Instituto Federal Goiano
Campus Morrinhos
Núcleo de Computação
Bacharelado em Ciência da Computação

Nome Completo do Autor
João Victor Rocha Vilela Godoi

Introdução a tabelas hash

Trabalho de Graduação Interdisciplinar

Morrinhos
2023

Nome Completo do Autor
João Victor Rocha Vilela Godoi

Introdução a tabelas hash

Morrinhos
2023

Às vezes, quando inova-se, comete-se erros. É melhor admiti-los rapidamente e seguir em frente com a melhoria de suas outras inovações.
Steve Jobs

Resumo

Tabelas hash são uma estrutura de dados eficiente para buscas. Elas são compostas por um vetor de células, cada uma das quais pode armazenar uma chave e um valor. A chave é usada para calcular um índice no vetor, que é então usado para acessar o valor associado.

- O objetivo deste trabalho é apresentar uma revisão da literatura sobre tabelas hash.

- O trabalho começa com uma introdução sobre o que são tabelas hash e como elas funcionam. Em seguida, o artigo discute os principais benefícios e limitações das tabelas hash.

Palavras-chave: Estrutura de dados. Computação. Tabelas Hash.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Tabelas hash são uma estrutura de dados muito importante na ciência da computação. Elas são usadas em uma ampla variedade de aplicações, como banco de dados, algoritmos de busca, algoritmos de ordenação, arquivos e aplicativos de computação gráfica.

O objetivo deste trabalho é apresentar uma revisão da literatura sobre tabelas hash. O trabalho começa com uma introdução sobre o que são tabelas hash e como elas funcionam. Em seguida, o trabalho discute os principais benefícios e limitações das tabelas hash. Por fim, o trabalho apresenta algumas aplicações de tabelas hash.

2 Aplicações

Tabelas hash são estruturas de dados que permitem o armazenamento e a recuperação de dados de forma eficiente. Elas são usadas em uma ampla variedade de aplicações, incluindo:

Bancos de dados: tabelas hash são usadas para armazenar dados em bancos de dados, como nomes de usuários, senhas e endereços de e-mail. Elas permitem que os dados sejam recuperados rapidamente, mesmo quando o banco de dados contém um grande número de registros.

- Compiladores: tabelas hash são usadas para armazenar informações sobre símbolos, como nomes de variáveis, funções e tipos de dados. Elas permitem que o compilador encontre rapidamente os símbolos necessários para gerar código-objeto.
- Sistemas operacionais: tabelas hash são usadas para armazenar informações sobre processos, arquivos e outros recursos do sistema operacional. Elas permitem que o sistema operacional gerencie esses recursos de forma eficiente.
- Redes: tabelas hash são usadas para armazenar informações sobre endereços IP, portas e outros dados de rede. Elas permitem que os roteadores e outros dispositivos de rede comuniquem-se de forma eficiente.
- Em geral, tabelas hash são usadas em aplicações que requerem o armazenamento e a recuperação de dados de forma rápida e eficiente.
- Aqui estão alguns exemplos específicos de como tabelas hash são usadas em aplicações reais:
- Em um banco de dados, uma tabela hash pode ser usada para armazenar os nomes de usuários e senhas. Quando um usuário faz login no banco de dados, o sistema usa a tabela hash para encontrar o nome de usuário e a senha correspondente.
- Em um compilador, uma tabela hash pode ser usada para armazenar as informações sobre os símbolos declarados no código-fonte. Quando o compilador precisa gerar código-objeto para uma variável, ele pode usar a tabela hash para encontrar o tipo de dados da variável.
- Em um sistema operacional, uma tabela hash pode ser usada para armazenar os processos em execução no sistema. Quando um processo precisa ser suspenso ou retomado, o sistema operacional pode usar a tabela hash para encontrar o processo rapidamente.
- Em uma rede, uma tabela hash pode ser usada para armazenar os endereços IP dos computadores na rede. Quando um computador precisa enviar um pacote de dados

para outro computador, ele pode usar a tabela hash para encontrar o endereço IP do destino.

- Tabelas hash são estruturas de dados poderosas que podem ser usadas em uma ampla variedade de aplicações. Elas são uma ferramenta essencial para qualquer desenvolvedor de software que precise armazenar e recuperar dados de forma eficiente.

3 Exemplos de uso

O uso de listas ou árvores opara organizar informações é interessante e produz bons resultados.

Porem, em nenhuma dessas estruturas se obtém o acesso direto a algumas informações, a partir do conhecimento de sua chave.

Hashing é uma maneira de organizar dados que:

- Apresenta bons resultados na prática.

- distribui os dados em posições aleatórias de uma tabela.

- Uma tabela hash é construída através de um vetor de tamanho n, no qual se armazenam as informações.

- Nele, a localização de cada informação é dada a partir do cálculo de um índice através de uma função de indexação, a função de hash.

Tabela 1 – Tabelas Hash

i	0	1	2	3	4	5	6	7	8	9
A [i]	-1	-1	-1	-1	34	45	-1	67	78	89

- A posição de um elemento é obtida aplicando-se ao elemento a função de hash que devolve a sua posição na tabela.
  - Daí basta verificar se o elemento realmente está nesta posição.

- O objetivo então é transformar a chave de busca em um índice na tabela.
  - Exemplo:
    - Construir uma tabela com os elementos 34, 45, 67, 78, 89.
    - Supõe-se uma tabela com 10 elementos e uma função de hash x%10 (resto da divisão por 10).

- -1 indica que não existe elemnto naquela posição.

```
 1 int hash(int x)
 2 {
 3  return x % 10;
 4 }
 5 void insere(int a[], int x)
 6 {
 7  a[hash(x)] = x;
 8 }
 9 int busca_hash(int a[], int x)
10 {
11  int k;
12  k = hash(x);
13  if (a[k] == x) return k;
14  return - 1;
15 }
```

Há muitas maneiras de determinar uma função de hash.

Divisão

- Uma função de hash precisa garantir que o valor retornado seja um índice válido para uma das células da tabela.

- A maneira mais simples é usar o módulo da divisão como h(k) = k%S, sendo K um número e S o tamanho da tabela.

- O método da divisão é bastante adequado quando se conhece pouco sobre as chaves.

Enlaçamento

- Neste método a chave é dividida em diversas partes que são combinadas ou "enlaçadas" e transformadas para criar o endereço.

- Existem 2 tipos de enlaçamento:
  - enlaçamento deslocado e
  - enlaçamento limite.

Enlaçamento deslocado

- As partes da chave são colocadas uma embaixo da outra e processadas.

- Por exemplo, um código 123-456-789 pode ser dividido em 3 partes: 123-456-789 que são adicionadas resultando em 1368.

- Esse valor pode usar o método da divisão valor%S, ou se a tabela contiver 1000 posições pode-se usar os 3 primeiros números para compor o endereço.

Elaçamento limite

- As partes da chave são colocadas em ordem inversa.

- Considerando as mesmas divisões do código 123-456-789.

- Alinha-se as partes sempre invertendo as divisões da seguinte forma 321-654-987

- O resultado da soma é 1566.

- Esse valor pode usar o método da divisão valor%S, ou se a tabela contiver 1000 posições pode-se usar os 3 primeiros números para compor o endereço.

Meio-quadrado

- A chave é elevado ao quadrado e a parte do resultado é usada como endereço.

Extração

- Neste método somente uma parte da chave é usada para criar o endereço.

- Para o código 123-456-789 pode-se usar os primeiros ou os últimos 4 dígitos ou outro tipo de combinação como 1289.

- Somente uma porção da chave é usada.

Transformação da raiz

- A chave é transformada para outra base numérica.

- O valor obtido é aplicado no método da divisão valor%S para obter o endereço.

Tabela 2 – Tabela hash

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	52	-1	-1	-1	-1	23	58	42	-1	-1	-1	12	-1	-1	-1

- Suponha agora os elementos 23, 42, 33, 52, 12, 58.

- Com a mesma função de hash, tem-se mais de um elemento para determinadas posições (42, 52 e 12; 23 e 33) para a função x%10.

- Pode-se usar a função x%17, com uma tabela de 17 posições.

- A função de hash pode ser escolhida à vontade de forma a atender da melhor forma a distribuição.

- A escolha da função é a parte mais importante

- É sempre melhor escolher uma função que use uma tabela com um número razoável de elementos.

- No exemplo se houvesse a informação adicional que todos os elementos estão entre 0 e 99, poderia se usar também uma tabela com 100 elementos onde a função de hash é o próprio elemento.
  - Mas seria uma tabela muito grande para uma quantidade pequena de elementos.

- A escolha da função é um compromisso entre a eficiência na busca o gasto de memória.

- A idéia central das técnicas de hash é sempre espalhar os elementos de forma que os mesmos sejam rapidamente encontrados.

3.1 Colisões - lista linear

Em tabelas hash, uma colisão ocorre quando duas ou mais chaves diferentes são mapeadas para a mesma posição da tabela. Isso pode acontecer devido à natureza probabilística das funções hash, que podem gerar valores repetidos mesmo para chaves distintas.

Uma das formas de resolver colisões em tabelas hash é usando a técnica de endereçamento aberto. Nessa técnica, cada posição da tabela é usada para armazenar uma lista encadeada de elementos com hash igual.

- A lista linear é uma implementação simples e eficiente de endereçamento aberto. Ela funciona da seguinte forma:

- Quando um elemento é inserido na tabela, sua chave é usada para calcular seu endereço na tabela.

- Se a posição da tabela estiver vazia, o elemento é inserido diretamente.

- Se a posição da tabela estiver ocupada, o elemento é adicionado à lista encadeada que está armazenada nessa posição.

A lista linear é uma solução eficiente para colisões em tabelas hash quando a taxa de colisões é baixa. No entanto, quando a taxa de colisões é alta, a lista linear pode se tornar ineficiente, pois a busca de um elemento pode exigir a varredura de toda a lista.

Aqui estão algumas vantagens e desvantagens da lista linear para resolver colisões em tabelas hash:

Vantagens:

- Simples de implementar
- Eficiente para taxas de colisões baixas

Desvantagens:

- Pode se tornar ineficiente para taxas de colisões altas
- Pode levar a desperdício de memória
- Para melhorar o desempenho da lista linear para taxas de colisões altas, pode-se usar técnicas como:
- Rehashing: Redimensionar a tabela periodicamente para reduzir a taxa de colisões.
- Colisão aberta: Usar uma função hash que gera endereços mais dispersos, o que reduz a probabilidade de colisões.

Tabela 3 – Considere os elementos do exemplo anterior e a função x%10.

i	0	1	2	3	4	5	6	7	8	9
A [i]	-1	-1	42	23	33	52	12	-1	58	-1

- Esta forma de tratamento de colisões tem uma desvantagem que é:
  - a tendência de formação de grupos de posições ocupadas consecutivas,
  - fazendo com que a primeira posição vazia, na prática, possa ficar muito longe da posição original, dada pela função de hash.

- Para inserir um determinado valor x na tabela, ou para concluir que o valor não se encontra na tabela, é necessário encontrar a primeira posição vazia após a posição h(x).

```
1 int hash(int x) { return x % 10;}
2 int insere(int a[], int x, int n) {
3  int i, cont = 0;
4  i = hash(x);
5  while (a[i] != -1) // procura a próxima posição livre
6  {if (a[i] == x) return -1; // valor já existente na tabela
7  if (++cont == n) return -2; // tabela cheia
8  if (++i == n) i = 0; // tabela circular
9  }
10  a[i] = x; // achou uma posição livre
11  return i;
12 }
13 int busca_hash(int a[], int x, int n) {
14  int i, cont = 0 ;
15  i = hash(x); // procura x a partir da posição i
16  while (a[i] != x)
17  {if (a[i] == -1) return -1; // não achou x
18  if (++cont == n) return -2; // a tabela está cheia
19  if (++i == n) i = 0; // tabela circular
20  }// encontrou
21  return i;
```

3.1.1 A função Hash e seus critérios de escolha

- A operação “resto da divisão por” (módulo – % em C) é a maneira mais direta de transformar valores em índices.

  Exemplos:

- Se o conjunto é de inteiros e a tabela é de M elementos, a função de hash pode ser simplesmente x%M.

- Se o conjunto é de valores fracionários entre 0 e 1 com 8 dígitos significativos, a função de hash pode ser floor(x*108)%M.

- Se são números entre s e t, a função pode ser floor ((x-s)/(t-s)*M)

  A escolha é bem livre, mas o objetivo é sempre espalhar ao máximo dentro da tabela os valores da função para eliminar as colisões.

3.1.2 A função de Hash

- A função de hash deve ser escolhida de forma a atender melhor a particularidade da tabela com a qual se trabalha.

- Os elementos procurados, não precisam ser somente números para se usar hashing.

- Uma chave com caracteres pode ser transformada num valor numérico.

3.1.3 Colisões - Duplo Hashing ou Rehash

- Se a tabela está muito cheia a busca sequencial pode levar a um número muito grande de comparações.

- No pior caso (N elementos ocupados), deve-se percorrer os N elementos antes de encontrar o elemento ou concluir que ele não está na tabela.

- A grande desvantagem da Lista Linear é o aparecimento de agrupamentos.

- Uma forma de permitir um espalhamento maior é fazer com que o deslocamento em vez de 1 seja dado por uma segunda função de hash.

- Essa segunda função de hash tem que ser escolhida com cuidado, não deve gerar um valor nulo (loop infinito).

- Deve ser tal que a soma do índice atual com o deslocamento (módulo N) dê sempre um número diferente até que os N números sejam verificados.

- Para isso N e o valor desta função devem ser primos entre si.

- Uma maneira é escolher N primo e garantir que a segunda função de hash tenha um valor K menor que N. Dessa forma N e K são primos entre si.

- Existem duas funções de Hash: Uma para usar normalmente e outra para usar quando há colisões
  - h1(x)=(x % N) = C1
  - h2(x) = (x % N-1)+1 = C2 $\rightarrow$ usada quando há colisões
  - Para calcular o primeiro índice usa-se C1
  - Para calcular o segundo índice (se existir colisões) usa-se (C1+C2) %N
  - Para calcular o terceiro índice (se também houver colisões) usa-se (C1+2C2) %N, depois (C1+3C2) %N, etc.

Tabela 4 – Exemplo: Inserir os elementos 5, 10, 12, e 19 na tabela com N = 7

0	1	2	3	4	5	6
19			10		5	12

- h1(x) = x % 7 = C1
- h2(x) = (x % 6) +1 = C2
  - 5 $\rightarrow$ h1 (5) = 5 $\rightarrow$ 5 está vazio;
  - 10 $\rightarrow$ h1 (10) = 3 $\rightarrow$ 3 está vazio;
  - 12 $\rightarrow$ h1 (12) = 5 $\rightarrow$ 5 está ocupado, faz-se h2;
    12 $\rightarrow$ h2 (12) = 1 $\rightarrow$ (C1+C2)%7 $\rightarrow$ 6 está vazio;
  - 19 $\rightarrow$ h1 (19) = 5 $\rightarrow$ 5 está ocupado, faz-se h2;
    19 $\rightarrow$ h2 (19) = 2 $\rightarrow$ (C1+C2)%7 $\rightarrow$ 0 $\rightarrow$ como o rehash é circular vai-se inserir no 0;

- Como a seleção da segunda função de hash é livre pode-se escolher um valor fixo.

- Exemplo : Inserir os elementos 25, 37, 48, 59, 32, 44, 70, 81 (nesta ordem) com N=11.

- h1(x) = x % 11 = C1

- h2(x) = 3 = C2
  - 25 → h1 (25) = 3 → vazio;
  - 37 → h1 (37) = 4 → vazio;
  - 48 → h1 (48) = 4 → ocupado, h2 = 3 (4+3)=7 → vazio;
  - 59 → h1 (59) = 4 → ocupado, h2 = 3 (4+3)=7 → ocupado (C1+2C2)%N = 10 → vazio;

```
1 int hash(item x, int N) {
2  return ...; // o valor da função
3 }
4 int hash2(item x, int N) {
5  return ...; // o valor da função
6 }
7 int insere(item a[], item x, int N) {
8  int i = hash(x);
9  int k = hash2(x);
10  int cont = 0;
11  // procura a próxima posição livre
12  while (a[i] != -1) {
13  if (a[i] == x) return -1; // valor já existente na
       tabela
14  if (++cont == N) return -2; // tabela cheia
15  i = (i + k) % N; // tabela circular
16  }
17  // achou uma posição livre
18  a[i] = x;
19  return i;
20 }
```

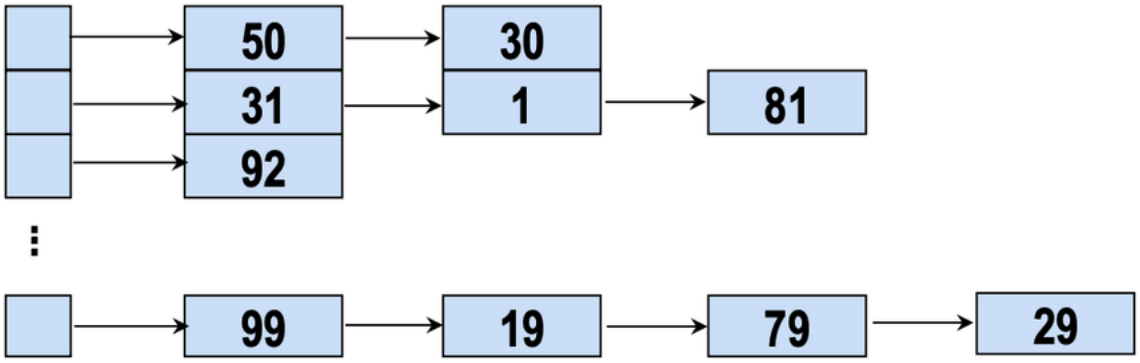
```
1 int busca_hash(item a[], item x, int N) { int i = hash(x); int k
     = hash2(x) int cont = 0 ;
2  // procura x a partir da posição i while (a[i] != x) { if (a[i]
      == -1) return -1; // não achou x, pois há uma vazia
3  if (++cont == N) return -2; // a tabela está cheia
4  i = (i + k) % N; // tabela circular
5  }
6  // encontrou
7  return i;
8 }
```

3.1.4 Colisões - Hash com lista encadeada

- Podemos criar uma lista ligada com os elementos que tem a mesma chave em vez de deixá-los todos na mesma tabela.
  - Com isso pode-se até diminuir o número de chaves geradas pela função de hash.
  - Tem-se, dessa forma, um vetor de ponteiros.

- Considere uma tabela de inteiros e como função de hash x%10.

Figura 1 – Hash Lista Encadeada



- A função pode ser melhorada:
  - Só inserir se já não estiver na tabela. Para isso deve-se percorrer a lista até o final;
  - Inserir elemento de modo que a lista fique em ordem crescente.

3.1.5 Prós e Contras

Cada um dos métodos apresentados tem seus prós e contras:

- Lista linear: é o mais rápido se o tamanho de memória permite que a tabela seja bem esparsa.

- Duplo hash: usa melhor a memória mas depende também de um tamanho de memória que permita que a tabela continue bem esparsa.

- Lista ligada: é interessante, mas precisa de um alocador rápido de memória.

A escolha de um ou outro depende da análise particular do caso.

- A probabilidade de colisão pode ser reduzida usando uma tabela suficientemente grande em relação ao número total de posições a serem ocupadas
  - Por exemplo, uma tabela com 1000 entradas para uma empresa que deseja armazenar 500 posições haveria uma probabilidade de 50% de colisão, se fosse feita a inserção de uma nova chave.

- Considera-se uma tabela hash bem dimensionada,
  - Média 1,5 acessos à tabela para encontrar um elemento.

3.1.6 Hash perfeita

- O ideal para a função hash é que sejam sempre fornecidos índices únicos para as chaves de entrada.

- A função perfeita (hash perfeita) seria:
  - para quaisquer entradas A e B, sendo A diferente de B, fornecesse saídas diferentes.

- A tabela deve conter o mesmo número de elementos.
  - Nem sempre o número de elementos é conhecido a priori.

- Na prática, funções hash perfeitas ou quase perfeitas são encontradas apenas onde a colisão é intolerável
  - por exemplo, nas funções hash da criptografia, ou quando se conhece previamente o conteúdo da tabela armazenada.

3.1.7 Remoção

Quando se remove um elemento, a tabela perde sua estrutura de hash.

Suponha que a tabela hash a seguir trata colisões por Lista Linear e o elemento x é removido.

Com a remoção de x apenas u e v continuariam acessíveis: o acesso a w, y e z seria perdido.

Para remover x, de forma correta, seria necessário mudar diversos outros elementos de posição na tabela.

422	423	424	425	426	427	428
	u	v	x	w	y	z

Com a remoção de x apenas u e v continuariam acessíveis: o acesso a w, y e z seria perdido.

Para remover x, de forma correta, seria necessário mudar diversos outros elementos de posição na tabela.

- Uma técnica simples é a de marcar a posição do elemento removido como apagada (mas não livre).
  - Isso evita a necessidade de movimentar elementos na tabela mas cria muito lixo.

- Uma melhoria nessa técnica é reaproveitar as posições marcadas como removidas no caso de novas inserções.
  - É eficiente se a frequência de inserções for equivalente à de remoções.

- Em casos extremos é necessário refazer o hashing completo dos elementos.

3.1.8 Tabelas Hash

Embora permita o acesso direto ao conteúdo das informações, o mecanismo das tabelas hash possui uma desvantagem em relação a listas e árvores.

- Numa tabela hash é virtualmente impossível estabelecer uma ordem para os elementos.
    - A função de hash faz indexação, mas não preserva ordem.
- Avaliar uma boa função hash é um trabalho difícil e relacionado à estatística.
- Dependendo da aplicação outras estruturas devem ser levadas em conta.

4 Conclusão

Tabelas hash são estruturas de dados poderosas e versáteis que podem ser usadas em uma ampla variedade de aplicações. Elas oferecem uma série de benefícios, incluindo:

- Eficiência: Tabelas hash permitem buscas rápidas, independentemente do número de elementos na tabela.
- Flexibilidade: Tabelas hash podem ser usadas para armazenar uma ampla variedade de dados.
- Escalabilidade: Tabelas hash podem ser facilmente escalonadas para lidar com grandes quantidades de dados.
- No entanto, tabelas hash também têm algumas limitações, incluindo:
- Colisões: Colisões podem ocorrer, o que pode levar a um aumento no tempo de busca.
- Consumo de memória: Tabelas hash podem consumir mais memória do que outras estruturas de dados.

Os benefícios de usar tabelas hash superam suas limitações em muitas aplicações. No entanto, é importante estar ciente das limitações ao escolher usar tabelas hash.

Recomendações

Para aproveitar ao máximo os benefícios de tabelas hash, é importante escolher uma função hash adequada. Uma função hash adequada deve distribuir as chaves de forma uniforme pelo vetor da tabela hash. Isso ajudará a reduzir o número de colisões e, consequentemente, melhorar o desempenho das buscas.

Além disso, é importante usar um tamanho de tabela hash adequado. Uma tabela hash muito pequena pode levar a um aumento no número de colisões, enquanto uma tabela hash muito grande pode desperdiçar memória.