Ifes Campus Serra
TII – Técnico Informática para Internet
Programação Orientada a Objetos 1

Introdução a Tabelas de Hashing

1. Motivação: O Problema da Busca Sequencial

Em muitas aplicações, precisamos armazenar e buscar dados rapidamente. Uma abordagem ingênua seria armazenar os dados em um vetor e percorrê-lo sequencialmente para encontrar o elemento desejado. No entanto, esse método tem um desempenho ineficiente, pois a complexidade no pior caso é O(n), tornando-se inviável para grandes quantidades de dados.

A solução para esse problema é o uso de tabelas de hashing, que permitem encontrar elementos em tempo O(1) na maioria dos casos.

2. Introdução ao Hashing

O hashing é uma técnica que usa uma função para mapear um conjunto de chaves a índices em um vetor, permitindo o acesso direto aos dados.

2.1 Função de Hashing Simples

Para ilustrar o conceito, usaremos uma função de hashing simples que opera sobre strings. O hashing será realizado somando os valores ASCII dos caracteres da string:

```c
#include <stdio.h>
#include <string.h>

int hash_simples(char *str) {
    int soma = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        soma += str[i];
    }
    return soma;
}

int main() {
    char *palavra = "teste";
    printf("Hash de %s: %d", palavra, hash_simples(palavra));
    return 0;
}
```

2.2 Armazenando Nomes em um Vetor

Agora, usaremos a função `hash_simples` **para armazenar um conjunto de 10 nomes em um vetor:**

```c
#include <stdio.h>
#include <string.h>

#define TAMANHO_TABELA 10

char *tabela[TAMANHO_TABELA];

int hash_simples(char *str) {
    int soma = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        soma += str[i];
    }
    return soma;
}

void armazenar_nome(char *nome) {
    int indice = hash_simples(nome) % TAMANHO_TABELA;
    tabela[indice] = nome;
}

int main() {
    char *nomes[10] = {"Ana", "Bruno", "Carlos", "Daniela",
                       "Eduardo", "Fernanda", "Gabriel",
                       "Helena", "Igor", "Julia"};

    for (int i = 0; i < 10; i++) {
        armazenar_nome(nomes[i]);
    }

    for (int i = 0; i < TAMANHO_TABELA; i++) {
        if (tabela[i] != NULL) {
            printf("Índice %d: %s
", i, tabela[i]);
        }
    }

    return 0;
}
```

3. Limitando o Resultado ao Tamanho do Vetor

O resultado da função de hash pode gerar valores muito grandes. Para garantir que o índice calculado esteja dentro dos limites do vetor, utilizamos a operação de módulo:

```c
#define TAMANHO_TABELA 10

int hash(char *str) {
    int soma = 0;
    for (int i = 0; str[i] != '\0'; i++) {
        soma += str[i];
    }
    return soma % TAMANHO_TABELA;
}
```

Agora, qualquer string será mapeada para um índice entre 0 e `TAMANHO_TABELA - 1`.

4. Dimensionamento da Tabela e Fator de Carga

A escolha do tamanho da tabela é crucial. O fator de carga (λ) é definido como:

Fator de Carga (fc) = N/M

Onde:

- `N` é o número de elementos armazenados.
- `M` é o tamanho do vetor.

Um fator de carga muito alto aumenta a probabilidade de colisões, reduzindo a eficiência. Um valor adequado é tipicamente menor que 0.7.

Melhor Fator de Carga:

Fator de Carga Ideal: Em geral, um fator de carga entre 0,5 e 0,75 é considerado ideal para a maioria das aplicações. Isso significa que a tabela está ocupada entre 50% e 75% de sua capacidade. Nesse intervalo, a tabela oferece um bom equilíbrio entre a eficiência de uso do espaço e a minimização de colisões.

Impacto no Desempenho:

Fator de carga baixo (< 0,5): Significa que a tabela está pouco preenchida, o que resulta em menos colisões, mas pode levar a um desperdício de memória.

Fator de carga alto (> 0,75): Aumenta o risco de colisões, o que pode prejudicar o desempenho da tabela de hashing, especialmente em operações como inserção e busca.

Pergunta: O que acontece quando 2 chaves distintas produzem o mesmo código de hashing (posição no vetor)?

5. Tratamento de Colisões

Quando duas chaves diferentes resultam no mesmo índice, ocorre uma colisão. Existem diversas técnicas para tratar esse problema. Vamos apresentar duas delas: busca linear e encadeamento.

5.1 Tratamento por Busca Linear

Quando uma posição está ocupada, buscamos a próxima posição disponível:

```c
#include <stdio.h>
#include <string.h>
#define TAMANHO_TABELA 10

char *tabela[TAMANHO_TABELA];

void inserir(char *str) {
    int indice = hash(str);
    while (tabela[indice] != NULL) {
        indice = (indice + 1) % TAMANHO_TABELA;
    }
    tabela[indice] = str;
}
```

No pior caso, ocorre o tratamento cíclico do vetor quando passa-se a verificar posiçõess anteriores à produzidas pela função de hashing.

5.2 Tratamento por Encadeamento

Outra abordagem é armazenar os elementos em listas encadeadas:

É construdo um vetor de listas. Quando 2 elementos submetidos a função de hashing produzem o mesmo resultado (mesma posição do vetor), eles são armazenados na mesma lista.

```c
#include <stdlib.h>
#include "tadlista.h"

Lista tabela[TAMANHO_TABELA];

void inserir_encadeado(char *str) {
    int indice = hash(str);
    appendLista(tabela[indice], str);
}
```

Na busca por um elemento, usa-se o código de hashing para determinar a lista, e a partir daí usase uma busca linear (ou binaria, se a lista é ordenada).

6. Funções de Hashing Mais Eficientes

A função de hash baseada apenas na soma dos valores ASCII pode gerar muitas colisões. Algumas alternativas mais eficientes incluem:

6.1 Hashing Multiplicativo

```c
unsigned int hash_multiplicativo(char *str) {
    unsigned int hash = 0;
    while (*str) {
        hash = (hash * 31) + *str;
        str++;
    }
    return hash % TAMANHO_TABELA;
}
```

6.2 Hashing de DJB2

```c
unsigned int hash_djb2(char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c;
    }
    return hash % TAMANHO_TABELA;
}
```

Essas funções geram distribuições de hash mais uniformes e reduzem colisões.

7. Conclusão

O uso de tabelas de hashing pode melhorar significativamente o desempenho de buscas em comparação com uma busca sequencial. No entanto, é fundamental escolher uma boa função de hash, definir um tamanho adequado para a tabela e tratar eficientemente colisões. O uso dessas técnicas é essencial para otimizar estruturas de dados como dicionários e bancos de dados.