde m é o tamanho da tabela hashing. Em outras palavras, ***k*** será inserido na tabela justamente na posição que é o resto da divisão entre ***k*** e ***m***. Ainda assim, é válido destacar que existem inúmeras funções de hash, em que se escolhe a mais adequada de forma estratégica.

Sabendo o que são as funções de hash e como elas funcionam, o primeiro método para geração de tabelas que será abordado neste estudo será o de encadeamento explícito com o uso de ponteiros usando alocação dinâmica de memória. Nesta técnica, cada posição da tabela de hashing contém espaço para um registro e também possui um ponteiro para o começo de uma lista encadeada. A alocação dinâmica de memória está na criação e na manipulação da lista encadeada, que nada mais é do que uma estrutura de dados simples e que permite adição e remoção eficiente de registros.

Quando um registro é mapeado para uma posição da tabela que já está ocupada, ocorre o que é conhecido como colisão. Neste método, quando ocorre uma colisão, o registro é inserido no final da lista encadeada do ponteiro cujo índice é dado pela função hash. Apesar de ser um método bastante flexível e proporcionar uma tabela onde é possível armazenar um número arbitrário de registros, é preciso ter cuidado, pois as operações de alocação e liberação de memória podem ocasionar vazamentos, algo que não é desejável.

O próximo método hashing a ser estudado é o de encadeamento explícito com o uso de ponteiros usando alocação estática de memória. Note que, por ser uma implementação com alocação estática, significa que já reservamos o espaço na memória de forma antecipada e este não é alterado durante a execução do programa. Nesta implementação, os registros podem ser vistos como nós, onde cada nó armazena um valor e um ponteiro para o próximo nó da sequência. Esta abordagem é mais limitada, já que ela só consegue alocar registros no tamanho exato da tabela.

O processo de inserção de registros em uma tabela com alocação estática segue o mesmo princípio visto anteriormente, o ***h(k)*** é calculado e o resultado é o índice da tabela o qual o registro será inserido. Porém, ao ocorrer uma colisão, duas coisas precisam ser analisadas. Primeiramente, se o nó de índice ***h(k)*** estiver ocupado por um registro mas não possuir um ponteiro para outro nó, o algoritmo busca, na tabela, o último índice livre e passível de inserção, percorrendo os índices de baixo para cima, ou seja, de ***n-1*** à ***0***, sendo ***n*** o tamanho da tabela. Ao encontrar, atualiza-se o ponteiro do nó anterior para apontar para este índice livre que foi encontrado e, por fim, o registro é inserido. Por outro lado, toda vez que um nó da tabela for acessado e este apontar para outro nó, significa que há ali uma espécie de lista encadeada dentro da própria tabela, lista essa que você deve seguir até encontrar o nó que irá guardar o ponteiro índice do último nó livre na tabela, de forma que o encadeamento seja mantido.

O próximo método abordado será, sem dúvidas, o mais simples. Trata-se do hashing com endereçamento aberto e sondagem linear. Nesta estratégia, você faz a inserção do registro na tabela conforme o índice resultante de ***h(k)*** e, caso haja colisão, tudo que você precisa fazer é pular para o próximo índice e, caso ele esteja livre, insere-se o registro, caso não esteja, segue para o próximo índice e assim por diante. Caso você chegue ao último índice, não tenha encontrado uma posição livre e a tabela não esteja cheia, você recomeça a sondagem a partir do índice 0. No fim, você estará percorrendo a lista de forma sequencial e circular, posição por posição, até encontrar um espaço livre para realizar a inserção.

Uma técnica simples e que dispensa o uso de estruturas auxiliares, mas que a longo prazo pode se tornar bastante custosa, já que, quanto mais cheia a tabela estiver, mais sondagens e acessos você precisará fazer, até que finalmente encontre uma posição vazia. O excesso de colisões irá implicar uma degradação no desempenho desta estrutura de hashing.

Por fim, o último método a ser implementado e analisado será o de endereçamento aberto com hashing duplo. Novamente, temos o mesmo princípio baseado na inserção cujo índice será dado por ***h(k)***. Porém, desta vez, para lidar com as colisões, uma função ***h2(k)*** será aplicada sobre o elemento **k**. Função que:

- $retorna\ 1,\ se\ k\ <\ m,\ sendo\ m\ o\ tamanho\ da\ tabela$
- $retorna\ \lfloor (k/m)\ \%\ m \rfloor,\ se\ k\ >=\ m,\ sendo\ m\ o\ tamanho\ da\ tabela$

Sendo assim, o que o algoritmo de uma implementação de double hashing faz é tentar inserir o registro no índice resultando do ***h(k)***, mas caso a posição já esteja ocupado, ele usa o valor de ***h2(k)*** para percorrer pela tabela, ou seja, sonda sequencialmente incrementando ***h2(k)*** vezes até encontrar a próxima posição que não esteja ocupada por nenhum registro. Por exemplo, vamos supor que você deseja inserir o registro 38 em uma tabela de tamanho 11, temos que ***h(38) = 5*** e ***h2(38) = 3***, entretanto a tabela no índice 5 já está ocupada pelo registro 16. O próximo passo será analisar o índice de número 8, já que estaremos incrementando a sondagem com o valor encontrado em ***h2(k)***.

Explicação dos experimentos

Todo o projeto foi desenvolvido em uma única etapa, utilizando a linguagem de programação C++. Os quatro métodos objeto de estudos foram implementados de forma que com apenas uma execução os quatro resultados sejam apresentados para o usuário, sem a necessidade de executar outros códigos em paralelo. Ao executar o projeto, a primeira entrada será referente ao elemento ***m***, o tamanho das quatro tabelas hashing que serão criadas. Em seguida, o programa aceitará como input números inteiros positivos até que um EOF seja identificado.

Para cada método hashing implementado no projeto, ocorre o cálculo e a impressão da média de acessos, uma informação imprescindível que ajuda a compreender o quanto ainda pode ser saudável realizar inserções naquela tabela. Neste ponto, vale ressaltar que o índice que mede o quanto a tabela está cheia é o fator de carga, e este também deve ser regularmente checado. Por exemplo, se você tiver uma tabela com baixo fator de carga, mas com uma média de acessos muito alta, significa que ou seus registros possuem um viés número e portanto resultando sempre no mesmo ***h(k)*** ou há algum problema na implementação que não está garantindo um espalhamento eficiente dos registros.

Para cada conjunto de entradas provido pelo professor, o programa foi executado. Com os resultados apresentados, isto foi salvo em uma tabela para posterior análise detalhada. Contudo, mesmo durante as execuções, era perceptível que alguns métodos entregavam uma melhor performance conforme o tamanho da tabela e a quantidade de registros inseridos iam crescendo. Por exemplo, o hashing com sondagem linear, em comparação com os outros, foi o que apresentou maior índice de média de acessos, mesmo com tabelas de tamanho diferente e com conjuntos de registros diferentes. Nesse sentido, cabe concluir que, apesar de ser fácil de implementar e resolver o problema apresentado de forma correta, a estratégia de sondagem linear acaba se tornando ineficiente e custosa, pois se há ocorrência de muitas colisões, várias requisições de acesso terão que ser feitas na tabela até que o elemento buscado seja encontrado.

Sobre os métodos com alocação de memória, foi possível observar que, para alguns conjuntos de dados, eles mantiveram a mesma média de acessos, porém, quanto maior era a quantidade de registros inseridos, mais memória era alocada e, portanto, a execução do programa se tornava um pouco mais lenta, em termos de

hardware. Além disso, é justo citar os problemas e as dificuldades em se trabalhar com ponteiros e alocação de memória, já que a todo momento seu código precisa dar conta de gerenciar a alocação e a liberação da memória que está sendo utilizada.

Esperava-se, antes da execução do projeto, que o método mais eficiente fosse o de double hashing e, para os conjuntos de entrada testados, ele até que não decepcionou. Para o maior conjunto de dados inseridos, ou seja, ***907*** registros em uma tabela de tamanho ***m = 997*** e fator de carga equivalente a 91%, a média de acessos dos registros ficou em ***2.7***, um número que é bem interessante, visto que isso significa que podemos encontrar a maioria dos registros com menos de 3 interações.

Apresentação e discussão dos resultados encontrados

Para fim de melhor apresentação e discussão dos resultados, foi disposto em duas tabelas – separadas de acordo com o tamanho da tabela hashing (m) – o resultado de média de acessos nos quatro métodos implementados - encadeamento explícito com alocação dinâmica, encadeamento explícito com alocação estática, sondagem linear (Linear Probing) e duplo hashing (Double Hashing) - para cada variação de Fator de Carga.

Tabela 1 – Média de acessos para tabela de tamanho 11

m	n	Fator de Carga	Encadeamento Explícito Dinâmico	Encadeamento Explícito Estático	Linear Probing	Double Hashing
11	2	18%	1	1	1	1
11	3	27%	1	1	1	1
11	4	36%	1	1	1	1
11	5	45%	1,6	1,6	1,6	1,4
11	6	55%	1,5	1,5	1,8	1,8
11	7	64%	1,1	1,1	1,3	1,1
11	8	73%	1,4	1,4	1,4	1,8
11	9	82%	1,3	1,6	1,8	1,4
11	10	91%	1,8	2,4	2,5	2,2

A apresentação dos resultados para a tabela hashing de tamanho 997 foi reduzida aos fatores de carregamento semelhantes aos registrados na Tabela 1 de forma a permitir uma apresentação objetiva e de fácil leitura para comparação entre os valores de média de acessos sob mesmo fator de carga e diferentes valores de m (tamanho da tabela hashing).

Tabela 2 – Média de acessos para tabela de tamanho 997

m	n	Fator de Carga	Encadeamento Explícito Dinâmico	Encadeamento Explícito Estático	Linear Probing	Double Hashing
997	199	20%	1,1	1,1	1,1	1,1
997	269	27%	1,1	1,1	1,2	1,2
997	358	36%	1,2	1,2	1,3	1,2
997	448	45%	1,2	1,3	1,4	1,3
997	548	55%	1,2	1,3	1,5	1,4
997	638	64%	1,3	1,4	2,2	1,6
997	727	73%	1,3	1,5	2,2	1,7

997	817	82%	1,4	1,6	2,8	2
997	907	91%	1,4	1,6	4,9	2,7

Os valores de média de acessos para os fatores de carga abaixo de 60% são explicados pela menor chance de haver colisão à medida que temos muito mais registros disponíveis que chaves a serem alocadas.

Importante observar que esta definição está sempre sujeita à composição das chaves e adequação da função hashing, isto é, a eficiência da função hashing em dispersar as chaves entre os registros de forma uniforme.

Levitin define para encadeamento explícito em “Introduction to the design & analysis of algorithms” [Levitin]:

> *The ratio α = n/m, called the load factor of the hash table, plays a crucial role in the efficiency of hashing. In particular, the average number of pointers (chain links) inspected in successful searches, S, and unsuccessful searches, U, turns out to be S ≈ 1 + α/2 and U = α*
>
> [O quociente α = n/m, chamado fator de carga da tabela hash, desempenha um papel essencial na eficiência do hashing. Em particular, o número médio de ponteiros (encadeamentos) verificados em buscas de sucesso, S, e buscas sem sucesso, U, é S ≈ 1 + α/2 and U = α]

Ao calcular o valor teórico S para os dados experimentais e comparar com os valores obtidos em teste, conclui-se que a função hashing realizou uma distribuição satisfatória dos dados, expressando média próxima ao valor teórico, vide Tabela 3.

Tabela 3 – Comparação Média de acessos teórica e prática do encadeamento explícito com o uso de ponteiros usando alocação estática

m	**n**	**Fator de Carga**	**Média de acessos**	**S**
11	3	27%	1	1,1
11	4	36%	1	1,2
11	5	45%	1,6	1,2
11	6	55%	1,5	1,3
11	7	64%	1,1	1,3
11	8	73%	1,4	1,4
11	9	82%	1,3	1,4
11	10	91%	1,8	1,5
997	199	20%	1,1	1,1

997	269	27%	1,1	1,1
997	358	36%	1,2	1,2
997	448	45%	1,2	1,2
997	548	55%	1,2	1,3
997	638	64%	1,3	1,3
997	727	73%	1,3	1,4
997	817	82%	1,4	1,4
997	907	91%	1,4	1,5

Já para a Sondagem Linear, devido às colisões serem tratadas de forma diferente, o valor teórico é calculado por $S \approx 1/2(1+1/(1 - \alpha))$ [Levitin]. Desta forma, o crescimento do fator de carga implica em crescimento exponencial da média de acessos esperado. Este resultado foi expresso também na prática. Apesar de apresentar para os dois tamanhos de tabela valor inferior ao esperado – variação que pode ser explicada pela adequação da função hashing, pelo conjunto de dados e valor de m (tamanho da tabela) – segundo mostra a tabela 4, ao plotar o gráfico de média de acessos por fator de carga para cada método e m=997 é expresso o crescimento da média de acessos da sondagem linear em comparação com os outros métodos.

Tabela 4 – Comparação Média de acessos teórica e prática da Sondagem Linear

m	**n**	**Fator de Carga**	**Média de acessos**	**S**
11	3	27%	1	1,2
11	4	36%	1	1,3
11	5	45%	1,6	1,4
11	6	55%	1,8	1,6
11	7	64%	1,3	1,9
11	8	73%	1,4	2,3
11	9	82%	1,8	3,3
11	10	91%	2,5	6,0
997	199	20%	1,1	1,1
997	269	27%	1,2	1,2
997	358	36%	1,3	1,3
997	448	45%	1,4	1,4
997	548	55%	1,5	1,6
997	638	64%	2,2	1,9
997	727	73%	2,2	2,3
997	817	82%	2,8	3,3

997	907	91%	4,9	6,0

Os resultados expressos no gráfico 1 estão de acordo com a literatura. Apesar de não haver cálculo teórico de média de acessos para endereçamento direto com alocação de memória estática e duplo Hashing, os resultados podem ser comparados com, respectivamente, endereçamento direto com alocação de memória dinâmica e sondagem linear. Enquanto os valores para os métodos de encadeamento direto são esperados semelhantes, o duplo hashing é esperado sobressair à sondagem linear.

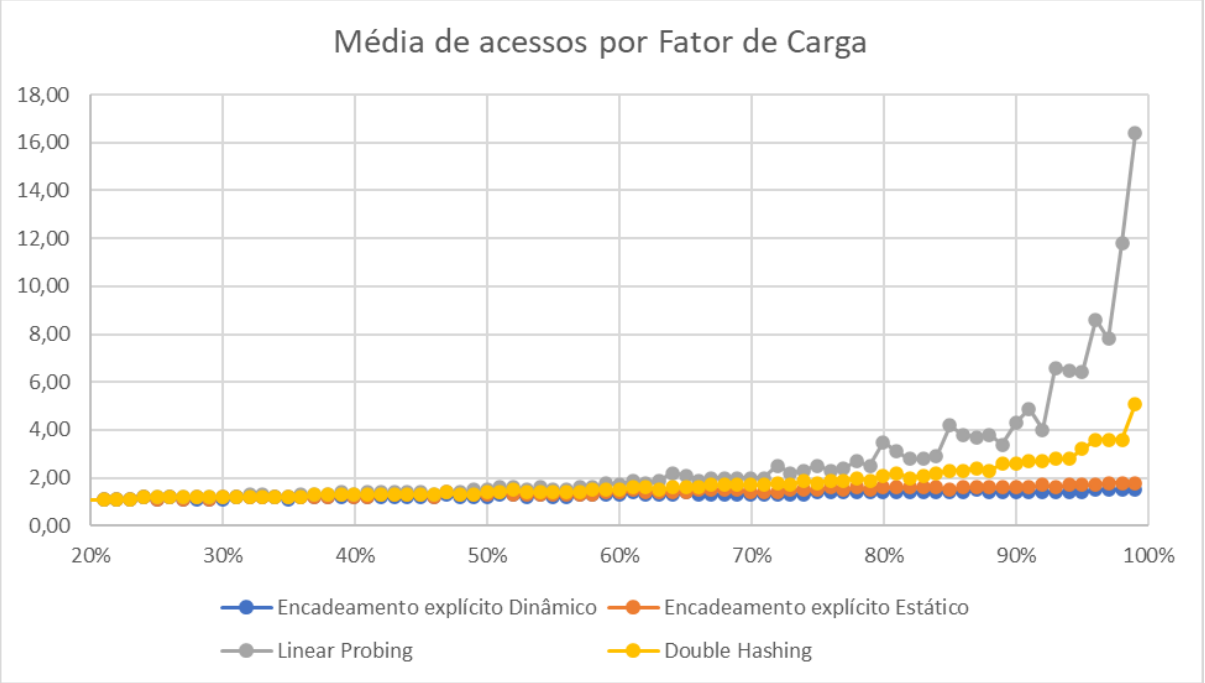

Gráfico 1 - Médias de acessos por fator de carga para tamanho de tabela 997

Conclusão

Sendo assim, concluímos com este estudo que os métodos de hashing são estrategicamente específicos quando desejamos ou temos a demanda de fazer um armazenamento eficiente de informações. As funções de hash, juntamente com as tabelas, tentam garantir um espalhamento uniforme dos registros, de modo que a cada registro que seja necessário acessar, isso ocorra com o mínimo possível de acessos.

Além dos métodos estudados e apresentados aqui, existem muitos outros, cada um com suas especificidades e diferentes implementações. Com certeza, há outros que, inclusive, são mais eficientes e podem alcançar resultados ainda mais satisfatórios.

Referências

[Tharp] A.L.Tharp. **File Organization and Processing**. John Wiley & Sons. 1988

[Cormen et al.] T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein. **Introduction to Algorithms - Third Edition**. MIT Press. 2009

[Levitin] Levitin, Anany. **Introduction to the design & analysis of algorithms** - 3rd ed. Pearson. 2012