INSTITUTO FEDERAL
Sudeste de Minas Gerais
Campus Manhuaçu

Estruturas de Dados I
Tabela Hash

Prof. Leonardo C. R. Soares - *leonardo.soares@ifsudestemg.edu.br*
Instituto Federal do Sudeste de Minas Gerais

18 de novembro de 2023

Introdução

Problema

Procurar uma informação desejada com base na comparação de suas chaves é computacionalmente custoso. Uma abordagem simplória possui custo $\mathcal{O}(n)$.

Abordagens mais eficientes exigem que os dados estejam ordenados. O custo para se ordenar um conjunto de dados, no melhor caso, é $\mathcal{O}(nlogn)$ - excetuando-se ordenação por contagem. O custo de uma busca por comparações de chaves, neste cenário, seria $\mathcal{O}(logn)$.

Introdução

Solução ideal

A melhor solução possível seria acessarmos diretamente o elemento procurado $\mathcal{O}(1)$, sem a necessidade de compararmos o valor das chaves.

Introdução

Solução ideal

A melhor solução possível seria acessarmos diretamente o elemento procurado $\mathcal{O}(1)$, sem a necessidade de compararmos o valor das chaves.

Isso poderia ser facilmente implementado utilizando-se arranjos para armazenar os elementos, **desde que o valor da chave fosse igual ao índice do vetor**.

Introdução

Solução ideal

A melhor solução possível seria acessarmos diretamente o elemento procurado $\mathcal{O}(1)$, sem a necessidade de compararmos o valor das chaves.

Isso poderia ser facilmente implementado utilizando-se arranjos para armazenar os elementos, **desde que o valor da chave fosse igual ao índice do vetor**.

Mas isso é diretamente viável?

Introdução

Solução ideal

A melhor solução possível seria acessarmos diretamente o elemento procurado $\mathcal{O}(1)$, sem a necessidade de compararmos o valor das chaves.

Isso poderia ser facilmente implementado utilizando-se arranjos para armazenar os elementos, **desde que o valor da chave fosse igual ao índice do vetor**.

Mas isso é diretamente viável? Se a chave for o CPF, por exemplo, 987.443.546-90, isso seria um índice aceitável para um vetor em qualquer linguagem de programação?

Introdução

Solução ideal

A melhor solução possível seria acessarmos diretamente o elemento procurado $\mathcal{O}(1)$, sem a necessidade de compararmos o valor das chaves.

Isso poderia ser facilmente implementado utilizando-se arranjos para armazenar os elementos, **desde que o valor da chave fosse igual ao índice do vetor**.

Mas isso é diretamente viável? Se a chave for o CPF, por exemplo, 987.443.546-90, isso seria um índice aceitável para um vetor em qualquer linguagem de programação? E para os casos em que a chave for uma *string*?

Introdução

Solução ideal

A melhor solução possível seria acessarmos diretamente o elemento procurado $\mathcal{O}(1)$, sem a necessidade de compararmos o valor das chaves.

Isso poderia ser facilmente implementado utilizando-se arranjos para armazenar os elementos, **desde que o valor da chave fosse igual ao índice do vetor**.

Mas isso é diretamente viável? Se a chave for o CPF, por exemplo, 987.443.546-90, isso seria um índice aceitável para um vetor em qualquer linguagem de programação? E para os casos em que a chave for uma *string*? Por favor, deixe PHP fora dessa discussão.

Tabela hash

Solução

Para que seja possível utilizarmos arranjos neste contexto, precisaremos mapear o valor da chave em um índice do vetor. Essa função de conversão é chamada de *função hash*.

Função hash

Características

- A função *hash* deve ser determinística. Para uma determinada chave a função sempre retorna o mesmo valor de *hash*.
- Para ser utilizada como uma função de indexação, a função de *hash* deve sempre retornar um valor de *hash* dentro dos limites do arranjo $[0, N-1]$, onde N é o tamanho do arranjo.
- Espalhamento uniforme: Todos os índices do arranjoo devem ter aproximadamente a mesma chance de serem mapeados pela função *hash*. Essa característica é importante para distribuir os elementos uniformemente pelo arranjo.
- Deve ser executada em $\mathcal{O}(1)$.

Função hash

Colisões

Quando duas (ou mais) chaves são mapeadas para um mesmo índice do arranjo, temos uma **colisão**. As colisões precisam ser tratadas. Temos duas possibilidades:

Função hash

Colisões

Quando duas (ou mais) chaves são mapeadas para um mesmo índice do arranjo, temos uma **colisão**. As colisões precisam ser tratadas. Temos duas possibilidades:

- Desenvolver uma função hash que não mapeie duas chaves diferentes para um mesmo índice (se você conseguir, me avisa);
- Tratar as colisões.

Resolução de colisões

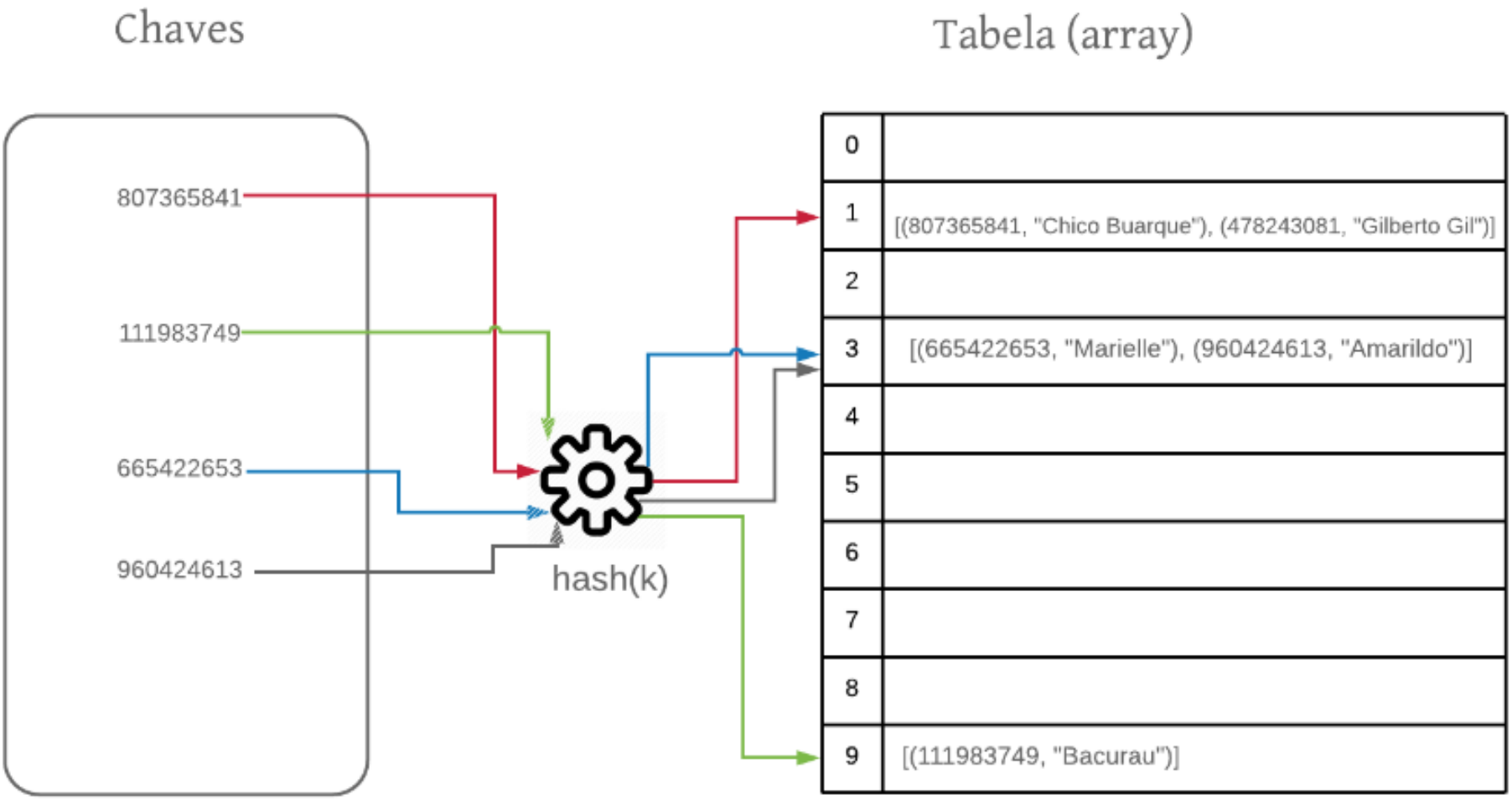
Existem duas estratégias principais para lidar com colisões: resolução de colisões por encadeamento e resolução de colisões por endereçamento aberto.

Resolução de colisões

Resolução de colisões por encadeamento

Ao invés de armazenarmos o objeto diretamente no arranjo, cada posição do arranjo conterá uma lista de objetos. Esta lista será composta por todos os objetos que contenham o mesmo valor de *hash* (**não a mesma chave**).

Resolução de colisões por encadeamento



Resolução de colisões por encadeamento

Para que as operações com este tipo de solução sejam eficientes, precisamos que a função hash espalhe os elementos pela arranjo da forma mais uniforme possível. Se todos as chaves forem mapeadas para o mesmo elemento (pior caso possível), uma operação de busca teria custo de execução limitado por $\mathcal{O}n$. Com uma boa função de espalhamento (executada em tempo constante), conseguimos reduzir o custo desta operação para $T(n) = 1 + \alpha$, onde α é o tamanho médio das listas.

Resolução de colisões por encadeamento

Hash modular

Uma forma simples de elaborarmos uma tabela *hash* é utilizar como índice do arranjo o valor da chave mod M. Onde M é o tamanho da tabela. Entretanto, quando utilizarmos tal estratégia, é importante que M seja um número primo para diminuirmos a quantidade de colisões.

Hash Modular

key	hash ($M = 100$)	hash ($M = 97$)
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
510	10	25
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22
304	4	13
714	14	35
857	57	81
801	1	25
900	0	27
413	13	25
701	1	22
418	18	30
601	1	19

Modular hashing

Resolução de colisões por encadeamento

Baixe o exemplo aqui.

Resolução de colisões

Resolução de colisões por endereçamento aberto

Ao invés de utilizar listas para tratar as colisões, nesta abordagem, sempre que você tentar armazenar um objeto em uma posição já ocupada, o algoritmo tentará encontrar outra posição, que esteja livre, para armazenar o objeto.

Resolução de colisões

Resolução de colisões por endereçamento aberto

Ao invés de utilizar listas para tratar as colisões, nesta abordagem, sempre que você tentar armazenar um objeto em uma posição já ocupada, o algoritmo tentará encontrar outra posição, que esteja livre, para armazenar o objeto.

Existem diversas formas de se implementar, vejamos uma implementação bem simples:

Resolução de colisões por endereçamento aberto

Neste exemplo, tabela2 é um vetor do tipo Livro (e não um ArrayList).

```
public void putAberto(Livro l) throws Exception{
    int salto;
    int hash;
    for(salto = 0; salto<tabela2.length;salto++){
        hash = (hash(l.getISBN()) + salto) % tabela2.length;
        // Se ta livre ou se já foi cadastrado
        if ((tabela2[hash] == null) || tabela2[hash].getISBN().equals(l.getISBN())) {
            tabela2[hash] = l;
            break;
        }
    }
    if (salto == tabela.length)
        throw new Exception(message:"Não há espaço disponível.");
}
```

Exercícios

- Implemente o exemplo utilizando endereçamento aberto para resolver as colisões (GIT).
- Implemente um pequeno programa em Java para cadastro e consulta de Times de Futebol utilizando uma tabela *hash* de tamanho igual a 17. Trate colisões por encadeamento. A classe Time deverá possuir os atributos, codigo, nome, anoFundacao, presidente e técnico. (GIT)


thank you
danke
gracias
merci
obrigado
спасибо
謝謝
ngiyabonga
teşekkür ederim
dziękuję
bedankt
go raibh maith agat
sukriya
kop khun krap
mochchakkeram
terima kasih
grazie
arigatô
dakujem
tapadh leat
dank je
takk
sagolun
hvala
paldies
grazzi
мерси
감사합니다
ありがとう
рахмат
Баярлалаа
хвала
ευχαριστώ
xiexie
rahmet
tanemirt
dhanyavadagalu
shukriya
chnorakaloutioun
gratias ago
gràcies
sulpay
trugarez
mamnun
asante
manana
obrigada
chokrane
murakoze
tenki
djiere dieuf
mahalo
mersi
barka
welalin
tack
spas
kia ora
vinaka
blagodaram
misaotra
matondo
faafetai lava
kiitos
dankie
dhanyavad
nandri
nanni
gracie
bayarlalaa
koszonom
dankon
aciu
akun
enkosi
sobodi
dėkuji
mèsi
didi madloba
najis tuke
kam sah hamnida
rahmat
diolch
merce
taiku
дякую
спасибі

Referências

- BACKES, A. **Notas de aula**. Disponível online em `https://www.facom.ufu.br/~backes/gsi011/Aula07-TabelaHash.pdf`. Acesso em 25 de outubro de 2023.
- BRUNET, J. A. **Estrutura de dados**. Disponível online em `https://joaoarthurbm.github.io/eda/posts/hashtable/`. Acesso em 25 de outubro de 2023.
- ZIVIANI, N. **Projeto de Algoritmos com implementações em Java e C++**, 2007.