Estruturas de Dados e Algoritmos

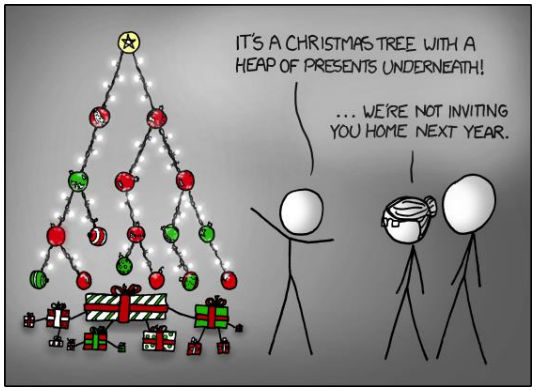
Tabela Hash

Adalberto Cajueiro

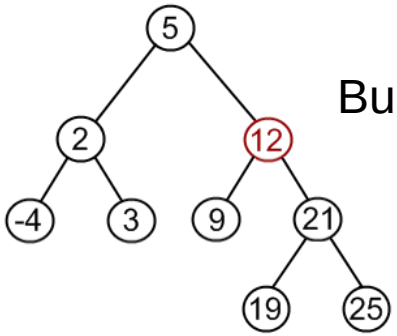
Departamento de Sistemas e Computação

Universidade Federal de Campina Grande

Estruturas vistas até agora



Busca O(n)



Busca O(log(n))

É POSSÍVEL?

- Add, search, remove em O(1)?

Relembrando: tabela, mapa, dicionario

- Conceito <chave,valor>
- Operações

  put(K key, V value) -> T[k] = v
  get(K key) -> T[k]
  remove(K key) -> T[k] = null

Sigla (chave)	Capital (valor)
"MS"	"Campo Grande"
"PB"	"João Pessoa"
"SP"	"São Paulo"
"RS"	"Porto Alegre"

- Universo de chaves: U = {"MS", "PB", "PA", "AM", "PE", "RJ", "CE", ….}
- Chaves em uso: K = {"MS", "PB", "SP", "RS"}

Como implementar?

Qual a estrutura que pode nos dar acesso direto?

Como implementar?

Sigla (chave)	pos
"MS"	0
"PB"	1
"SP"	2
"RS"	3

	Capital
0	"Campo Grande"
1	"João Pessoa"
2	"São Paulo"
3	"Porto Alegre"

Como mapear as chaves em valores? Tabela (vetor) indexada por inteiros.

Tabelas Hash

- Função Hash: calcula um valor inteiro a partir de uma chave.
  - Determinística

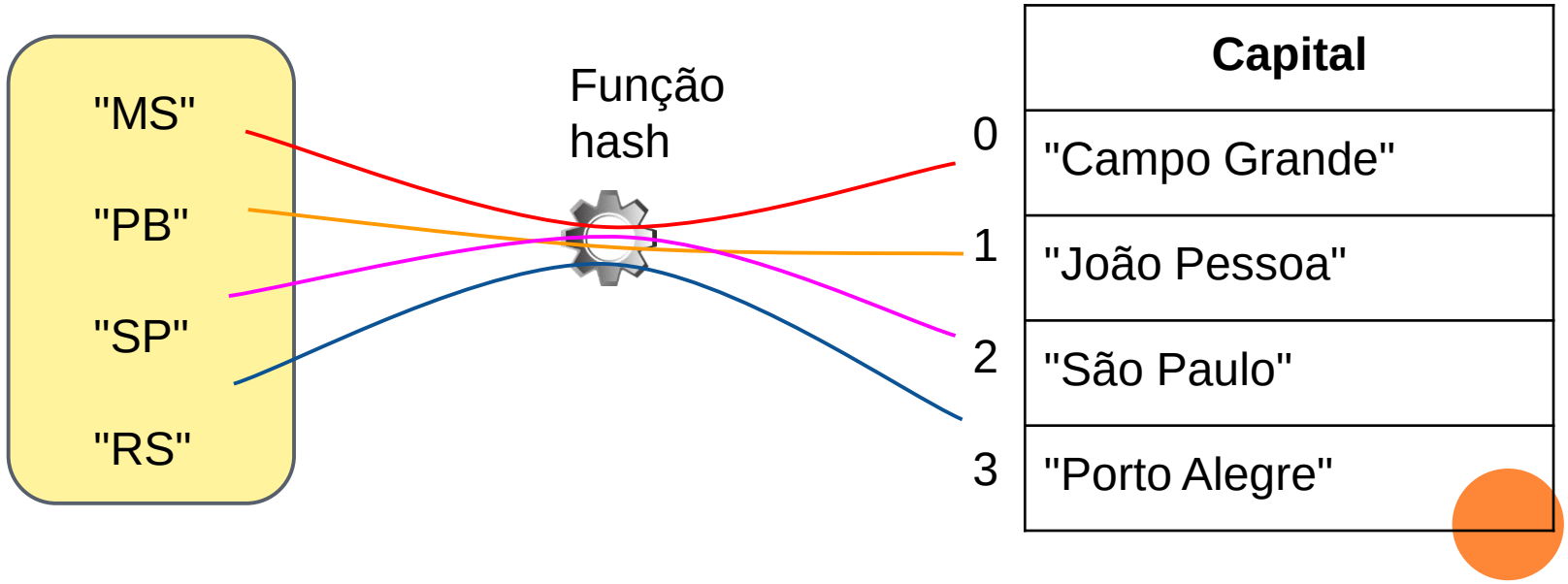


Tabela de Acesso Direto

|U| == |T|

hash(K key) -> {0,1,2,3,...,m}

- Sobrejetora: toda chave é mapeada para um valor
- Injetora: cada valor é referente a apenas uma chave

No nosso caso

hash(K key) -> {0,1,2,3,...,26}

```
put(Key k, Element e)
    T[hash(k)] = e

remove(Key k)
    T[hash(k)] = null

search(Key k)
    return T[hash(key)]
```

O(1)

E SE...

Conjunto de chaves (U) for muito grande? Muito maior que o tamanho da estrutura (T) para armazena-lás?

$$|U| >> |T|$$

Exemplo: Matrículas de alunos da UFCG (8 dígitos)

- Quantas chaves possíveis?

Tabela Hash

- K: conjunto de chaves efetivamente utilizadas
  - Apenas dos alunos de CC, por exemplo
- |U| >> |K|
  - A tabela precisa apenas ter |K| posições
  - Espaço gasto O(|K|)

Tabelas Hash

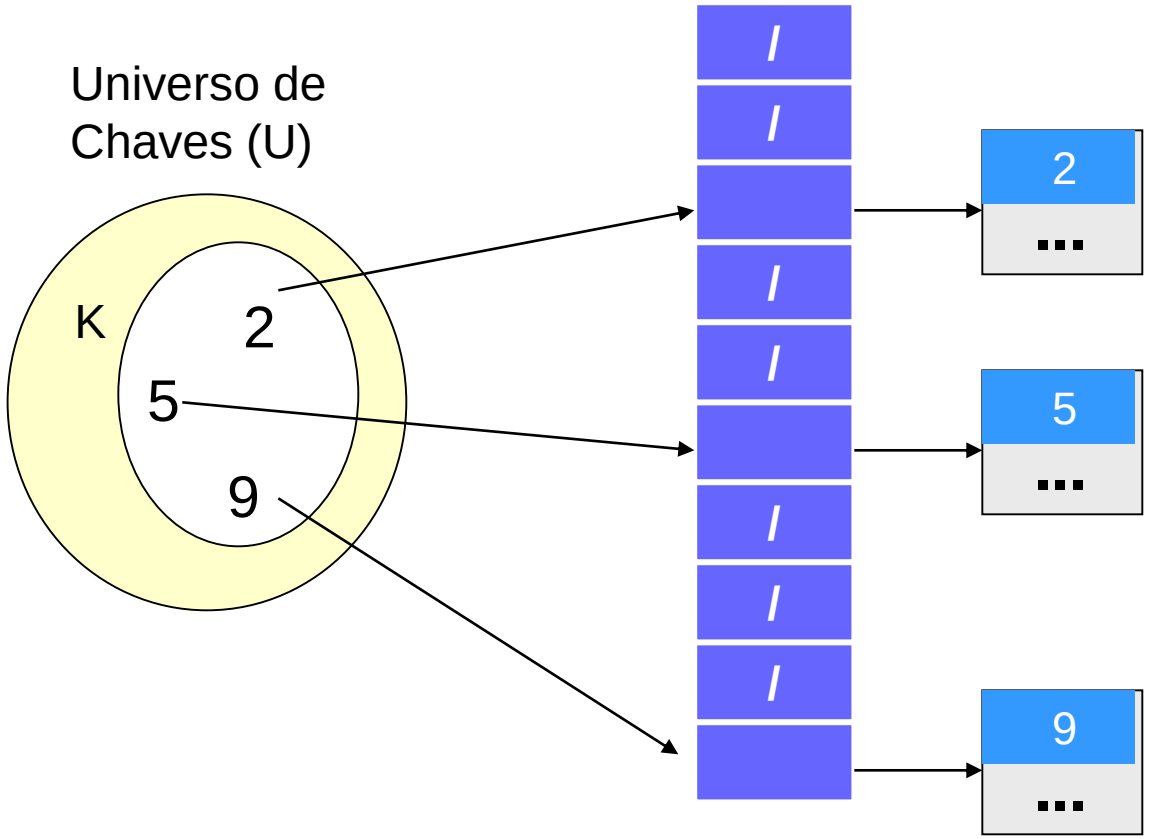
- Também conhecidas com o tabelas de dispersão
- Resolve o problema quando |U| > length(T) mantendo o tempo de acesso em O(1) em média
- Usa uma funcao de hashing (tem que ser determinística) para mapear chaves em índices

$$h : U \rightarrow \{0,\ldots,m-1\}$$

- Se |U| > length(T) pelo menos duas chaves são mapeadas para o mesmo índice (colisão)
  - É impossível não ter colisões
  - Elas devem ser minimizadas

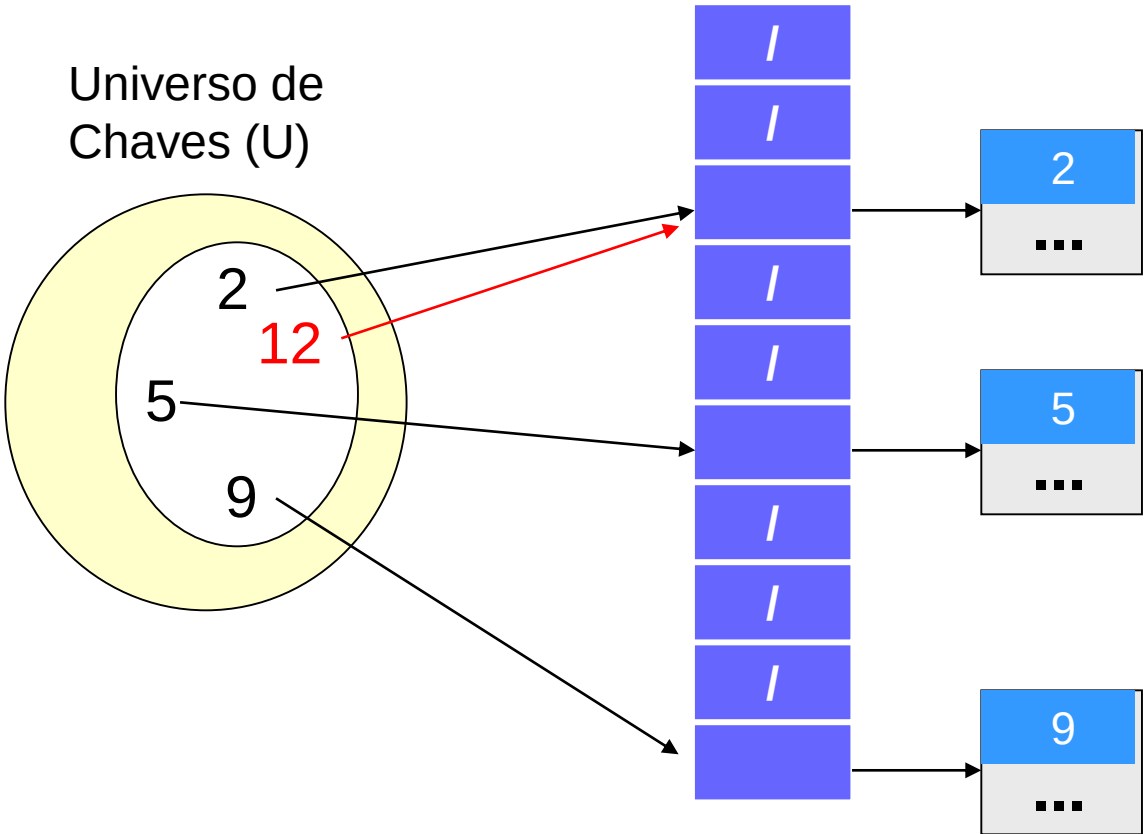
Exemplo

- Seja h(k)=k%10 uma função hash que mapeia de chaves do universo K em {2,5,9,100,107}. Considere uma tabela com m=10 células.



Exemplo

- O que acontece se utilizarmos a função hash h(k)=k%10 para inserir os elementos {2,5,9,12}?



Colisões

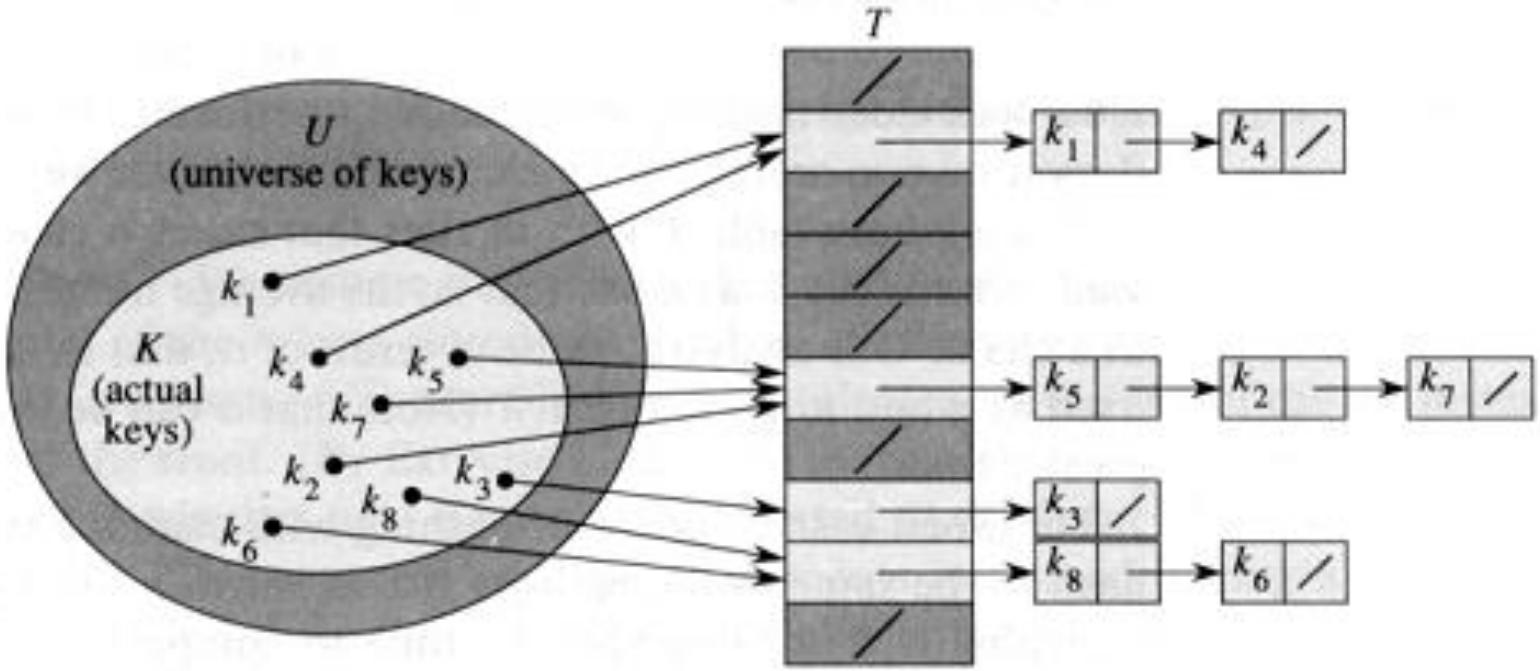
- Tabela hash ideal seria aquela sem colisão
  - Impossível quando |U| > m (tamanho da tabela)
- Existem mais chaves que posições
  - Duas ou mais chaves são mapeadas em um mesmo índice
- Escolher uma boa função hash para minimizar as colisões

Resolução de Colisões (Abordagens)

- **Endereçamento fechado**
  - Elementos com mesmo código hash são colocados na mesma posição da tabela (chaining)
    - Na verdade, inseridos na cabeça da lista já que existe a idéia que o o elemento inserido será usado em breve
- **Endereçamento aberto**
  - Chaves com mesmo código hash são colocadas em posições diferentes

Endereçamento Fechado: Chaining

- Usar uma lista encadeada



Exercício

- Implemente as operações (inserir, remover, pesquisar) utilizando a resolução de conflitos by chaining?
  - Inserir(T,x)
  - Remover(T,x)
  - Pesquisar(T,k)

Exercício

- Implemente as operações (inserir, remover, pesquisar) utilizando a resolução de conflitos by chaining?
  - Inserir(T,x)
    - Inserir x na cabeça de T[h(key(x))];
  - Remover(T,x)
    - Deletar x da lista T[h(key(x))];
  - Pesquisar(T,k)
    - Buscar x na lista T[h(k)];

Exercício

- Implemente as operações (inserir, remover, pesquisar) utilizando a resolução de conflitos by chaining?
  - Inserir(T,x)
    - Inserir x na cabeça de T[h(key(x))];
  - Remover(T,x)
    - Deletar x da lista T[h(key(x))];
  - Pesquisar(T,k)
    - Buscar x na lista T[h(k)];

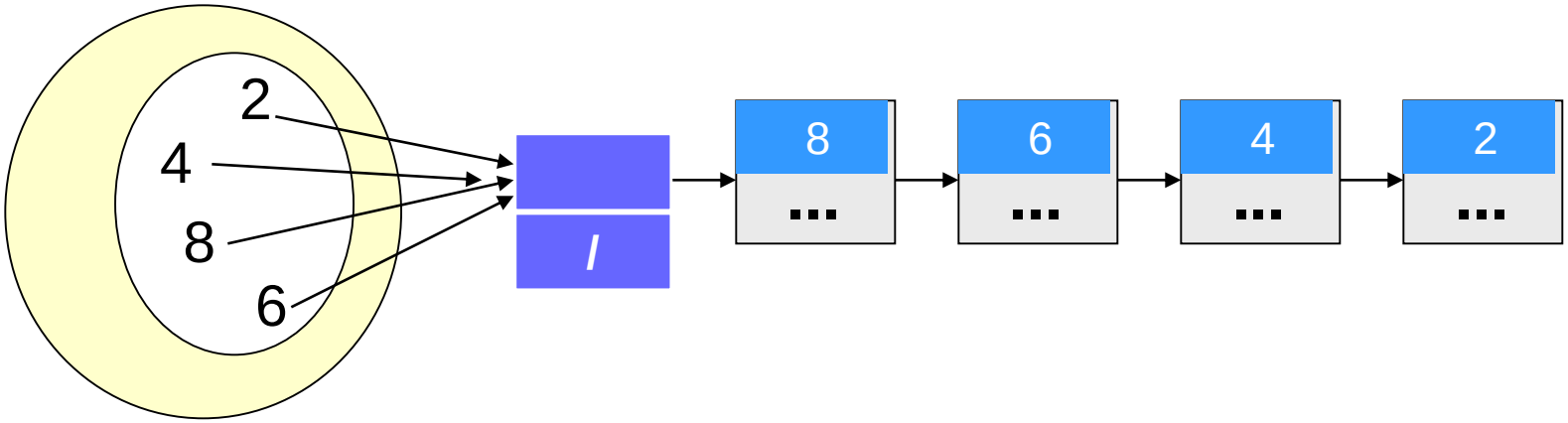
Faça a análise do pior caso das operações (inserir, remover, pesquisar)

Exercício

- Qual seria a tabela final considerando endereçamento fechado (resolução por chaining) e função de hash h(k)=k%2 e chaves do universo K={2,4,6,8}?

Exercício

- Qual seria a tabela final considerando endereçamento fechado (resolução por chaining) e função de hash h(k)=k%2 e chaves do universo K={2,4,6,8}?

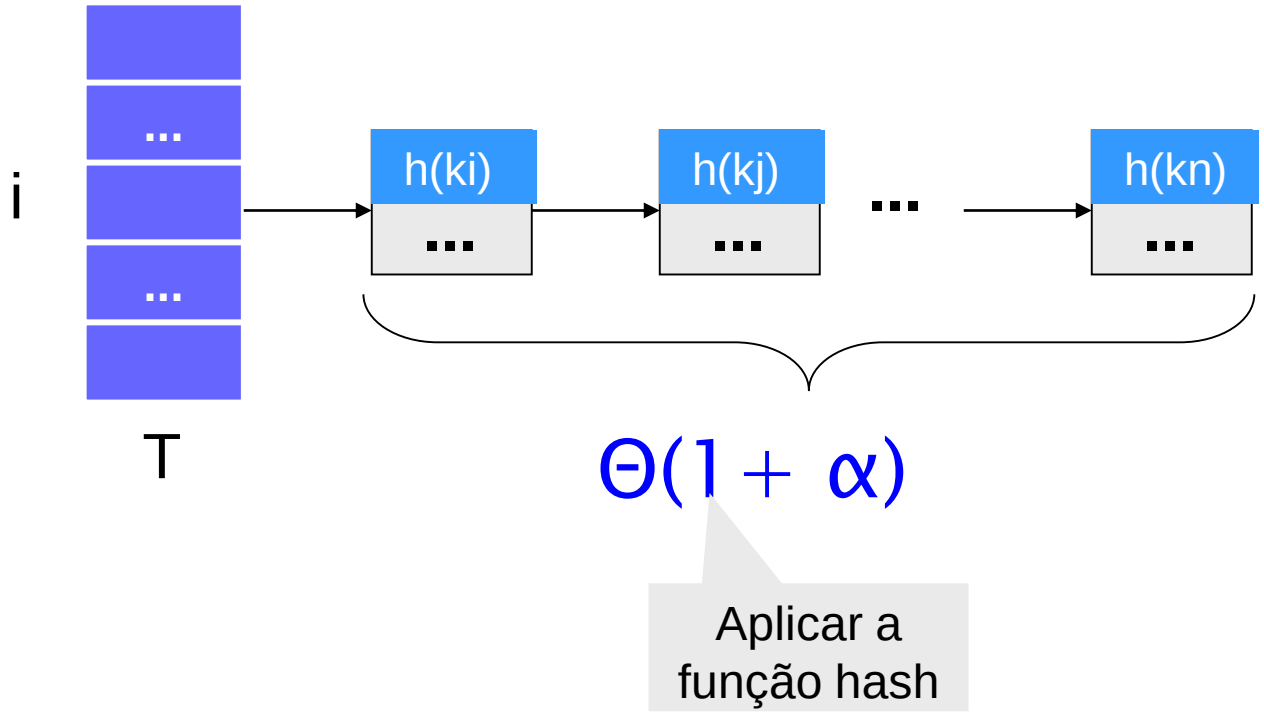


Fator de Carga

- A função h faz um hashing uniforme simples
  - Chaves possuem igual probabilidade de serem mapeadas para qualquer um dos indices
- Cada célula da tabela vai ter um número mais ou menos igual de elementos
- Origina a noção de *load factor* (fator de carga) α (numero medio de elementos em cada lista)
  - n = número de chaves
  - m = número de células na tabela
  - α = n/m
- Exemplo: qual o fator de carga de uma tabela hash com 50 células e 100 chaves?

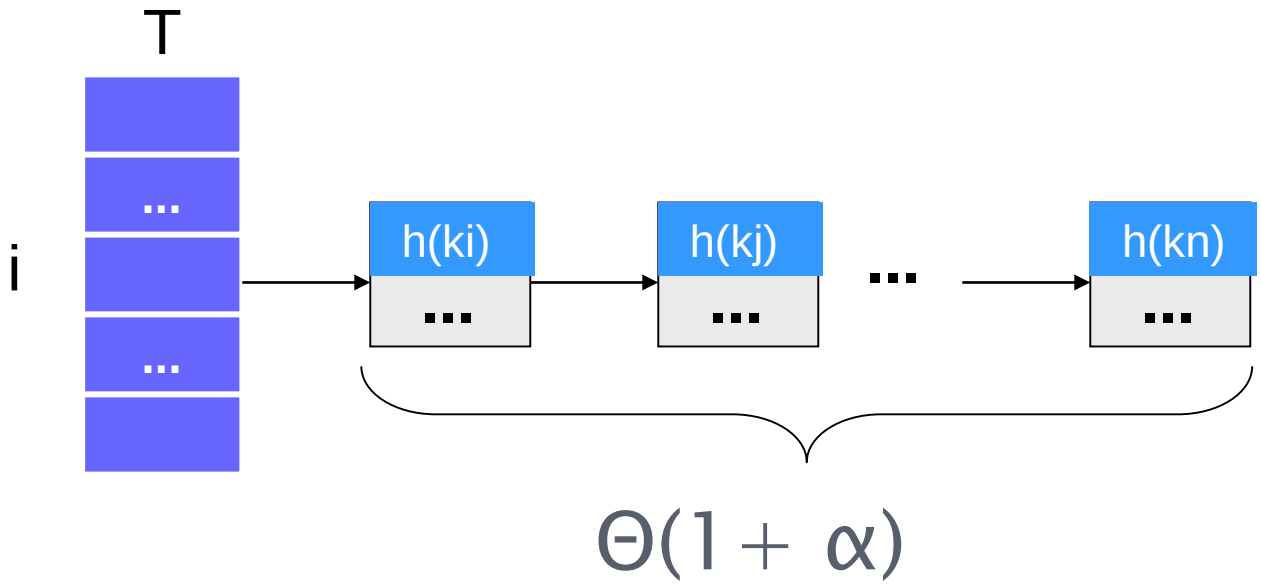
Exercício

- Qual o limite assintoticamente restrito usando o fator de carga (α) para uma pesquisa?



Exercício

- Quando uma pesquisa será feita em Θ(1)?
  - α = n/m = O(1)
  - n = O(m)



Exercício

- Vamos utilizar a função hash h(k) = k%m (m=10) para guardar os CPFs. Como vai ficar a tabela?
- Será que tem alguma função hash melhor?

Como escolher uma função hash?

Características da Função Hashing

- Minimizar o número de colisões
- Satisfazer a hipótese de hashing uniforme
  - Qualquer chave tem a mesma probabilidade de ser mapeada em qualquer um dos m slots. Mas é difícil
- Alguns métodos ajudam
  - Divisão
  - Multiplicação
- Consideramos as chaves como números
  - Fácil de mapear Strings,... para números

Método da Divisão

- Usa o resto da divisão para encotrar valores hash
- Neste método a função hash é da forma:

$$h(k) = k \bmod m$$

Número de células da tabela

- Método rápido (requer apenas uma divisao)

Exercício

- Seja K = {2,4,12,10,5}, desenhe a tabela final considerando as seguintes funções hash (utilize a resolução de conflitos por chaining). As tabelas possuem 2, 5 e 10 células
  - h=k mod 2
  - h=k mod 5
  - h=k mod 10

Método da Divisão

- O método da divisão garantes boas funcoes de hashing?
- O que basicamente muda entras as funções hash no método da divisão?
- Que valores escolhemos para m?

Método da Divisão

- O método da divisão garantes boas funcoes de hashing?
- O que basicamente muda entras as funções hash no método da divisão?
- Que valores escolhemos para m?
- Não escolha um m tal que tenha um divisor d bem pequeno
- Não escolha $m = 2^r$,
- Escolha m sendo um número primo que não seja próximo das potências de 2 ou 10

Método da Divisão

- Suponha que um conjunto de 2000 elementos será armazenado em uma tabela hash onde o fator de carga estimado é 3. Qual seria a melhor função de hash para o problema usando o método da divisão?

Método da Divisão

- Suponha que um conjunto de 2000 elementos será armazenado em uma tabela hash onde o fator de carga estimado é 3. Qual seria a melhor função de hash para o problema usando o método da divisão?

  - h(k) = k % **m**
  - 2000/3 ~ 667
  - Primos: 673, 677,683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, ...
  - Potencias de 2: 256, 512, 1024, 2048, ...

Exercício

- Suponha as chaves {200, 205, 210, 215, 220, ..., 595}.
  - Se eu escolher a função h(k) = k %100. Vai existir colisão em alguma célula? Se sim, quantas?
  - Se eu escolher a função h(k) = k %101. Vai existir colisão? Se sim, quantas?

Exercício

- Suponha as chaves {200, 205, 210, 215, 220, ..., 595}.
  - Se eu escolher a função h(k) = k %100. Vai existir colisão em alguma célula? Se sim, quantas?
    - Cada célula preenchida terá quatro chaves
  - Se eu escolher a função h(k) = k %101. Vai existir colisão? Se sim, quantas?
    - Cada célula preenchida terá uma chave

K % 100

0	200, 300, 400, 500
...	
5	205, 305, 405, 505
...	
10	210, 310, 410, 510
...	

K % 101

Multiplos de 101: 101 202 303 404 505

200	205	210	215	220, ... , 295, 300
099	003	008	013	018, ... , 093, 098
305	310	315	320	325, ... , 395, 400
002	007	012	017	022, ... , 092, 097
405	410	415	420	425, ... , 495, 500
001	006	011	016	021, ... , 091, 096

Método da Multiplicação

- Neste método a função hash é da forma:

$$h(k) = \lfloor m\ (k.A \bmod 1) \rfloor$$

Extrai a parte fracionária de k.A

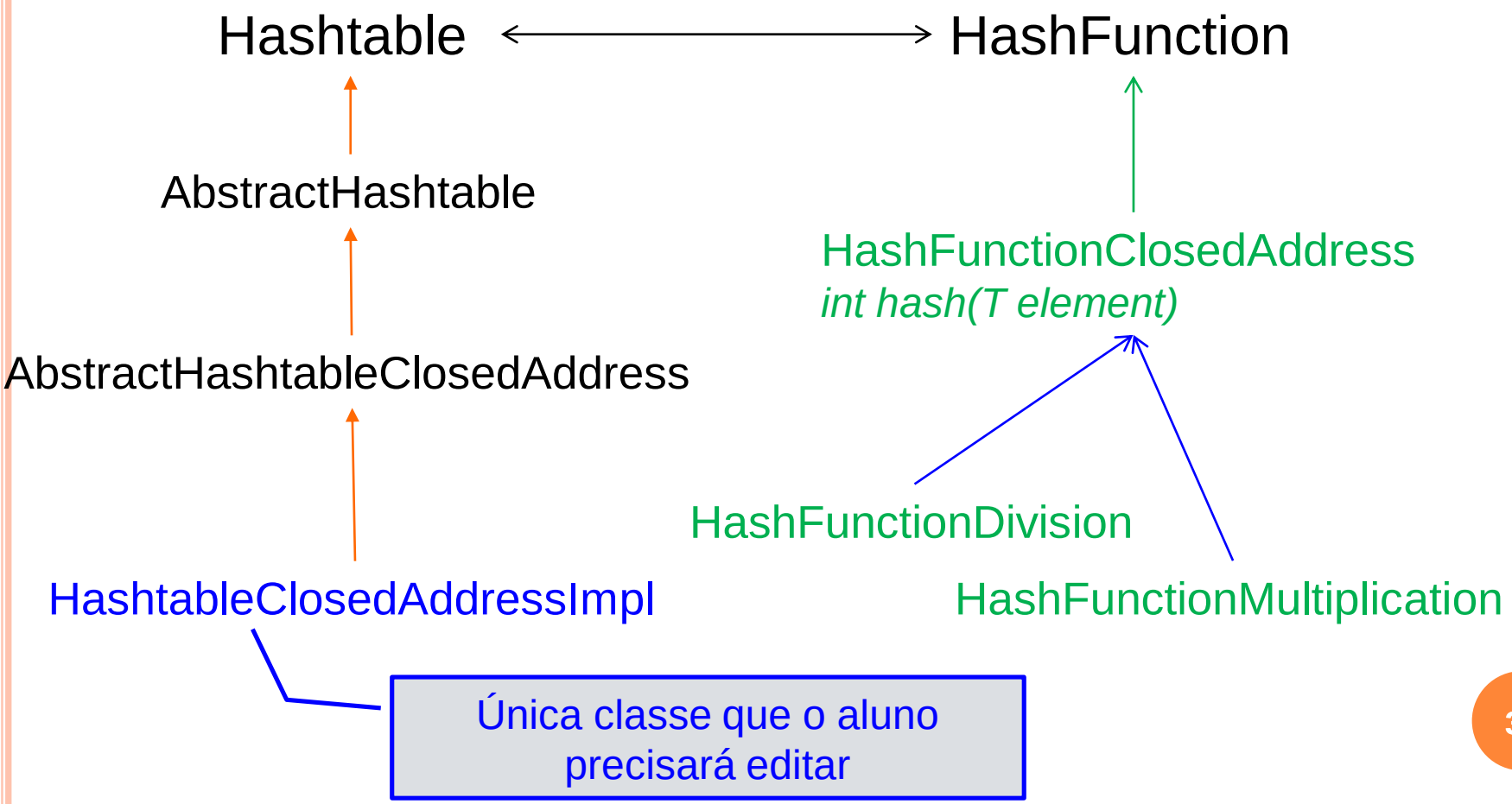
  - k é a chave
  - A é uma constante entre (0..1)
    - Um bom valor é $A \approx (\sqrt{5} - 1)/2 = 0.6180339887...$
  - m (número de células) geralmente é 2^p, onde p é inteiro
- O valor de m não é crítico como no método da divisão
- Um pouco mais lento do que o da divisão

Exercício

- Calcule o codigo hash $h = \lfloor m\ (kA \bmod 1) \rfloor$ das chaves ({2, 3, 4, 5, 6, 10, 15}), onde A = 0.2 e m = 1000

$h(2) = \lfloor 1000\ (2x0.2 \bmod 1) \rfloor = 400$
$h(3) = \lfloor 1000\ (3x0.2 \bmod 1) \rfloor = 600$
$h(4) = \lfloor 1000\ (4x0.2 \bmod 1) \rfloor = 800$
$h(5) = \lfloor 1000\ (5x0.2 \bmod 1) \rfloor = 000$
$h(6) = \lfloor 1000\ (6x0.2 \bmod 1) \rfloor = 200$
$h(10) = \lfloor 1000\ (10x0.2 \bmod 1) \rfloor = 000$
$h(15) = \lfloor 1000\ (15x0.2 \bmod 1) \rfloor = 000$

Questões de Implementação



Questões de Implementação

```
public interface Hashtable<T> {

   public boolean isEmpty();
   public boolean isFull();
   public int capacity();
   public int size();
   public void insert(T element);
   public void remove(T element) ;
   public T search(T element);
   public int indexOf(T element);
}
```

```
public interface HashFunction<T> {

}
```

Questões de Implementação

```
public abstract class AbstractHashtable<T> implements Hashtable<T> {

    Object[] table;
    int elements;
    int COLLISIONS;
    HashFunction<T> hashFunction;

    public AbstractHashtable(){
      elements = 0;
      COLLISIONS = 0;
    }
    protected void initiateInternalTable(int size){
      this.table = Util.<T>makeArray(size);
    }

    @Override
    public boolean isEmpty() {
      return (elements == 0);
    }
```

Questões de Implementação

```
    @Override
    public boolean isFull() {
      return (elements == table.length);
    }

    @Override
    public int size() {
      return elements;
    }

    @Override
    public int capacity(){
      return this.table.length;
    }

    public int getCOLLISIONS() {
      return COLLISIONS;
    }
}
```

Questões de Implementação

```
public abstract class AbstractHashtableClosedAddress<T>
        extends AbstractHashtable<T> {

    public AbstractHashtableClosedAddress(){
        super();
    }

    @Override
    protected void initiateInternalTable(int size){
        this.table = Util.<LinkedList<T>>makeArray(size);
    }
}
```

```
public static <T> T[] makeArray(int size) {
   T[] array = (T[]) new Object[size];
   return array;
}
```

Questões de Implementação

```
public interface HashFunction<T> {
}
```

```
public interface HashFunctionClosedAddress<T>
       extends HashFunction<T> {

  public int hash(T element);
}
```

Questões de Implementação

```
public class HashFunctionDivisionMethod<T>
       implements HashFunctionClosedAddress<T> {

  protected int tableSize;

  public HashFunctionDivisionMethod(int tableSize) {
    this.tableSize = tableSize;
  }

  public int hash(T element) {
    int hashKey = -1;
    int key = element.hashCode();

    hashKey = (int) key % tableSize;

    return hashKey;
  }
}
```

Questões de Implementação

```
public class HashFunctionMultiplicationMethod<T>
       implements HashFunctionClosedAddress<T>{

  protected int tableSize;
  private static final double A = (Math.sqrt(5)-1)/2;

  public HashFunctionMultiplicationMethod(int tableSize){
    this.tableSize = tableSize;
  }

  public int hash(T element) {
    int hashKey = -1;
    int key = element.hashCode();
    double fractionalPart = key*A - Math.floor(key*A);
    hashKey = (int) (tableSize * fractionalPart);;

    return hashKey;
  }
}
```

Questões de Implementação

```
public enum HashFunctionClosedAddressMethod {
   DIVISION, MULTIPLICATION;
}

public class HashFunctionFactory<T> {
    public static HashFunction
        createHashFunction(HashFunctionClosedAddressMethod
           method, int tableSize){
        HashFunction result = null;
        switch (method) {
            case DIVISION:
                result = new
                HashFunctionDivisionMethod(tableSize);
            case MULTIPLICATION:
                result = new
                HashFunctionMultiplicationMethod(tableSize);
        }
        return result;
    }
}
```

Questões de Implementação

```
public class HashtableClosedAddressImpl<T> extends
      AbstractHashtableClosedAddress<T> {

   public HashtableClosedAddressImpl(int desiredSize,
      HashFunctionClosedAddressMethod method) {
      int realSize  = desiredSize;

      if(method == DIVISION){
        realSize = this.getPrimeAbove(desiredSize);
      }
      initiateInternalTable(realSize);
      HashFunction function =
       HashFunctionFactory.createHashFunction(method,realSize);
      this.hashFunction = function;
   }
```

Questões de Implementação

```
    int getPrimeAbove(int number){
      //use Util.isPrime(n)
    }

    public void insert(T element) {
    }

    public void remove(T element) {
    }

    public T search(T element) {
    }

    public int indexOf(T element) {
    }

}
```

Tabela com Endereçamento Aberto

- Até agora vimos a resolução de conflitos por chaining (Endereçamento Fechado)
  - Precisa de uma estrutura de dados auxiliar (lista)
- E se não tivermos listas para resolver as colisões?
  - Precisamos de um número m bem maior de células.
  - Chaves que colidam precisam ter seu hash code "recalculado" para encontrar um slot vazio
    - Todas as chaves estão em alguma célula. Cada entrada da tabela ou contem um elemento ou NIL

Outra Forma de Resolver Conflitos

- Endereçamento aberto faz uso de um *probe*.
  - Examina a tabela (sequencial ou nao) até encontrar um slot vazio.
  - A sequencia de posicoes sendo examinadas dependem da chave
- Função hash é estendida

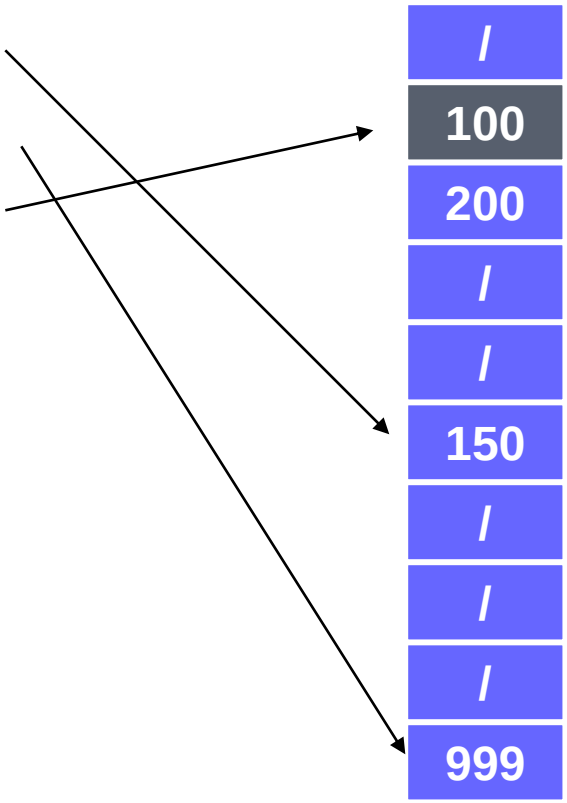
$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

  - posicao = h(k,p)
  - k = chave
  - p = probe

Exemplo: Inserção

- Insira a chave k=100
  - Probe h(100,0)
  - Probe h(100,1)
  - Probe h(100,2)

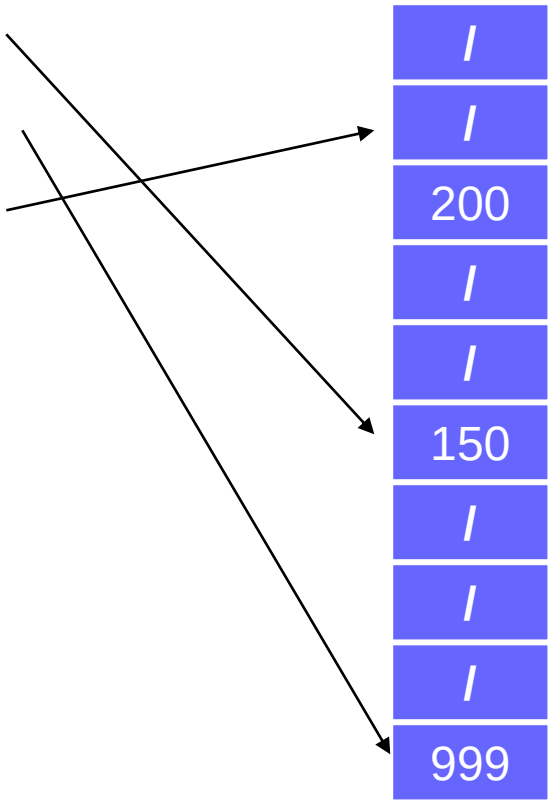


Algoritmo: Inserção

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j] = NIL
4            then T[j] ← k
5                 return j
6            else i ← i + 1
7     until i = m
8  error "hash table overflow"
```

Exemplo: Pesquisa

- Pesquisa a chave k=100
  - Probe h(100,0)
  - Probe h(100,1)
  - Probe h(100,2)



ALGORITMO: PESQUISA

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j] = k
4            then return j
5         i ← i + 1
6    until T[j] = NIL or i = m
7  return NIL
```

Endereçamento Aberto (Deleção)

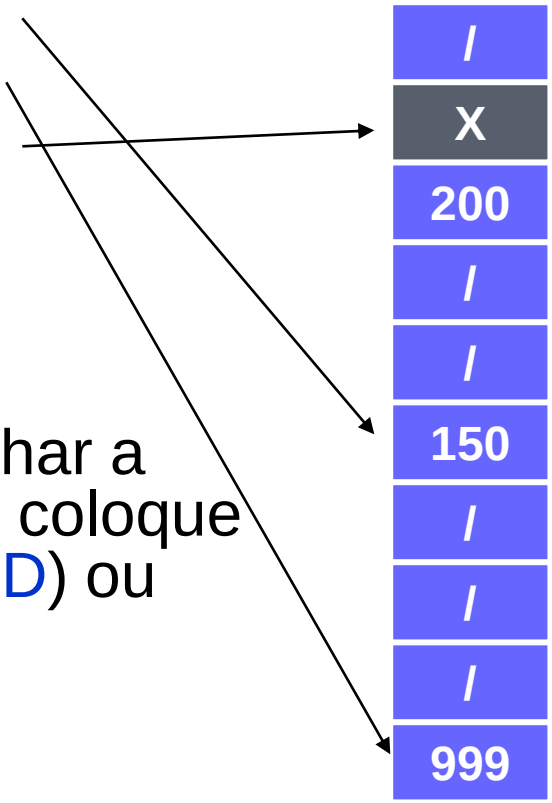
- Como fazer uma remocao com enderecamento aberto?
  - Podemos atribuir NIL diretamente na posição da chave a ser removida?

Endereçamento Aberto (Deleção)

- Como fazer uma remocao com enderecamento aberto?
  - Nao podemos simplesmente dizer que o slot esvaziou colocando NIL
  - Usar uma flag (valor especial DELETED) ao invés de NIL
  - Alterar as funções
    - Inserir (inserir no flag se ele for DELETED)
    - Pesquisar (continuar a pesquisa no flag ao invés de NIL)

EXEMPLO

- Remova a chave k=100
  - Probe h(100,0)
  - Probe h(100,1)
  - Probe h(100,2)



E como ficaria a inserção?

- Pesquisar até achar a chave (remova – coloque um flag DELETED) ou uma célula vazia

EXEMPLO

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat j ← h(k, i)
3         if T[j] = NIL  || T[j] = DELETED
4            then T[j] ← k
5                 return j
6            else i ← i + 1
7     until i = m
8  error "hash table overflow"
```

Hashing com Probing

- Como escolher a função hash no endereçamento aberto usando o probe?
- Por que não utilizar a função h(k,i) = k%m?
  - Precisamos usar i para que o endereço mude.
- Tres técnicas são usadas para computar as sequencias de probes
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

Linear Probing

- Dada uma funcao de hash ordinaria $h' : U \rightarrow \{0,1,\ldots,m-1\}$ (obtida por divisão ou multiplicação) o método usa a função de hash da forma:

$$h(k,i) = (h'(k) + i) \bmod m$$

  - Slots buscados linearmente

    $T[(h'(k)]$

    $T[h'(k)+1]$

    ...

    $T[m-1]$

  - Facil de implementar mas apresenta problemas
    - Clustering Primário – clusters podem surgir quando slots ocupados se agrupam. Isso aumenta o tempo da busca.

Exercício

- Seja m=10, h(k) = k mod 5. Utilize o probing linear para inserir os valores ({10,15,2,12,4,8})

$$h(k,i) = (k \bmod 5 + i) \bmod 10$$

Probing Quadrático

- Neste método a função hash é da forma:

$$h(k,i) = (\underbrace{h'(k)}_{} + c_1 i + c_2 i^2) \bmod m$$

Método da Divisão ou Multiplicação

  - c_1 e c_2 são constantes
  - Primeiro elemento buscado T[h'(k)]. Demais variam com funcao quadratica de i.
    - Evita o clustering primário
    - Pode ocorrer um Clustering Secundário – duas chaves com mesmo probe inicial geram mesma sequencia de probes
  - Mas é melhor do que o anterior

Exercício

- Seja m=10, c_1=3, c_2=5, h(k) = k mod 5. Utilize o probing quadrático para inserir os valores ({10,15,2,12,4,8}):

$$h(k,i) = (k \bmod 5 + 3i + 5i^2) \bmod 10$$

Exercício

- Compare a resposta anterior com o linear probing e o endereçamento fechado (h(k) = k mod 10) com chaining para inserir os valores ({10,15,2,12,4,8}):

$$h(k) = k \bmod 10$$
$$h(k,i) = (k \bmod 5 + i) \bmod 10$$
$$h(k,i) = (k \bmod 5 + 3i + 5i^2) \bmod 10$$

Hashing Duplo

- Um dos melhores metodos para endereçamento aberto
- Neste método a função hash é da forma:

$$h(k,i) = (h_1(k) + i.h_2(k)) \bmod m$$

Método da Divisão ou Multiplicação

  - A sequencia de probe depende de duas formas em k.
  - Dicas
    - Faça $h_2(k)$ retornar um número ímpar e m ser um número 2^p
    - Faça m ser um numero primo e $h_2(k)$ retornar um inteiro positivo menor que m
  - Excelentes resultados
  - Produz Θ(m) sequências (se aproxima mais do hashing uniforme simples)

Exercício

- Seja m=24, $h_1(k) = k \bmod 5$ e $h_2(k) = k+1 \bmod 7$. Utilize o hashing duplo para inserir os valores ({10,15,2,12,4,8}):

$$h(k,i) = (k \bmod 5 + (k+1 \bmod 7).i) \bmod 24$$

POSCOMP 2009

Questão 31. [FUN]

Considere uma tabela de espalhamento (tabela *hash*) de comprimento $m = 11$, que usa endereçamento aberto (*open addressing*), a técnica de tentativa linear *(linear probing*) para resolver colisões e com a função de dispersão (função *hash*) $h(k) = k \ mod \ m$, onde k é a chave a ser inserida. Considere as seguintes operações sobre essa tabela:

- Inserção das chaves 3, 14, 15, 92, 65, 35 (nesta ordem);
- Remoção da chave 15; e
- Inserção da chave 43.

Escolha a opção que representa esta tabela após estas operações:

A) 65 – ø – 35 – 14 – ø – 92 – 3 – ø – ø – ø – 43
B) 43 – ø – 35 – 3 – 14 – 92 – ø – ø – ø – ø – 65
C) 65 – ø – 35 – X - 14 – 92 – 3 – ø – ø – ø – 43
D) 65 – ø – 35 – 3 – 14 – 92 – ø – ø – ø – ø – 43
E) 43 – ø – 35 – 3 – 14 – X – 92 – ø – ø – ø – 65

POSCOMP 2009

Questão 31. [FUN]

Considere uma tabela de espalhamento (tabela *hash*) de comprimento $m = 11$, que usa endereçamento aberto (*open addressing*), a técnica de tentativa linear *(linear probing*) para resolver colisões e com a função de dispersão (função *hash*) $h(k) = k \ mod \ m$, onde k é a chave a ser inserida. Considere as seguintes operações sobre essa tabela:

- Inserção das chaves 3, 14, 15, 92, 65, 35 (nesta ordem);
- Remoção da chave 15; e
- Inserção da chave 43.

Escolha a opção que representa esta tabela após estas operações:

A) 65 – ø – 35 – 14 – ø – 92 – 3 – ø – ø – ø – 43
B) 43 – ø – 35 – 3 – 14 – 92 – ø – ø – ø – ø – 65
C) 65 – ø – 35 – X - 14 – 92 – 3 – ø – ø – ø – 43
D) 65 – ø – 35 – 3 – 14 – 92 – ø – ø – ø – ø – 43
E) 43 – ø – 35 – 3 – 14 – X – 92 – ø – ø – ø – 65

Fator de Carga

- Qual o fator de carga no endereçamento aberto?
  - O load factor é ≤ 1
- Por que?

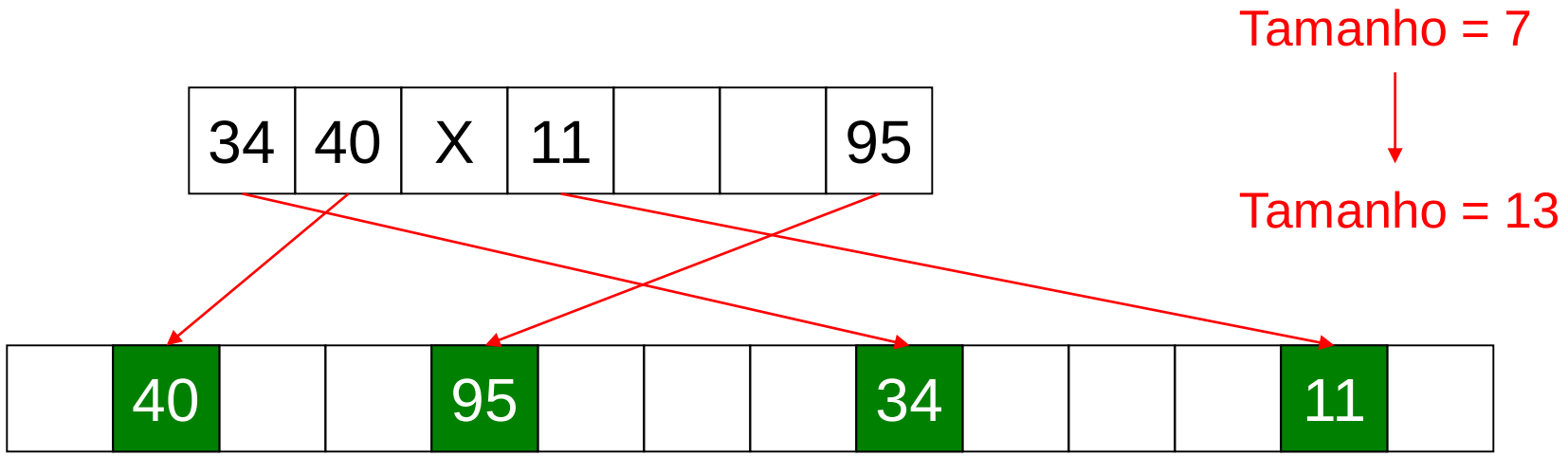
Lidando com Overflow

- O que fazer quando o vetor interno da tabela hash ficar **cheio**?
  - **Duplicar** quando estiver cheio?
  - Duplicar quando atingir uma certa **capacidade?**

Rehashing

- Quando a capacidade preenchida (exceder 50%), as operações na tabela passam a demorar mais, pois o número de colisões aumenta
- Expandir o array que constitui a tabela, e reorganizar os elementos na nova tabela
  - Novo tamanho: número primo próximo ao dobro da tabela atual

Exemplo




```
Object put() { ...
    if (++size > threshold)
        rehash(); ...
}
void rehash() { ...
    newcapacity = prime number near 2*buckets;
    threshold = (int) (newcapacity * loadFactor);
    buckets = new HashEntry[newcapacity]; ...
}
```

Exercício

- Quando devemos fazer o rehashing?
  - No endereçamento aberto não conseguirmos mais inserir um elemento
  - Quando metade da tabela estiver ocupada (limite de capacidade)

Implementação

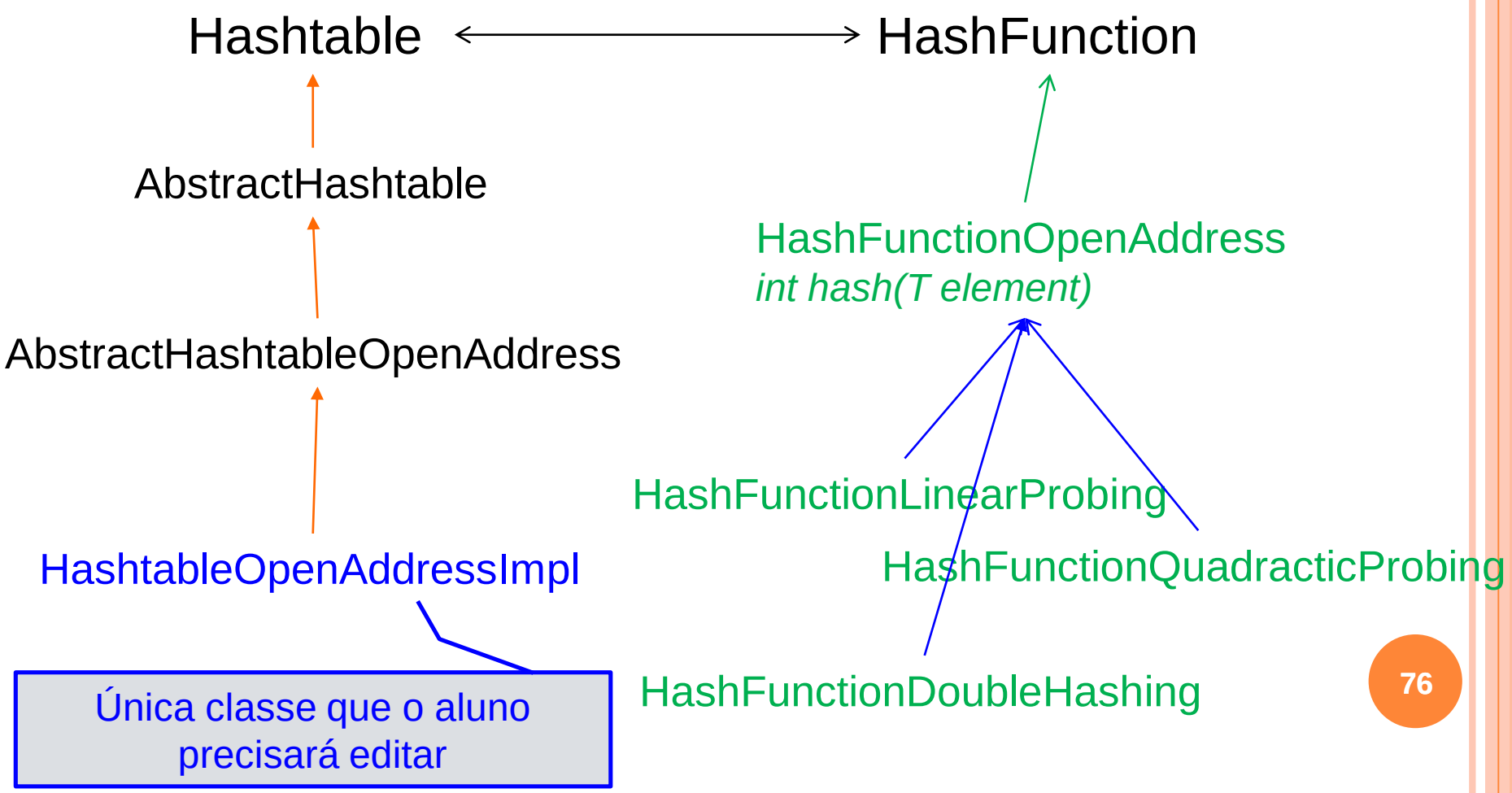
- Hashtable
  - http://www.docjar.com/html/api/java/util/Hashtable.java.html
- Ver funções
  - hash()
    - O código hash é delegado para cada classe
  - containsKey()
  - get()
  - put()
  - remove()
  - rehash()
- equals( ) x hashCode( ).
  - a.equals(b) ⇔ hashCode(a) = hashCode(b)

Tabela Hash (Implementação)

- Separando a tabela de sua função hash?

Hashtable ⟷ HashFunction

Questões de Implementação



Questões de Implementação

```
public abstract class AbstractHashtableOpenAddress<T>
        extends AbstractHashtable<T> {

    protected final DELETED deletedElement = new DELETED();
    private int tableSize;

    public AbstractHashtableOpenAddress(){
        this.tableSize = tableSize;
        this.initiateInternalTable(size);
    }
    @Override
    protected void initiateInternalTable(int size){
        this.table = Util.<Storable>makeArray(size);
    }
}
```

```
public static <T> T[] makeArray(int size) {
  T[] array = (T[]) new Object[size];
  return array;
}
```

Questões de Implementação

```
public abstract class AbstractHashtableOpenAddress<T>
        extends AbstractHashtable<T> {

    protected final DELETED deletedElement = new DELETED();
    private int tableSize;

    public AbstractHashtableOpenAddress(){
        this.tableSize = tableSize;
        this.initiateInternalTable(size);
    }
    @Override
    protected void initiateInternalTable(int size){
        this.table = Util.<Storable>makeArray(size);
    }
}
```

```
public interface Storable {
}
```

Questões de Implementação

```
public interface HashFunction<T> {
}
```

```
public interface HashFunctionOpenAddress<T>
       extends HashFunction<T> {

  public int hash(T elemento, int probe);
}
```

Questões de Implementação

```
public HashFunctionLinearProbing<T> implements
        HashFunctionOpenAddress<T> {
  protected HashFunctionClosedAddress<T> originalHashFunction;
  protected int tableCapacity;

  public HashFunctionLinearProbing(int tableCapacity,
        HashFunctionClosedAddressMethod method) {
    this.tableCapacity = tableCapacity;
    if(method == HashFunctionClosedAddressMethod.DIVISION){
        originalHashFunction = new
        HashFunctionDivisionMethod<T>(tableCapacity)
    }else{
        originalHashFunction = new
        HashFunctionMultiplicationMethod<T> (tableCapacity);
    }
  }
}
```

Questões de Implementação

```
@Override
public int hash(T element, int probe) {
    int generatedIndex = 0;

    generatedIndex= ((originalHashFunction.hash(element)
    + probe) % this.hashtable.capacity());

    return generatedIndex;
}
```

Questões de Implementação

```
public class HashFunctionQuadraticProbing<T> implements
        HashFunctionOpenAddress<T> {
    private int tableCapacity;
    protected HashFunctionClosedAddress<T> originalHashFunction;
    protected int c1;
    protected int c2;
    public HashFunctionQuadraticProbing(int tableCapacity,
        HashFunctionClosedAddressMethod method,int c1, int c2) {
        this.tableCapacity = tableCapacity;
        if(method == HashFunctionClosedAddressMethod.DIVISION){
            originalHashFunction = new
            HashFunctionDivisionMethod<T>(tableCapacity);
        }else{
            originalHashFunction = new
            HashFunctionMultiplicationMethod<T>(tableCapacity);
        }
        this.c1 = c1;
        this.c2 = c2;
    }
```

Questões de Implementação

```
@Override
public int hash(T element, int probe) {
    int generatedIndex = 0;

    generatedIndex = originalHashFunction.hash(element);
    generatedIndex = ((generatedIndex + c1*probe +
        c2*probe*probe)% tableCapacity);

    return generatedIndex;
}
```

Questões de Implementação

```
public abstract class AbstractHashtableOpenAddress<T extends
        Storable> extends AbstractHashtable<T> {

    protected final DELETED deletedElement = new DELETED();
    private int tableSize;

    public AbstractHashtableOpenAddress(int size) {
        this.tableSize = size;
        this.initiateInternalTable(size);
    }

    @Override
    protected void initiateInternalTable(int size) {
        this.table = Util.<Storable>makeArray(size);
    }
}
```

Questões de Implementação

```
public abstract class AbstractHashtableOpenAddress<T extends
        Storable> extends AbstractHashtable<T> {

    protected final DELETED deletedElement = new DELETED();
    private int tableSize;

    public AbstractHashtable
        this.tableSize = siz
        this.initiateInternal
    }

    @Override
    protected void initiateInternalTable(int size) {
        this.table = Util.<Storable>makeArray(size);
    }
}
```

```
public class DELETED implements
        Storable{
    public DELETED() {
    }
}
```

Questões de Implementação

```
public class HashtableOpenAddressLinearProbingImpl<T extends
       Storable> extends AbstractHashtableOpenAddress<T> {

    public HashtableOpenAddressLinearProbingImpl(int size,
        HashFunctionClosedAddressMethod method) {
        super(size);
        hashFunction = new HashFunctionLinearProbing<T>(size,
                                method);
        this.initiateInternalTable(size);

    }

     //métodos a implementar
}
```

Questões de Implementação

```
public class HashtableOpenAddressQuadraticProbingImpl<T extends
        Storable> extends AbstractHashtableOpenAddress<T> {

    public HashtableOpenAddressQuadraticProbingImpl(int size,
        HashFunctionClosedAddressMethod method, int c1, int c2) {
        super(size);
        hashFunction = new HashFunctionQuadraticProbing<T>(size,
                                method, c1, c2);
        this.initiateInternalTable(size);
    }

    //métodos a implementar
}
```

Questões de Implementação

```
public class HashtableOpenAddressDoubleHashing<T extends
        Storable> extends AbstractHashtableOpenAddress<T> {

    public HashtableOpenAddressDoubleHashing(int size,
        HashFunctionClosedAddressMethod methodFunction1,
        HashFunctionClosedAddressMethod methodFunction2) {
        super(size);
        hashFunction = new HashFunctionDoubleHashing<T>(size,
                                methodFunction1, methodFunction2);
        this.initiateInternalTable(size);
    }

     //métodos a implementar
}
```

Referências

- Capítulo 11
  - 2a edição