ESTRUTURAS DE DADOS

2024/2025

Aula 13

- *Hashing*
- Funções de *hash*
- Tratamento de Colisões
- Tabelas de *hash* na plataforma de colecções do *Java*

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Ricardo Santos | rjs@estg.ipp.pt

Hashing

- Nas colecções que temos discutido até agora, fizemos uma das três hipóteses sobre ordem:
- A ordem não é importante
- A ordem é determinada pela forma como os elementos são adicionados
- A ordem é determinada pela comparação dos valores dos elementos

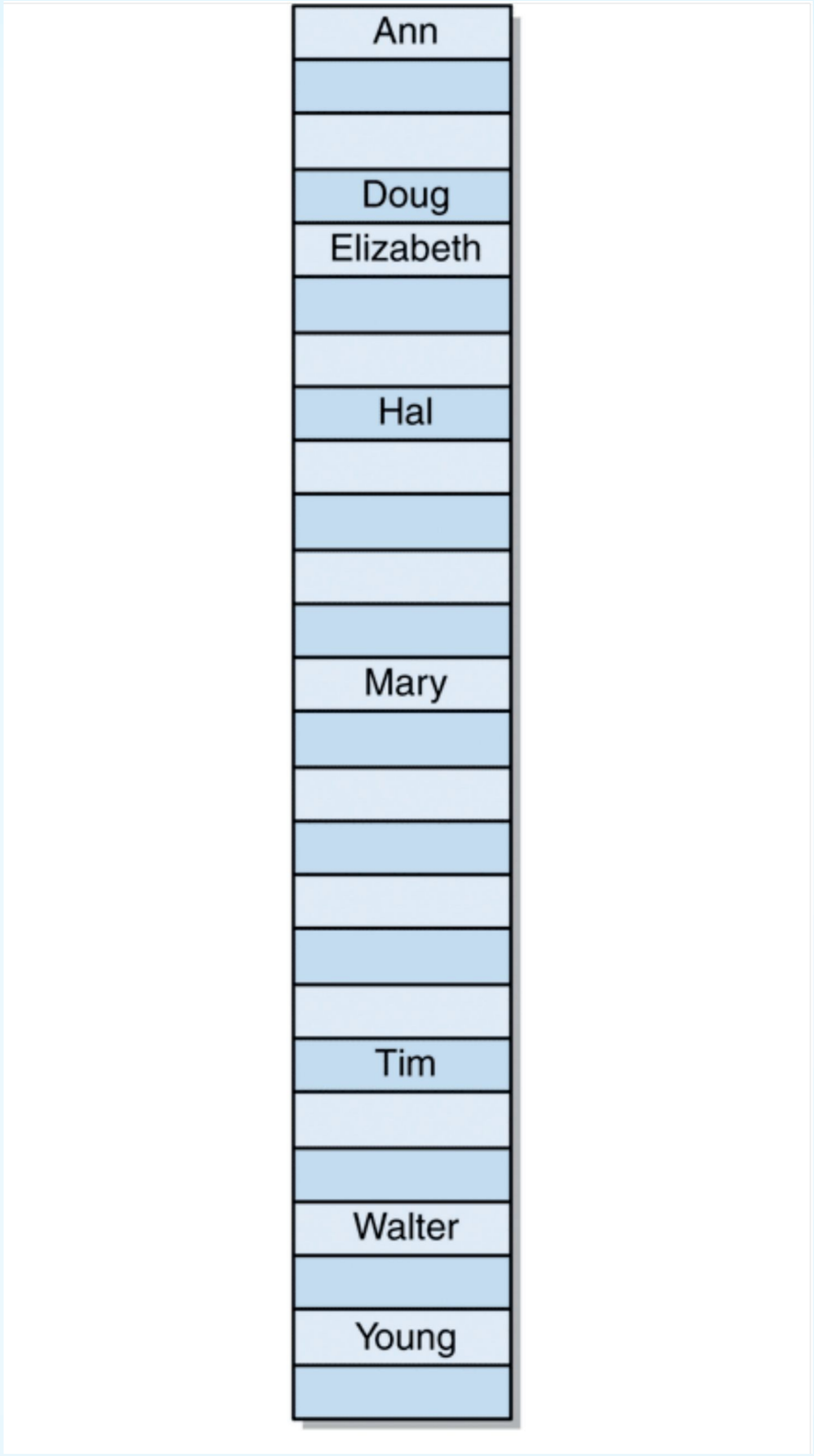
- *Hashing* é o conceito em que a ordem é determinada por alguma função relativa ao valor do elemento a ser armazenado
  - Mais precisamente, o local dentro da colecção é determinado por alguma função do valor do elemento a ser armazenado

- Em *hashing*, os elementos são armazenados numa tabela de *hash* (tabela de dispersão) com a sua localização na tabela determinada por uma função de *hash*

- Cada posição na tabela de *hash* é chamada de célula ou balde (*bucket*)

- Considere um exemplo onde criamos um *array* que irá armazenar 26 elementos

- Pretendemos armazenar nomes no nosso *array*, para isso criamos uma simples função de *hash* que usa a primeira letra de cada nome


Ann
Doug
Elizabeth
Hal
Mary
Tim
Walter
Young

- Observe que ao usar uma abordagem de *hashing* irá resultar num tempo de acesso a um elemento em particular independente do número de elementos na tabela

- Isso significa que todas as operações num elemento de uma tabela hash devem ser **O(1)**

- No entanto, essa eficiência é conseguida somente se cada elemento for mapeado para uma posição única na tabela

- Uma colisão ocorre quando dois ou mais elementos forem mapeados para o mesmo local
- Uma função *hash* que mapeia cada elemento para uma localização única, é denominada de função *hash* perfeita
- Embora seja possível desenvolver uma função de *hashing* perfeita, uma função que faz um bom trabalho de distribuição dos elementos na tabela ainda vai resultar em **O(1)** operações

- Outra questão em torno do *hashing* é o tamanho que a tabela deve ter
- Se temos a certeza de um conjunto de tamanho **n** e uma função de hashing perfeita, então a tabela deve ser de tamanho **n**
- Se uma função *hash* perfeita não está disponível, mas sabemos o tamanho do conjunto de dados (**n**), então uma boa regra geral é fazer com que a tabela seja **150%** do tamanho do conjunto de dados

- O terceiro caso é o caso em que não sabemos o tamanho do conjunto de dados
- Neste caso, irá depender do redimensionamento dinâmico
- O redimensionamento dinâmico de uma tabela *hash* envolve a criação de uma nova tabela de *hash* maior e, de seguida inserir todos os elementos da tabela antiga na nova

- Decidir quando redimensionar é a chave para o redimensionamento dinâmico
- Uma possibilidade é simplesmente redimensionar quando a tabela está cheia
- No entanto, é da natureza das tabelas *hash* que o seu desempenho vai piorando à medida que a tabela começa a ficar cheia
- Uma abordagem melhor é usar um **factor de carga**
- O **factor de carga** de uma tabela de hash é a percentagem de ocupação da tabela em que a tabela será redimensionada

Funções de *hash*: extracção

- Existe uma grande variedade de funções de *hash* que proporcionam uma boa distribuição dos vários tipos de dados
- O método que usámos no exemplo anterior é chamado de extracção
- A extracção envolve o uso de apenas uma parte do valor de um elemento ou uma chave para calcular o local onde armazenar o elemento
- No exemplo anterior, foi simplesmente extraída a primeira letra de uma *string* e calculado o seu valor em relação à letra **A**

Funções de *hash*: divisão

- Uma função de *hash* por divisão significa simplesmente que vamos usar o resto da divisão inteira da chave (*key*) dividida por um qualquer valor inteiro positivo

```
HashCode(key) = Math.abs(key)%p
```

- Esta função irá gerar um resultado no intervalo de **0** a **p-1**
- Se usarmos o tamanho da tabela de **p**, então temos um índice que mapeia directamente para a tabela

Funções de *hash*: desdobramento ou *folding*

- A chave do método de desdobramento é dividir a chave em duas partes que são depois combinadas e agregadas de forma a criar um índice numa tabela
- Isso é feito dividindo-se a primeira chave em partes onde cada uma das partes da chave será do mesmo comprimento da chave desejado
- Existem dois tipos de desdobramento (*folding*)
  - ***shift folding***
  - ***limit folding***

- No método ***shift folding***, as partes são somadas para criar o índice
- Usando o `SSN 987-65-4321` dividimos-o em três partes, `987`, `654`, `321`, e de seguida, somamos todas essas partes para obter `1962`
- Teríamos, agora de usar por exemplo o método da divisão ou da extracção para obter um índice de três dígitos

- No método ***limit folding***, algumas partes da chave são invertidas antes de se adicionar
- Usando o mesmo exemplo de um `SSN 987-65-4321`
- dividir em partes o `SSN 987, 654, 321`
- inverter o elemento do meio `987, 456, 321`
- somar tudo, o que irá dar `1764`
- de seguida, usar o método da divisão ou da extracção para obter um índice de três dígitos
- O *folding* também pode ser usado em strings

Funções de *hash*: Meio do Quadrado

- No método do meio do quadrado, a chave é multiplicada por si mesmo e, de seguida, o método de extracção é usado para extrair o número necessário de dígitos a partir do meio do resultado
- Por exemplo, se a nossa chave é `4321`
- Multiplicar a chave por si mesmo -> `18671041`
- Extrair os três dígitos necessários

- É fundamental que sejam sempre os mesmos três dígitos por extracção de cada vez

- Podemos também extrair *bits* e de seguida reconstruir um índice a partir de *bits*

Funções de *hash*: Análise de Dígitos

- No método de análise de dígitos, o índice é formado pela extracção, e de seguida pela manipulação de dígitos específicos da chave

- Por exemplo, se a nossa chave é `1234567`, podemos seleccionar os dígitos nas posições de `2` a `4` obtendo `234`

- A manipulação pode assumir muitas formas
  - Inverter os dígitos (`432`)
  - Executar um *shift* circular à direita (`423`)
  - Executar um *shift* circular à esquerda (`342`)
  - trocar cada par de dígitos (`324`)

Funções de *hash* na linguagem *Java*

- A classe `java.lang.Object` define um método denominado hashcode que retorna um inteiro com base na localização de memória do objecto
  - Geralmente não é muito útil
- As classes que são derivadas de `Object`, muitas vezes redefinem a definição herdada de hashcode para fornecer sua própria versão
- Por exemplo, as classes `String` e `Integer` definem seus próprios métodos hashcode
- As funções `hashcode` mais específicas são mais eficazes

Hashing: Tratamento de colisões

- Se formos capazes de desenvolver uma função de *hashing* perfeito, não precisamos de nos preocupar com as colisões ou com o tamanho da tabela

- No entanto, muitas vezes não sabemos o tamanho do conjunto de dados e não somos capazes de desenvolver uma função de *hashing* perfeito

- Nestes casos, devemos escolher um método para o tratamento de colisões

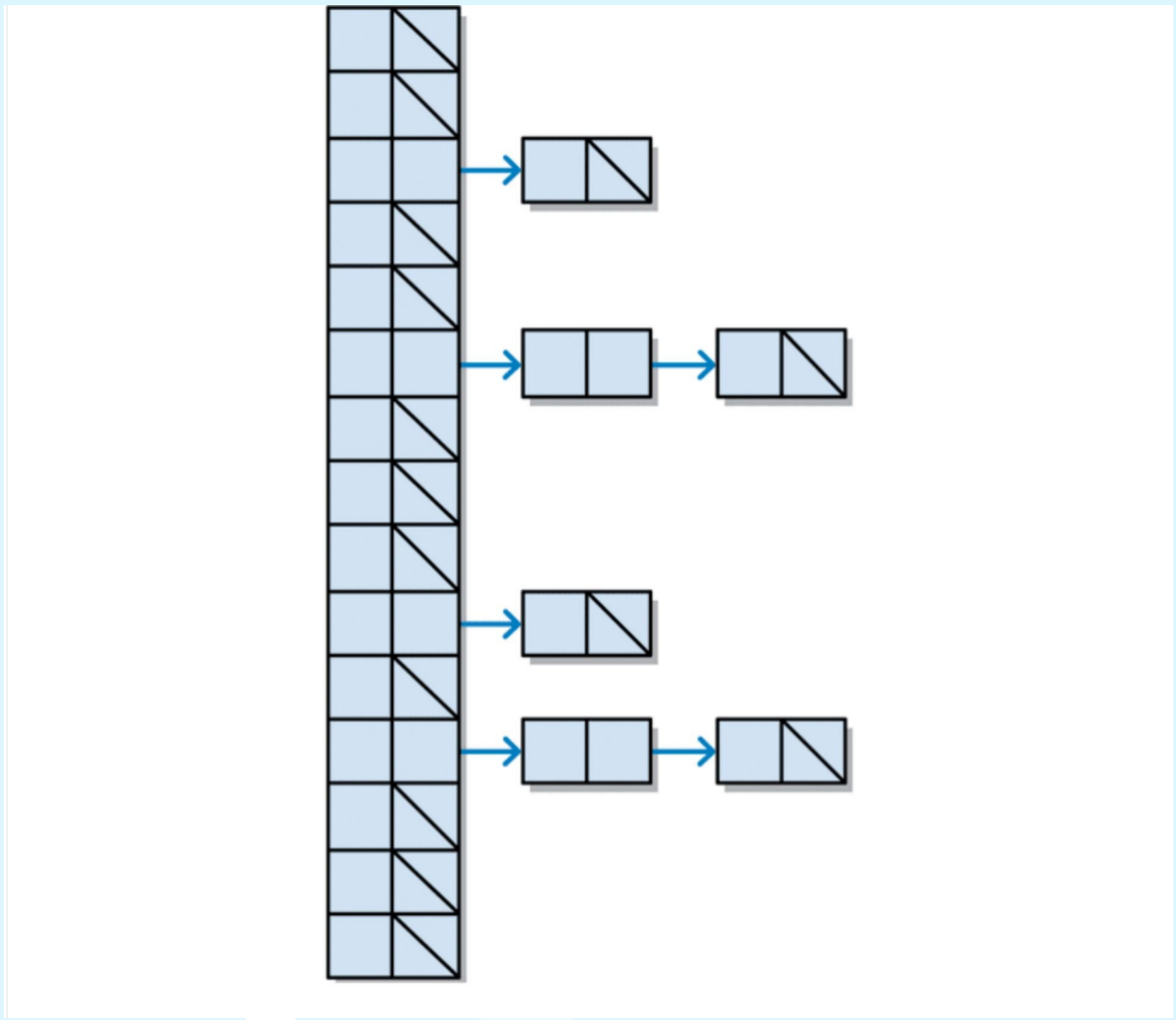
Tratamento de colisões: Encadeamento

- O método de encadeamento, trata conceptualmente a tabela de *hash* como uma tabela de colecções, em vez de uma tabela de elementos individuais
- Desta forma, cada célula é uma referência para a colecção associada com essa referência na tabela

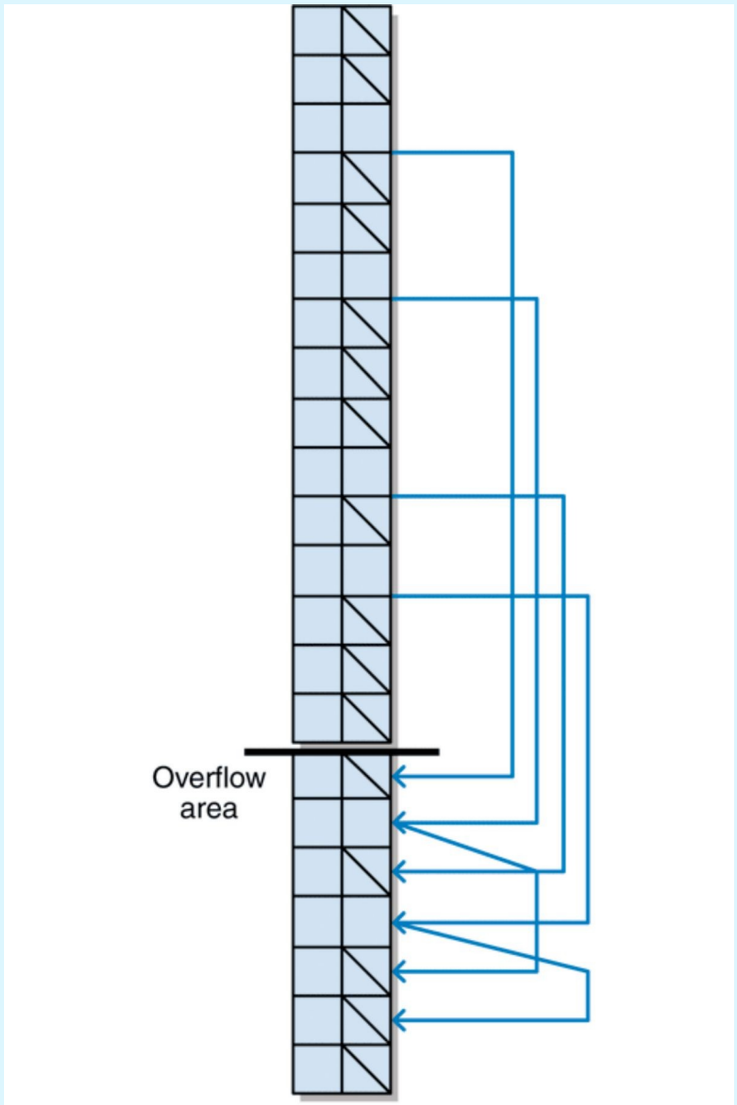
- Normalmente, esta colecção interna ou é uma lista não ordenada, ou uma lista ordenada

- O encadeamento pode ser realizado através de uma estrutura ligada ou de uma estrutura baseada em *array* com uma área de *overflow*

Tratamento de Colisões do Método de Encadeamento



Encadeamento com recurso a uma área de *overflow*



Tratamento de colisões: Endereçamento Aberto

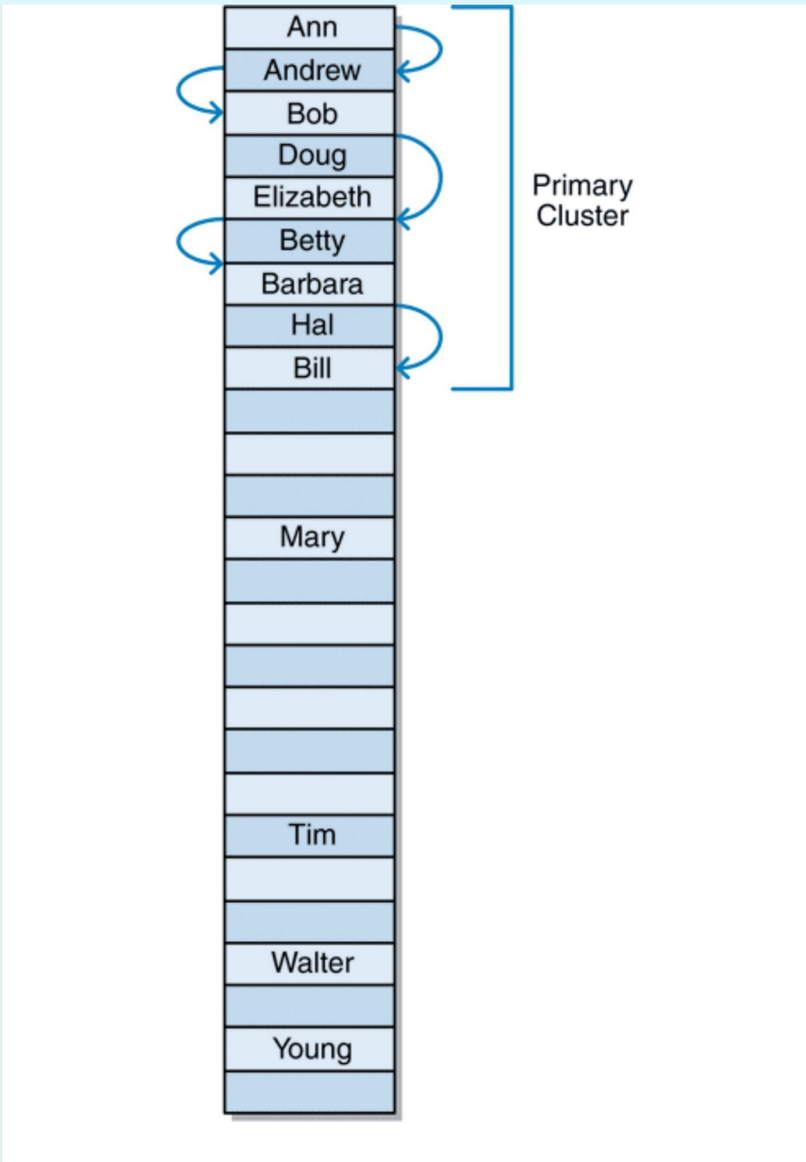
- O método de endereçamento aberto procura por outra posição em aberto na tabela diferente daquele ao qual o elemento é originalmente atribuído (através da função de hash)

- O mais simples desses métodos é a tentativa linear

- Na tentativa linear, se um elemento é calculado para a posição **p** e essa posição é ocupada simplesmente tentamos a posição **(p +1)% s**, onde **s** é o tamanho da tabela

- Um problema conhecido da tentativa linear é o desenvolvimento de aglomerados de células ocupadas chamados *clusters* primários

Endereçamento Aberto com Tentativa Linear

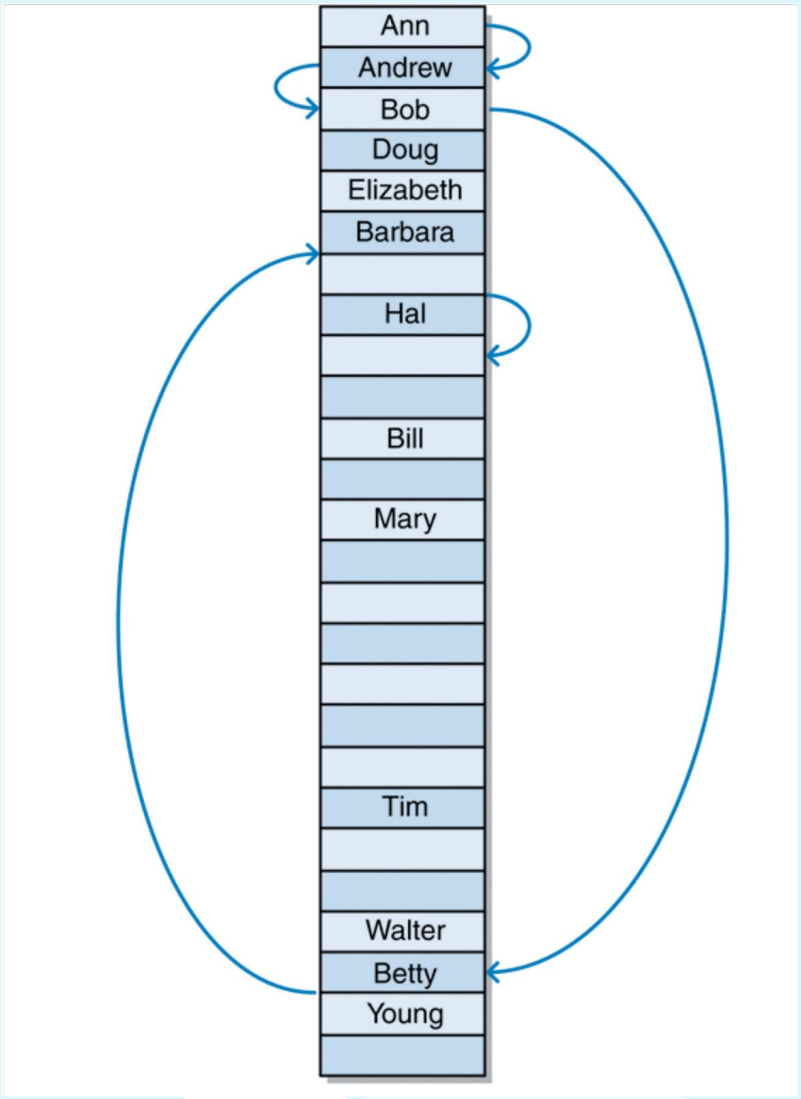


- Uma segunda forma de endereçamento aberto é a tentativa quadrática
- Neste método, uma vez que temos uma colisão, o próximo índice a testar é calculado pela função

$$newhashcode(x) = hashcode(x) + (-1)^{i-1}((i+1)/2)^2$$

- A tentativa quadrática ajuda a eliminar o problema de *clusters* primários

Endereçamento Aberto com Tentativa Quadrática

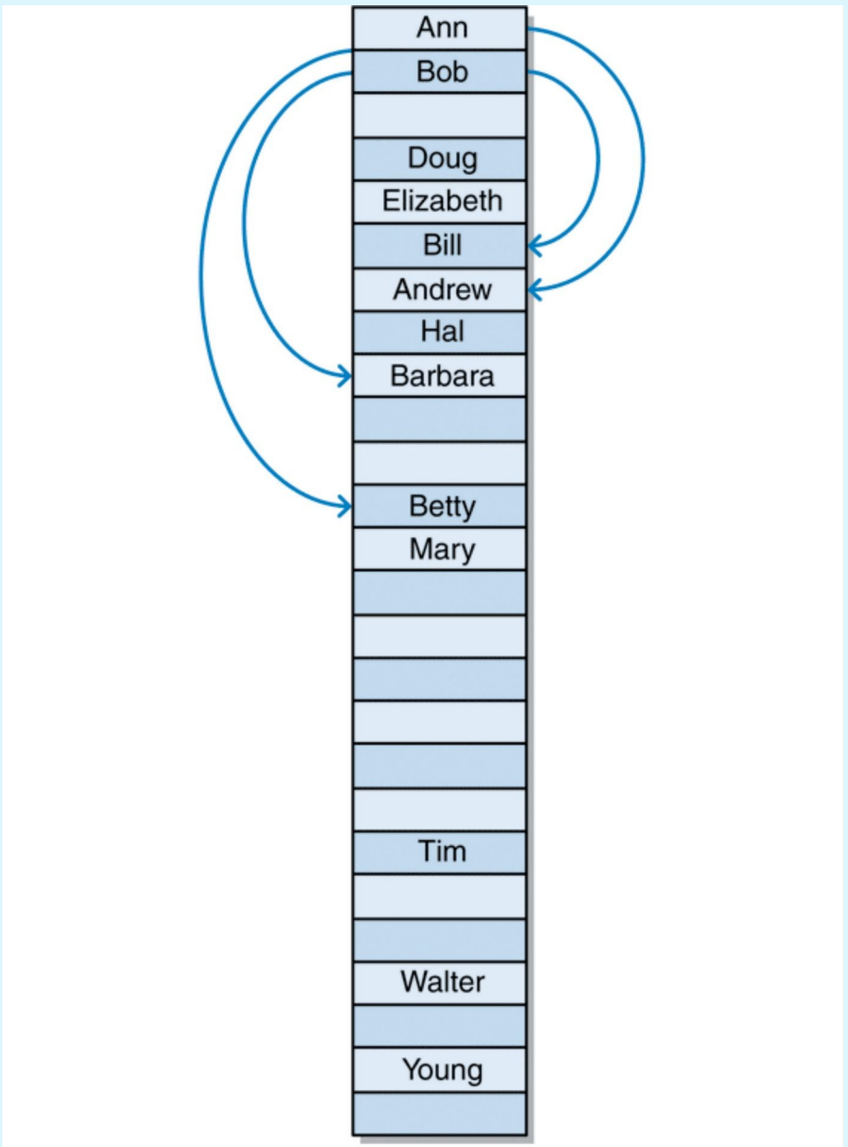


- A terceira forma de endereçamento aberto é a dispersão dupla
- Neste método, as colisões são resolvidas através do fornecimento de uma segunda função de *hash*
- Por exemplo, se uma chave **x** é calculada para uma posição **p** já ocupada, então a próxima posição **p'** será

```
p '= p + secondaryhashcode (x)
```

- Dado o mesmo exemplo, se usarmos uma função secundária de *hash* que usa o comprimento da *string*, obtemos o seguinte resultado

Endereçamento Aberto com Dispersão Dupla



Remover elementos da Tabela de *hash*

- Existem cinco casos para a remoção de um elemento de uma implementação encadeada
- Se o elemento que pretendemos remover é o único mapeado para a posição, temos de simplesmente remover o elemento, definindo a posição da tabela como *null*

- Se o elemento que estamos a tentar remover é armazenado na tabela, mas tem um índice para a área de *overflow*
  - Substituir o elemento e o próximo valor do *index* na tabela pelo elemento e pelo próximo valor do *index* da posição do *array* apontada pelo elemento a ser removido
  - De seguida devemos definir a posição na área de *overflow* como *null* e adicioná-la de volta à lista de células disponíveis de *overflow*

- Se o elemento que estamos a tentar remover estiver no final da lista de elementos armazenados nessa localização da tabela
  - Definir a sua posição na tabela como *null*
  - Definir a referência do próximo elemento do elemento anterior a *null*
  - Adicionar essa posição para a lista de células *overflow* disponível

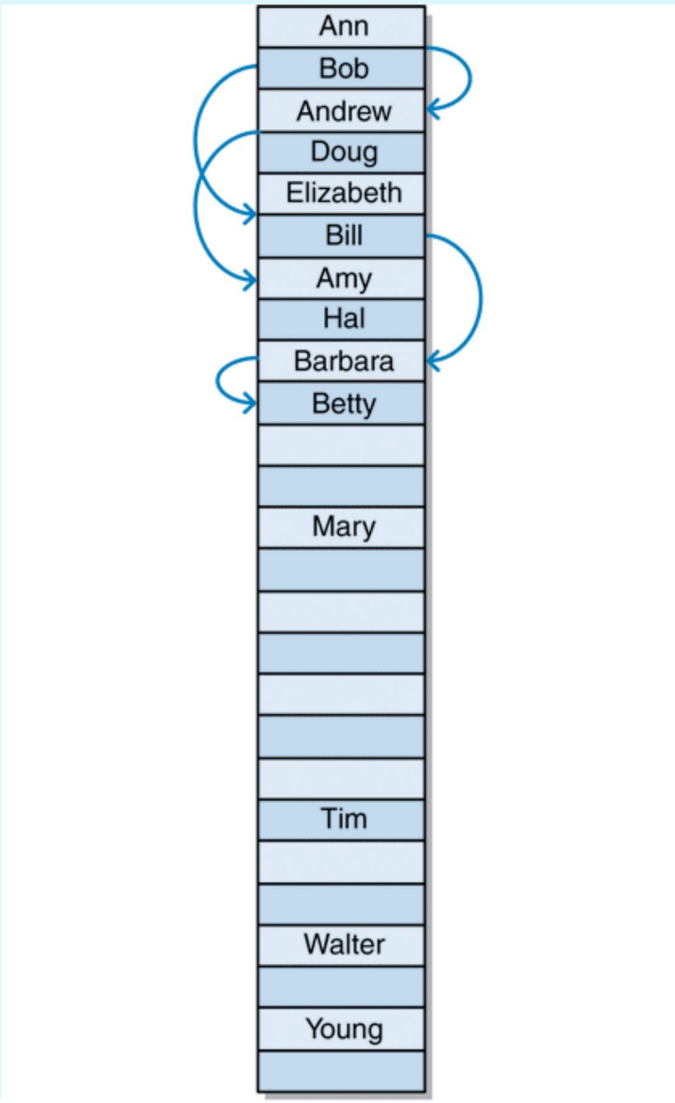
- Se o elemento que estamos a tentar remover estiver no meio da lista de elementos armazenados nessa localização
  - Definir a sua posição na área de *overflow* como *null*
  - Definir a referência do próximo elemento do elemento anterior para apontar para o próximo elemento do elemento a ser removido
  - Adicionar a posição de volta à lista de células de *overflow* disponíveis
- Se o elemento que estamos a tentar remover não estiver na tabela, então temos que lançar uma excepção

- Se implementarmos a tabela com endereçamento aberto a remoção de um elemento torna-se num desafio maior

- Torna-se um desafio porque não podemos remover um item excluído com a possibilidade de afectar a nossa capacidade de encontrar itens que colidiram com esta entrada

- A solução para esse problema é marcar os itens como excluídos, mas não os remover da tabela até que o item seja reescrito nessa posição ou a tabela seja recriada

Remoção em Endereçamento Aberto



Tabelas de *hash* na plataforma de colecções do *Java*

- A plataforma de colecções do Java fornece sete implementações de hash:
  - `Hashtable`
  - `HashMap`
  - `HashSet`
  - `IdentifyHashMap`
  - `LinkedHashSet`
  - `LinkedHashMap`
  - `WeakHashMap`