Tabelas Hash
Hashing

Prof. Tiago Massoni
Baseado em material do Cin-UFPE

Engenharia da Computação

Poli – UPE

Mapeamento

- Associação de cada objeto de um tipo a uma chave, permitindo a indexação
  - Dicionário
- Ex: Inteiro → String

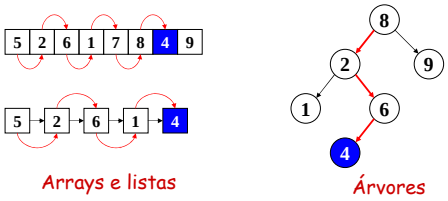
- O número correspondente será usado para uma busca direta (de tempo constante)
- Utopia: endereçamento direto (array com posição=chave)

Tabelas Hash

- Até agora fizemos buscas por uma chave diretamente através de comparações
- Tabelas hash permitem procuras com muitas vezes apenas uma (1) operação de leitura
- Idéia: gerar, a partir de uma chave, o endereço (posição) onde o elemento desta chave está
  - Hashing

Por que usar Hashing?

- Estruturas de busca sequencial levam tempo até encontrar o elemento desejado



Arrays e listas | Árvores

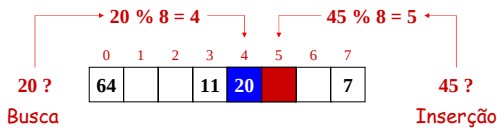
Hashing

- Saber a posição em que o elemento se encontra, sem precisar varrer todas as chaves
- Se o número de chaves possíveis for muito grande, distribuir as chaves entre as posições disponíveis (de 0 a tamanho-1)
- Função de mapeamento chave-posição é a função hash
  - Posição de k = h(k)

Tabela hash

- Ex: uma função hash clássica é tirar o resto da divisão pelo tamanho da tabela

```
h(chave) = chave % array.length
```



Colisões

- Devem existir mais chaves que posições
  - Duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela
  - Quando isto ocorre, dizemos que houve uma colisão
- O que fazer quando há colisão?
  - Tabela hash ideal seria aquela sem colisão!

Ex: 45 % 8 = 5 1256 % 15 = 11
21 % 8 = 5 356 % 15 = 11
93 % 8 = 5 506 % 15 = 11

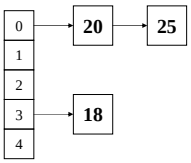
Endereçamento fechado

- Posição de inserção não muda para resultados de hash iguais
- Todos devem ser inseridos na mesma posição, através de uma lista ligada em cada uma
  - *Chaining*

20 % 5 = 0

18 % 5 = 3

25 % 5 = 0
colisão com 20



Endereçamento fechado

- A tabela hash, neste caso, contém um array de listas

```
class TabelaHash {
  private Lista[] listas;

  public TabelaHash(int n) {
    listas = new Lista[n];
    for (int i = 0; i < n; i++)
      listas[i] = new Lista();
  }
}
```


Função hash em Java

- A classe Object oferece o método
  - public int hashCode()
- Retorna o hash code (posição) para o objeto (chave) no qual o método foi chamado
- Usado em coleções baseadas em hashing
  - Hashtable, HashMap, HashSet

Definição do método hashCode em Java

- Quando invocado no mesmo objeto duas vezes, deve retornar o mesmo resultado
- Se dois objetos desta classe forem iguais de acordo com o método equals, eles devem produzir o mesmo resultado em hashCode

```
class Account {
  private int number; ..

  public int hashCode() {
    return number % TABLESIZE;
  }
}
```

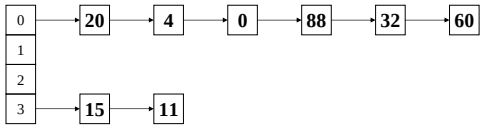

Endereçamento fechado

- Quando uma chave for inserida, a função hash é aplicada, e ela é acrescentada à lista adequada
- Neste exemplo, só vamos se não encontrarmos o elemento

```
void inserir(Object chave) {
  int i = chave.hashCode();
  ListIterator it= listas[i].find(chave);
  if (it.isPastEnd())
    listas[i].inserir(chave,it.zeroth());
}
```

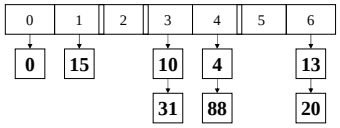
Endereçamento fechado

- A busca é feita do mesmo modo: calcula-se o valor da função hash para a chave, e a busca é feita na lista correspondente.
- Se o tamanho das listas variar muito, a busca pode se tornar ineficiente, pois a busca nas listas é seqüencial:



Endereçamento fechado

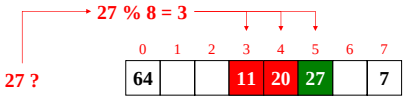
- Por esta razão, a função hash deve distribuir as chaves entre as posições uniformemente:



- Se o tamanho da tabela for um número primo, há mais chances de ter uma melhor distribuição.

Endereçamento aberto

- Quando uma nova chave é mapeada para uma posição já ocupada, uma nova posição é indicada para esta chave
- Idéia é chegar mais próximo a uma hash perfeita
- Com linear probing, a nova posição é incrementada até que uma posição vazia seja encontrada:



Endereçamento aberto

A tabela *hash*, neste caso, contém um array de objetos, e posições vazias são indicadas por **null**

Exemplo: objetos serão do tipo **Integer**

```
class TabelaHash {
  Integer[] posicoes;

  public TabelaHash(int n) {
    posicoes = new Integer[n];
  }
}
```


Linear Probing

Na inserção, a função hash é calculada, e a posição incrementada, até que uma posição esteja livre:

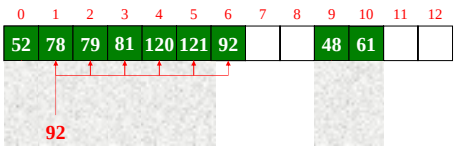
```
void inserir(Integer chave) {

  int i= chave.hashCode();
  while (posicoes[i] != null)
    i= (i + 1) % posicoes.length;

  posicoes[i] = chave;
}
```


Linear Probing

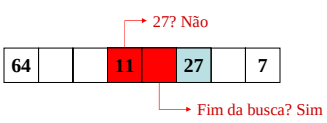
Valores: 52, 78, 48, 61, 81, 120, 79, 121, 92
Função: hash(k) = k % 13
Tamanho da tabela: 13



Remoção

•Para fazer uma busca com endereçamento aberto, basta aplicar a função hash e a função de incremento até que o elemento ou uma posição vazia sejam encontrados

•Porém, quando um elemento é removido, a posição vazia pode ser encontrada antes, mesmo que o elemento pertença a tabela:



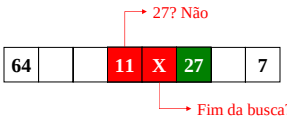
Inserção do 27

Remoção do 20

Busca pelo 27

Remoção

Para contornar esta situação, mantemos um bit (ou um campo booleano) para indicar que um elemento foi removido daquela posição:



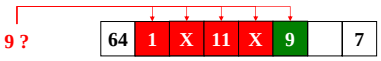
Esta posição estaria livre para uma nova inserção, mas não seria tratada como vazia numa busca.

Rehashing

• Fator de carga (load factor)

•Indica a porcentagem de células da tabela hash que estão ocupadas, incluindo as que foram removidas.

• Quando este fator fica muito alto (excede 50%), as operações na tabela passam a demorar mais, pois o número de colisões aumenta

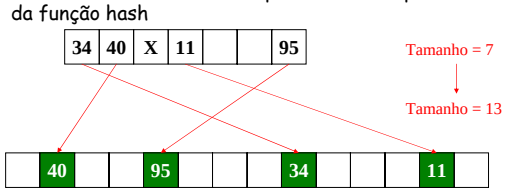


Rehashing

• Expandir o array que constitui a tabela, e reorganizar os elementos na nova tabela

• Novo tamanho: número primo próximo ao dobro da tabela atual

•O tamanho atual da tabela passa a ser um parâmetro da função hash



Rehashing

O momento de expandir a tabela pode variar:

• Quando não for possível inserir um elemento

• Quando metade da tabela estiver ocupada

• Quando o *load factor* atingir um valor escolhido

A terceira opção é a mais comum, pois é um meio termo entre as outras duas.

Quando não usar Hashing?

•Muitas colisões diminuem muito o tempo de acesso e modificação de uma tabela hash. Para isso é necessário escolher bem

- a função hash
- o tratamento de colisões
- o tamanho da tabela

•Quando não for possível definir parâmetros eficientes, pode ser melhor utilizar árvores balanceadas (como AVL), em vez de tabelas hash