

Tabelas hash

Estrutura de dados II

Prof. Allan Rodrigo Leite

Problema de busca

- A maioria dos métodos de busca usam comparação de chaves
 - Dada uma chave que representa o critério de pesquisa, são realizadas sucessivas comparações sobre os dados mantidos em uma estrutura
- Algoritmos eficientes de busca necessitam dos dados ordenados
 - Limite inferior para problemas de ordenação é $O(N \log N)$
 - Complexidade temporal de algoritmos eficientes de busca é $O(\log N)$
- Uma busca ideal seria um acesso direto
 - Ou seja, sem a necessidade de procurar algo no conjunto de dados
 - Complexidade temporal igual a $O(1)$

Acesso direto em vetores

- São estruturas de dados que utilizam índices para armazenar dados
 - O acesso para leitura e gravação de dados em vetores é $O(1)$
 - Dado um índice, o acesso para leitura e gravação nesse índice é direto
- Porém, vetores não possuem mecanismos para calcular um índice
 - Dado um valor do vetor, como obter diretamente o índice desse valor?
 - Neste caso, a busca não é $O(1)$, mas sim $O(N)$ no pior caso
- Uma alternativa para o problema de busca em vetores é a aplicação de uma estrutura de dados conhecida como tabela hash

Tabelas hash

- Também conhecida como tabela de dispersão ou espalhamento
 - Estrutura de dados não linear que associa chaves de pesquisa a valores
 - É uma generalização do conceito de vetor
 - Dada uma chave é possível fazer uma busca e obter o valor associado
 - O acesso direto aos valores é realizado por uma função hashing
 - Isto é, uma função que espalha os dados na tabela
 - Operações sobre uma tabela hash
 - Gravar um novo valor na estrutura a partir de uma chave
 - Recuperar um valor existente na estrutura a partir de uma chave

Tabelas hash

- Estratégia de armazenamento
 - Suponha que existam n chaves a serem armazenadas em uma tabela T
 - A tabela armazena dados sequencial e possui uma capacidade m
 - Neste caso, usa-se um vetor para armazenar os dados da tabela
 - As posições da tabela se situam no intervalo $[0, m - 1]$
 - A função hashing tem como objetivo espalhar os dados na tabela
 - Os dados serão armazenados de forma dispersa e não ordenada
 - Os dados estarão acessíveis diretamente por intermédio da função hashing

Tabelas hash

- Estratégia de armazenamento (cont.)
 - O objetivo é armazenar cada chave no bloco referente ao seu endereço
 - O endereço é calculado por uma função de transformação (*hashing*)

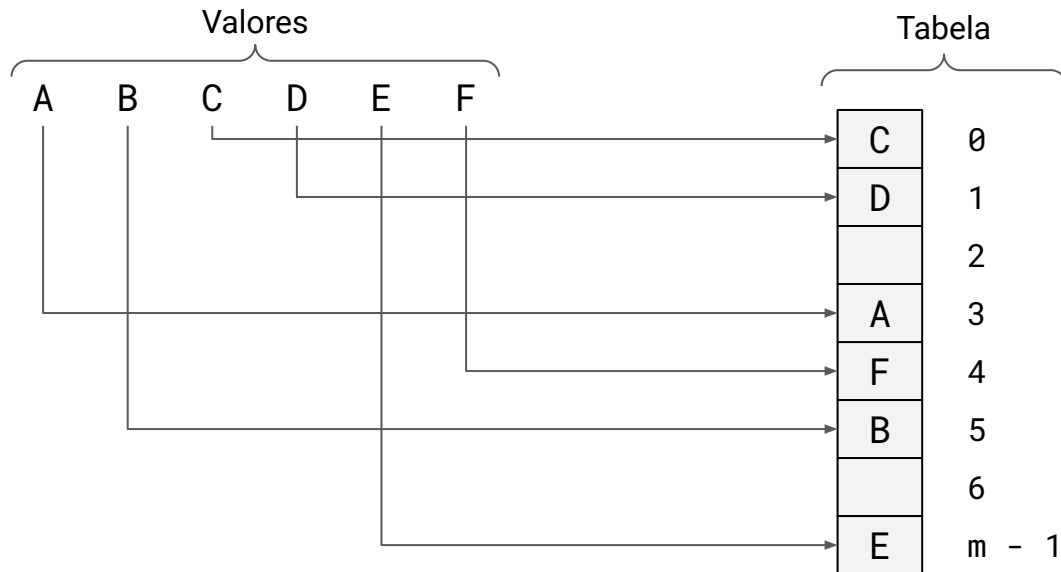
$$E = H(c)$$

Onde:

E = Endereço (índice no vetor)

H = Função hashing

c = Chave



Tabelas hash

- Vantagens

- Tempo de busca é praticamente independente do número de elementos armazenados na tabela
 - Na média a operação de busca tem esforço computacional $O(1)$
- Implementação simples, sendo uma generalização de vetores

- Desvantagens

- Alto custo para recuperar os dados da tabela de forma ordenada
 - Nesse caso é necessário ordenar a tabela
- No pior caso a busca pode ter esforço computacional $O(N)$
 - Relação com a frequência de colisões que a função hashing pode gerar
- Requer a definição do número de elementos que será armazenar
 - Necessário definir o tamanho do vetor de dados

Tabelas hash

- Principais aplicações
 - Cache de dados: bancos de dados em memória
 - Criptografia: algoritmos MD5 e SHA
 - Armazenamento de senhas
 - Verificação de integridade e autenticidade dos dados
 - Tabelas de símbolos ou alfabetos bem definidos: compiladores

Tabelas hash

- Estrutura de uma tabela hash

```
typedef struct {  
    int chave;  
    void* valor;  
} Entrada;
```

```
typedef struct {  
    int total;  
    int tamanho;  
    Entrada** elementos;  
} Hash;
```

Tabelas hash

- Criar uma tabela hash

```
Hash* criar(int tamanho) {  
    Hash* hash = malloc(sizeof(Hash));  
  
    hash->tamanho = tamanho;  
    hash->total = 0;  
    hash->elementos = malloc(tamanho * sizeof(Entrada*));  
  
    for (int i = 0; i < tamanho; i++) {  
        hash->elementos[i] = NULL;  
    }  
  
    return hash;  
}
```

Tabelas hash

- Criar uma tabela hash

```
void liberar(Hash* hash) {  
    if (hash != NULL) {  
        for (int i = 0; i < tamanho; i++) {  
            if (hash->elementos[i] != NULL) {  
                free(hash->elementos[i]);  
            }  
        }  
  
        free(hash->elementos);  
        free(hash);  
    }  
}
```

Tabelas hash

- Como escolher o tamanho de uma tabela hash
 - O ideal é escolher um número primo
 - Números primos reduzem a probabilidade de colisões em funções hashing
 - Inclusive em funções hashing pouco eficazes
 - Evitar valores que sejam potência de 2
 - Valores que são potência de 2 aumentam os problemas de colisão
 - Este efeito é mais sensível em funções hashing mais simples

Funções hashing

- Transforma uma chave em uma posição da tabela de dispersão
 - Princípio da tabela hash é a associação de valores a chaves
 - Chave: parte da informação que compõe o elemento a ser manipulado pela tabela hash
 - Valor: posição (índice) onde o elemento se encontra no vetor de dados que define a tabela hash
- Portanto, a partir de uma chave é possível acessar diretamente o dado na posição do vetor
 - Na média essa operação tem esforço computacional $O(1)$

Funções hashing

- Uma função de dispersão deve idealmente satisfazer às condições
 - Produzir um número baixo de colisão
 - Ser facilmente calculável
 - Ser uniforme, isto é, distribuir de forma equilibrada os dados na tabela
- Colisão de chaves
 - O valor da chave deve gerar um endereço tão único quanto possível
 - Quando houver coincidência, esta situação é chamada de colisão
 - Isto é, duas ou mais chaves diferentes gerando o mesmo endereço

Funções hashing

- Técnicas populares para uma função hashing
 - Método da divisão
 - Método da multiplicação
 - Método da dobra
- Método da divisão
 - Também conhecido como método da congruência linear
 - Consiste em calcular a posição da chave a partir do resto da divisão
 - $H(c) = c \bmod m$

```
int hashingDivisao(int chave, int tamanho) {  
    return (chave & 0x7FFFFFFF) % tamanho;  
}
```

Funções hashing

- Método da multiplicação
 - Também conhecido como método da congruência linear multiplicativo
 - Necessário determinar uma constante A, onde $0 < A < 1$
 - Usa-se esta constante A para:
 - Multiplicar o valor da chave
 - Parte fracionária resultante é multiplicada pelo tamanho da tabela

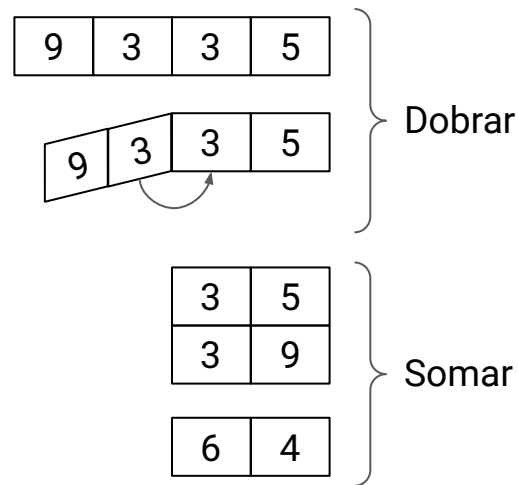
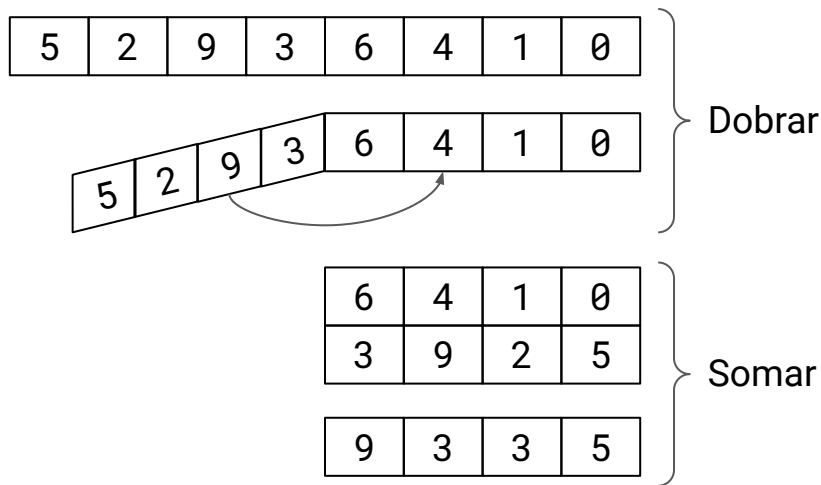
```
int hashingMultiplicacao(int chave, int tamanho) {  
    float A = 0.61803387; //constante  $0 < A < 1$   
    float i = chave * A; //realiza a multiplicação  
    i = i - (int) i; //obtem a parte fracionária  
    return (int) (tamanho * i);  
}
```


Funções hashing

- Método da dobra
 - Utiliza um esquema de dobrar e somar os dígitos do valor para calcular a posição da chave na tabela
 - Estratégia do método
 - Considera uma chave como uma sequência de dígitos escritos em um papel
 - Enquanto a chave for maior do que o tamanho da tabela, o papel é dobrado
 - Os dígitos sobrepostos após a dobra são somados, desconsiderando dezena

Funções hashing

- Método da dobra (cont.)
 - Chave igual a 52936410 e tamanho da tabela igual a 73



Funções hashing

- Método da dobra (cont.)
 - Forma mais fácil de implementar o método utiliza:
 - Operador ou exclusivo
 - Dobra é realizada de k em k bits

```
int hashingDobra(int chave, int tamanho) {  
    int k = 10;  
    int parte1 = chave >> k;  
    int parte2 = chave & (tamanho - 1);  
    return (int) (parte1 ^ parte2);  
}
```

Operações de leitura e gravação sem colisão

- Quando não é necessário tratamento de colisão
 - Utilizado em situações específicas quando o hashing é perfeito
 - Neste cenário colisões não são aceitas
 - Esforço computacional será $O(1)$ sempre em todos os casos
 - Quando conhecida a quantidade e quais dados serão armazenados
 - Exemplo: tabela de símbolos de compiladores
- Operação de inserção
 - Calcula a posição do elemento na tabela a partir da função hashing
 - Armazena o elemento na tabela na posição calculada

Operações de leitura e gravação sem colisão

- Operação de inserção (cont.)

```
void adicionar(Hash *hash, int chave, void* valor) {  
    if (hash != NULL && hash->total < hash->tamanho) {  
        int indice = hashingDivisao(chave, hash->tamanho);  
  
        Entrada *entrada = malloc(sizeof(Entrada));  
        entrada->chave = chave;  
        entrada->valor = valor;  
  
        hash->elementos[indice] = entrada;  
        hash->total++;  
    }  
}
```

Operações de leitura e gravação sem colisão

- Operação de busca
 - Calcula a posição do elemento na tabela a partir da função hashing
 - Verifica a existência de um elemento na posição obtida
 - Caso positivo retorna o elemento encontrado, caso contrário nulo

```
void* buscar(Hash *hash, int chave) {  
    if (hash != NULL) {  
        int indice = hashingDivisao(chave, hash->tamanho);  
        if (hash->elementos[indice] != NULL) {  
            return hash->elementos[indice]->valor;  
        }  
    }  
  
    return NULL;  
}
```

Tratamentos de colisão

- Quando é necessário tratamento de colisão
 - Situações predominantes são hashing imperfeito, sujeitos a colisão
 - Colisões são toleráveis, embora não sejam desejáveis
 - Colisões podem ocorrer devido ao:
 - Tamanho da tabela, isto é, mais chaves para armazenar do que capacidade
 - Valores a serem armazenados na tabela
 - Função hashing utilizada e espalhamento não uniforme
 - Há situações extremas onde cada elemento adicionado gera colisão
 - Ataque de negação de serviço conhecido como hash flooding
 - Esforço computacional é $O(N)$ na leitura dos elementos

Tratamentos de colisão

- Quando é necessário tratamento de colisão (cont.)
 - Não há garantia que uma função hashing produzirá um endereço único
 - Exceto nas condições de um hashing perfeito
 - Portanto, colisões precisam ser tratadas
- Principais técnicas para tratamento de colisão
 - Endereçamento aberto
 - Endereçamento separado

Tratamentos de colisão

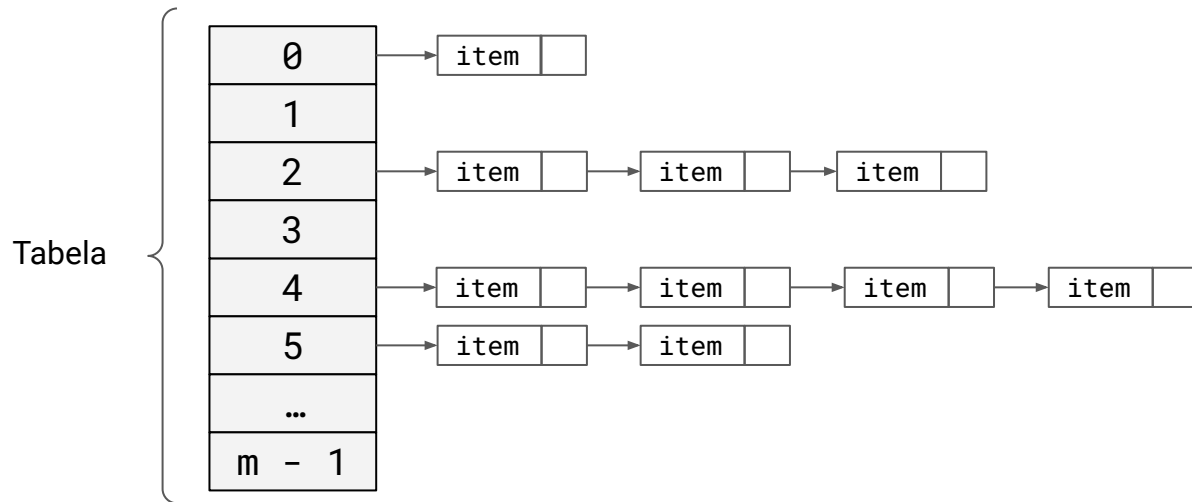
- Endereçamento aberto
 - O elemento com chave colidida é armazenado no primeiro endereço livre
 - Também conhecido como rehash
 - Vantagens
 - Busca é realizada dentro da própria tabela, sendo assim mais rápida
 - Se beneficia do princípio da localidade espacial
 - Não requer espaço adicional além do definido como tamanho da tabela
 - Desvantagens
 - Mais cálculo de posições na adição de elementos, quando detectada colisão
 - Esforço computacional da adição será $O(N)$ no pior caso
 - Neste caso, quando todos elementos adicionados geram colisão

Tratamentos de colisão

- Endereçamento separado
 - O elemento com chave colidida é armazenado em uma lista encadeada
 - Também conhecido como separate chaining
 - Vantagens
 - Esforço computacional da adição sempre será $O(1)$
 - Dado que o esforço para adição em uma lista encadeada é $O(1)$
 - Desvantagens
 - Não requer espaço adicional além do definido como tamanho da tabela
 - Não se beneficia do princípio da localidade espacial
 - Tempo de busca é proporcional ao número de elementos na lista encadeada

Tratamentos de colisão

- Endereçamento separado (cont.)



Operações de leitura e gravação com colisão

- Uso de endereçamento separado para tratamento de colisão

```
typedef struct {  
    int chave;  
    void* valor;  
} Entrada;
```

```
typedef struct no {  
    Entrada *entrada;  
    struct no *proximo;  
} No;
```

```
typedef struct {  
    int total;  
    int tamanho;  
    No** elementos;  
} Hash;
```

Operações de leitura e gravação com colisão

- Uso de endereçamento separado para tratamento de colisão (cont.)

```
Hash* criar(int tamanho) {  
    Hash* hash = malloc(sizeof(Hash));  
  
    hash->tamanho = tamanho;  
    hash->total = 0;  
    hash->elementos = malloc(tamanho * sizeof(No*));  
  
    for (int i = 0; i < tamanho; i++) {  
        hash->elementos[i] = NULL;  
    }  
  
    return hash;  
}
```

Operações de leitura e gravação com colisão

- Operação de inserção com encadeamento separado

```
void adicionar(Hash *hash, int chave, void* valor) {  
    if (hash != NULL) {  
        int indice = hashingDivisao(chave, hash->tamanho);  
  
        Entrada *entrada = malloc(sizeof(Entrada));  
        entrada->chave = chave;  
        entrada->valor = valor;  
  
        No *no = malloc(sizeof(No));  
        no->proximo = hash->elementos[indice];  
        no->entrada = entrada;  
  
        hash->elementos[indice] = no;  
        hash->total++;  
    }  
}
```

Operações de leitura e gravação com colisão

- Operação de busca com encadeamento separado

```
void buscar(Hash *hash, int chave) {  
    if (hash != NULL) {  
        int indice = hashingDivisao(chave, hash->tamanho);  
        No *no = hash->elementos[indice];  
  
        while (no != NULL) {  
            if (no->entrada->chave == chave) {  
                return no->entrada->valor;  
            }  
            no = no->proximo;  
        }  
    }  
  
    return NULL;  
}
```

Exercícios

Dado um vetor de números inteiros v de tamanho n e um número k , retorne verdadeiro se a soma de qualquer par de números em v for igual a k .

Exemplo: dado $v = [10, 15, 3, 7]$ e $k = 17$, a saída deve ser true, pois $10 + 7$ é 17

Resolva o problema considerando:

- Solução com esforço $O(N^2)$: utilizando busca sequencial
- Solução com esforço $O(N \log N)$: utilizando busca pesquisa binária
- Solução com esforço $O(N)$: utilizando tabela hash

Tabelas hash

Estrutura de dados

Prof. Allan Rodrigo Leite