Hash

1. Introdução

Uma tabela de espalhamento, tabela de dispersão ou **tabela hash** (hash table) é um vetor no qual cada uma das posições armazena zero, um ou mais elementos. Considere que cada elemento tem uma chave e outros valores associados. Por exemplo, o cadastro dos alunos de uma turma pode ter como chave o número de mátricula do aluno e o valor associado o nome do aluno.

Além do vetor, a estrutura de dados Hash também inclui uma função hash, responsável por converter a chave de identificação do elemento no índice do vetor que indica a posição onde o elemento deve ser encontrado.

- **Exemplo 1.** Imagine um pequeno país (com bem menos de 100 mil cidadãos) onde os números de CPF têm apenas 5 dígitos decimais, não consecutivos.

Chave	**Valor associado**
01555	Rivelino
01567	Pelé
…	…
70444	Ronaldo
71899	Roberto Carlos
95225	Neymar

  Qual uma boa forma de armazenar os dados?
  Resposta: Uma boa forma de armazenar os dados é um vetor com 100.000 posições para armazenar o CPF e outro vetor de strings de 100.000 posições para armazenar os nomes.
  Nessa solução o vetor é conhecido como **tabela hash** e terá muitas posições vagas (desperdício de espaço), mas a busca e a inserção de elementos serão extremamente rápidas.

- **Exemplo 2.** Imagine uma lista encaeada ordenada cujas chaves são nomes de pessoas.

chave	**valor associado**
Antonio Silva	8536152
Arthur Costa	7210629
Bruno Carvalho	8536339
...	...
Vitor Sales	8535922
Wellington Lima	5992240
Yan Ferreira	8536023

  Para acelerar as buscas, divida a lista em 26 pedaços. Os nomes que começam com “A”, os nomes que começam com “B”, etc. Nesse caso, **o vetor de 26 posições é a tabela hash** e **cada posição do vetor aponta para o começo de uma das listas**.

2. Funções hash

Uma tabela de dispersão ou tabela de hash (hash table) é um vetor no qual cada uma de cujas posições armazena zero, uma, ou mais chaves (e valores associados).

Parâmetros importantes:

- `M` : número de posições na tabela de hash
- `N` : número de elementos a serem armazenados

- α (alpha) = `N/M` : fator de carga (load factor)

Função de espalhamento ou função de hashing (hash function): transforma cada chave em um índice da tabela de hash. A função de hashing responde a pergunta "Em qual posição da tabela de hash devo colocar esta chave?".

A função de hashing *espalha* as chaves pela tabela de hash.

A função de hashing associa um *valor hash* (hash value), entre `0` e `M-1`, a cada chave.

No exemplo dos CPFs temos $\alpha < 1$ e a função de hashing é a identidade.
No exemplo dos nomes de pessoas temos $\alpha > 1$ e a função de hashing é

```
nome[0] - 'A';
```

A função de hashing produz uma **colisão quando duas chaves diferentes têm o mesmo valor hash** e portanto são levadas na mesma posição da tabela de hash:

No Exemplo 2, as chaves "Antônio Silva" e "Arthur Costa" começam com a mesma letra "A", sendo convertidas para a mesma chave hash "0", indicando a mesma posição da tabela de hash.

Quando ocorre uma colisão entre duas chaves, é necessário encontrar um outro lugar vago para a referida chave, que é tratado na seção Tratamento de Colisões.

A função hash deve espalhar os dados de modo mais uniforme possível na tabela de espalhamento. Por exemplo, se queremos escolher uma função hash para distribuir os dados dos alunos de uma turma com base no número de matrícula em uma tabela hash de 100 posições, não é útil usar os primeiros dois dígitos da chave.

Chave	**Nome**
201914402	Altair
201914423	Natanael
202014401	Adriana
202014402	Alexandre
202014404	Carlos
202014406	Daniel

Observe que se usarmos os dois primeiros dígitos da chave para escolher o índice do vetor, todos os elementos seriam direcionados para a mesma posição da tabela, ou seja, não distribui os elementos pela tabela, dado que o número de matrícula é formado pelo ano de entrada na instituição seguido pelo semestre, pelo número do curso e pelo número do aluno na turma.

Uma opção melhor nesse caso seria usar os dois últimos dígitos.

Note que ainda assim aconteceu uma colisão entre as chaves dos elementos `Altair` e `Alexandre`, com o valor `02`, mas os demais valores são direcionados para posições diferentes da tabela hash.

Para cada tipo de dados o programador deve definir uma função hash e implementá-la.
As **propriedades desejáveis em uma função hash** são:
- Gerar códigos hash entre `0` e `M-1`
- Fácil de ser executada (execução leve, eficiente)
- Distribuição das chaves pela tabela de espalhamento mais próxima possível de uma distribuição uniforme (onde todas as posições da tabela tem a mesma probabiliade de receber elementos).

Uma função hash frequentemente utilizada é a função resto da divisão (em linguagem c: `%`).

Para nossos exemplos didáticos, utilizaremos o resto da divisão por uma potência de 10. Assim fica fácil de localizarmos os dados.
Em conjuntos de dados maiores, os estudos mostram que para grande parte dos problemas o resto da divisão por um número primo costuma produzir uma distribuição um pouco mais uniforme.

3. Tratamento de colisões

Quando duas chaves são transformadas pela função hash no mesmo código hash (índice do vetor ou tabela hash), dizemos que há uma colisão de chaves.
Para acomodar os elementos que são direcionados para a mesma posição da tabela em que já há outro elemento anterior (colisão), existem duas abordagens clássicas: Armazenamento interno e Lista de colisões.

3.1. Armazenamento interno. Quando ocorre uma colisão e a tabela de espalhamento é maior que a quantidade de dados ($\alpha < 1$), pode-se armazenar o elemento na próxima posição livre da tabela.

Por exemplo, se queremos escolher uma função hash para distribuir os dados dos alunos de 40 alunos de uma turma com base no número de matrícula em uma tabela hash de 100 posições, e usamos os dois últimos dígitos da chave como função hash, produziremos:

Chave	**Nome**	**Código hash**
201914402	Altair	02
201914423	Natanael	23
202014401	Adriana	01
202014402	Alexandre	02
202014403	Carlos	03
202014406	Daniel	06

O armazenamento na tablela hash será:

Posição	**Chave**	**Nome**
0		
1	202014401	Adriana
2	201914402	Altair
3	202014402	Alexandre
4	202014403	Carlos
5		
6	202014406	Daniel
7		
:		
23	201914423	Natanael
:		
99		

Observe que houve uma colisão na posição 2, com as chaves 201914402 e 202014402.

Nesse caso, o primeiro inserido (`Altair`) ocoupa a posição `2`, e o segundo (`Alexandre`) vai para a primeira posição vaga (`3`). O mesmo aconteceu com o código `202014403` do `Carlos`, que a função hash produziu o código `3`, mas essa posição já estava ocupada. Assim, o registro do `Carlos` é alocado na posição `4` que estava livre.

3.2. Armazenamento em listas encadeadas. Outra forma de armazenar os dados, especialmente quando há colisões é usar a tabela hash como um vetor de listas, armazenando os elementos em listas encadeadas, sendo uma lista encadeada para cada código gerado pela função hash (índice da tabela hash). Essa é a solução usada no exemplo 2 da introdução.

Para o mesmo conjunto de dados e função hash usados na opção por armazenamento interno, se quisermos usar o armazenamento em listas encadeadas, considerando uma tabela hash com 10 elementos, e a função hash retornando como código somente o último dígito da chave, o resultado será:

Chave	**Nome**	**Código hash**
201914402	Altair	2
201914423	Natanael	3
202014401	Adriana	1
202014402	Alexandre	2
202014403	Carlos	3
202014406	Daniel	6

A tabela hash vazia é ilustrada na figura 1.

O armazenamento na tablela hash depende do código usado para a inserção de elementos na lista. O código mais simples é a inserção no início da lista. Por esse código, os três primeiros elementos inseridos serão `Altair` (na lista 2), `Natanael` (na lista 3) e `Adriana` (na lista 1). O resultado após a inserção desses três elementos é ilustrado na figura 2:

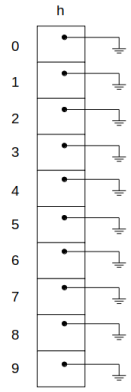

Figura 1. Tabela hash h vazia.

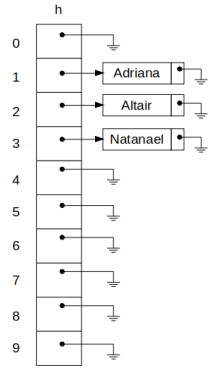

Figura 2. Tabela hash h após a inserção de Altair, Natanael e Adriana.

Até esse ponto a inserção é simples pois não houve colisão. Assim, cada elemento é inserido na lista indicada pelo código hash (resultado da aplicação da função hash sobre a chave).

A partir da inserção do quarto elemento (`Alexandre`), ocorre a primeira colisão, pois o código gerado (`2`) colide com o código do `Altair`. Nesse caso, o novo registro também é inserido na lista `2`. O mesmo acontece com o quinto registro (`Carlos`), cujo código (`3`) colide com o código do `Natanael`. Já o último registro (`Daniel`) produz um código ainda vago (`6`), logo será inserido na lista `6` que ainda está vazia. Como estamos optando pela inserção no início da lista, o resultado final, da inserção dos últimos 3 registros é ilustrado na figura 3.

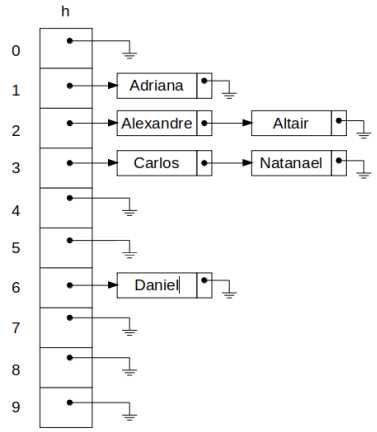

Figura 3. Tabela hash h ao final da inserção de todos os dados

Obviamente, o código para a implementação de listas encadeadas pode ser incorporado ao hash. Como já implementamos essas estruturas de dados, podemos simplesmente reaproveitar o código das listas para implementar o hash.

Observe que nada impede de usarmos o código de árovre binária no lugar do código de listas, caso queiramos otimizar o acesso aos dados no caso de colisões. Em geral, isso não é muito discutido, porque o objetivo da estrutura de dados hash é ter poucas colisões, o que não justificaria a maior complexidade do código de árvore. Entretanto, sem algumas situações pode ser útil, especialmente quando o fator de carga α é maior que 1 e o número de colisões pode ser significativamente elevado.

4. Análise da estrutura de dados Hash

4.1. Aplicação. A estrutura de dados hash é útil para armazenar dados de modo que a **busca seja muito eficiente**. Isso decorre da propriedade bastante interessante de acesso direto ao dado (ou muito próximo ao dado), **se o fator de carga α for baixo** (preferencialmente menor que 1, ou seja, há mais espaço disponível do que dados) **e a função hash conseguir uma boa distribuição** dos dados (poucas colisões). Nessa situação, mesmo para um número razoavelmente grande de dados, o número de comparações necessárias para encontrar uma informação no hash pode ser, em média, menor que 2.

Em termos comparativos, se tivermos um conjunto com 100.000 dados armazenados em uma lista encadeada ordenada, em uma árvore binária balanceada e em um hash com 200.000 posições e boa distribuição, os números de comparações necessários para localizar um elemento na estrutura, no caso médio e no pior caso para cada estrutura são:

Estrutura	**Caso médio**	**Pior caso**
Lista encadeada ordenada	50.000	100.000
Árvore binária balanceada	16	17
Hash com 200.000 posições	1,1	<5

4.2. Limitações. A estrutura de dados hash também possui limitações, e algumas desvantagens em relação à árvore, por exemplo.

Desperdício de espaço. Para conseguir uma boa eficiência na busca, frequentemente há uma necessidade de muitos espaços vazios, o que gera desperdício de espaço.

Tamanho do conjunto de dados. A quantidade de elementos deve ser estimada antecipadamente, para alocar o tamanho da tabela hash. Isso funciona bem para casos onde a quantidade de elementos é bem conhecida. Por exemplo, a quantidade de alunos de uma escola tem uma limitação física, fácil de estimar. Mas a quantidade de clientes de uma loja virtual é muito mais difícil de estimar. Caso a quantidade de elementos seja superior à prevista, a eficiência da estrutura diminui, uma vez que aumentam as colisões. Caso o tamanho da tabela hash seja superestimada, há um desperdício de espaço de memória. As listas encadeadas e as árvores não apresentam essa limitação.
Ordenação. A estrutura de dados hash distribui os dados na tabela hash de acordo com a função hash, que objetiva um espalhamento mais uniforme possível. Ela é otimizada para busca, mas não serve para ordenação dos dados.