ALGORITMOS E ESTRUTURAS DE DADOS 2018/2019
DICIONÁRIO

Armanda Rodrigues

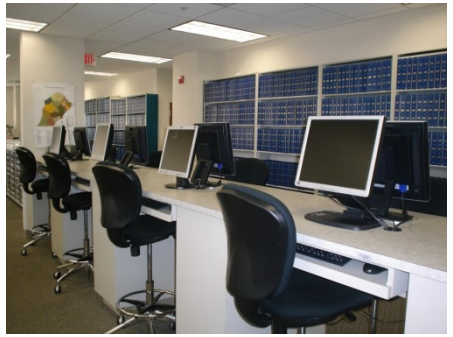
16 de Outubro 2018

Guardar todos os documentos da biblioteca

- Vamos voltar ao nosso exemplo da biblioteca
- Os utilizadores do sistema da biblioteca devem poder executar as seguintes tarefas:
  - Inserir um novo documento na biblioteca – esta operação recebe toda a informação relativa a um documento e só deve ter sucesso se a cota do documento não existe já na biblioteca
  - Remoção de um documento da biblioteca – esta operação só deve ter sucesso se a cota do documento existe na biblioteca e deve devolver o documento removido
  - Aceder à informação associada a um documento da biblioteca - esta operação só deve ter sucesso se a cota do documento existe na biblioteca







TAD Biblioteca

Interface Library – incompleto (1)

```
package library;

public interface Library {

……
    // Add new Book to library
    // Requires: documentCode does not identify an existing document
    void addNewBook(String title, String subject, String documentCode,
         String publisher, String author, long ISBN)
         throws ExistingDocException;

    // Add new Journal to library
    // Requires: documentCode does not identify an existing document
    void addNewJournal(String title, String subject, String documentCode,
         String publisher, int ISSN, String URL)
         throws ExistingDocException;


}
```

Interface Library – incompleto (2)

```
package library;

public interface Library {

……

// Removes document identified by documentCode from library and returns
// removed document.
// Requires: documentCode identifies document in library
Document removeDocument(String documentCode)
                                        throws NonExistingDocException;

// Returns document identified by documentCode from library
// Requires: documentCode identifies document in library
Document getDocument(String documentCode)
                                        throws NonExistingDocException;
}
```

Library – perguntas

- O que representa a String documentCode neste sistema ?
  - Que características deve ter?
- O que representa o tipo Document neste Sistema ?
- Quais as características que gostaríamos de ver associadas às operações apresentadas no TAD ?
- Qual deveria ser a complexidade temporal das operações associadas a implementações do TAD ? Será possível cumprir esse requisito ?

Respostas

- A String documentCode é uma chave de acesso a um documento
- O TAD Dicionário reflete os serviços de acesso a objetos genéricos com uma chave associada
  - Uma implementação de dicionário permitirá guardar a informação relativa a todos os documentos da biblioteca
- Seria aconselhável um acesso rápido a um objeto, a partir da sua chave

TAD Dicionário – Acesso por chave

TAD Dicionário
Chaves do Tipo K e Valores do Tipo V

```
// Se existir uma entrada no dicionário cuja chave é a especificada,
// retorna o seu valor; no caso contrário, retorna NIL.
V ∪ {NIL} pesquisa( K chave );

// Se existir uma entrada no dicionário cuja chave é a especificada,
// substitui o seu valor pelo valor especificado e retorna o valor antigo;
// no caso contrário, insere a entrada (chave, valor) e retorna NIL.
V ∪ {NIL} insere( K chave, V valor );

// Se existir uma entrada no dicionário cuja chave é a especificada,
// remove-a do dicionário e retorna o seu valor;
// no caso contrário, retorna NIL.
V ∪ {NIL} remove( K chave );
```

Interface Dicionário (K,V) (1)

```
package dataStructures;

public interface Dictionary<K,V>{

    // Returns true iff the dictionary contains no entries.
    boolean isEmpty( );

    // Returns the number of entries in the dictionary.
    int size( );

    // Returns an iterator of the entries in the dictionary.
    Iterator<Entry<K,V>> iterator( );
```

- O dicionário guarda valores deste tipo: K refere o tipo que permite fazer o acesso (a chave), V o tipo do valor identificado pela chave
- Quando criamos um dicionário temos de saber qual o tipo da chave e qual o tipo do valor

Interface Dicionário (K,V) (2)

```
    // If there is an entry in the dictionary whose key is the specified key,
    // returns its value; otherwise, returns null.
    V find( K key );

    // If there is an entry in the dictionary whose key is the specified key,
    // replaces its value by the specified value and returns the old value;
    // otherwise, inserts the entry (key, value) and returns null.
    V insert( K key, V value );

    // If there is an entry in the dictionary whose key is the specified key,
    // removes it from the dictionary and returns its value;
    // otherwise, returns null.
    V remove( K key );

}// End of Dictionary
```

TAD Entrada

Acesso aos campos Chave e Valor

Interface Entrada (K,V)

```
package dataStructures;

public interface Entry<K,V>{

    // Returns the key in the entry.
    K getKey( );

    // Returns the value in the entry.
    V getValue( );
}
```

TAD Objecto Comparável

Comparação com outro Objecto

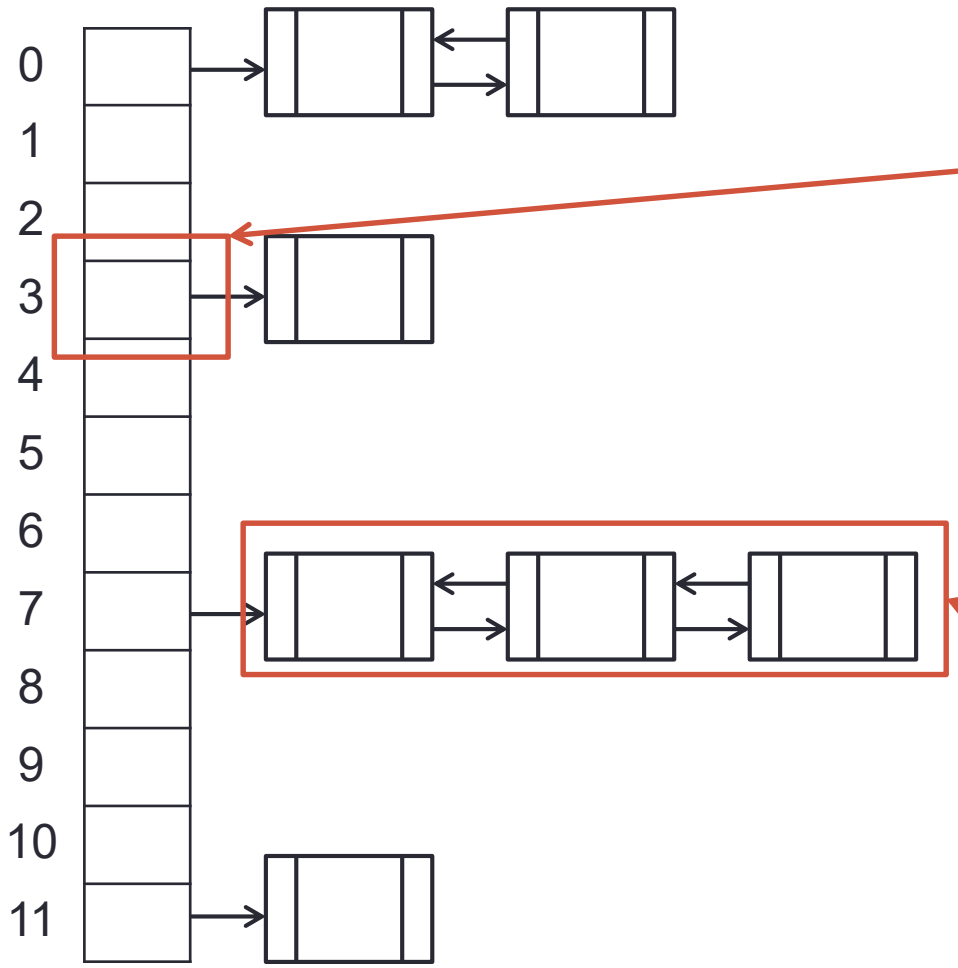
Interface Objeto Comparável Objetos do Tipo T

```
package java.lang;

public interface Comparable<T>{


   // Compares this object with the specified object for order.
   // Returns a negative integer, zero, or a positive integer
   // as this object is less than, equal to, or greater than
   // the specified object.
   int compareTo( T object );
}
```

Tabela de Dispersão

- A estrutura de dados Tabela de Dispersão é suportada pelo conceito de função de dispersão
- Dada uma chave que identifica univocamente um objeto, a função de dispersão converte a mesma chave num número inteiro dentro de um intervalo determinado
- A utilização da função de dispersão permite aceder ao objeto que a chave identifica de uma forma expedita
- A utilização da função de dispersão para localizar um objeto comporta a possibilidade de se darem colisões
  - Dois objetos diferentes podem ter a mesma dispersão, o que implica que estes dois objetos poderão ser armazenados na mesma zona
  - As colisões têm de ser tratadas e resolvidas

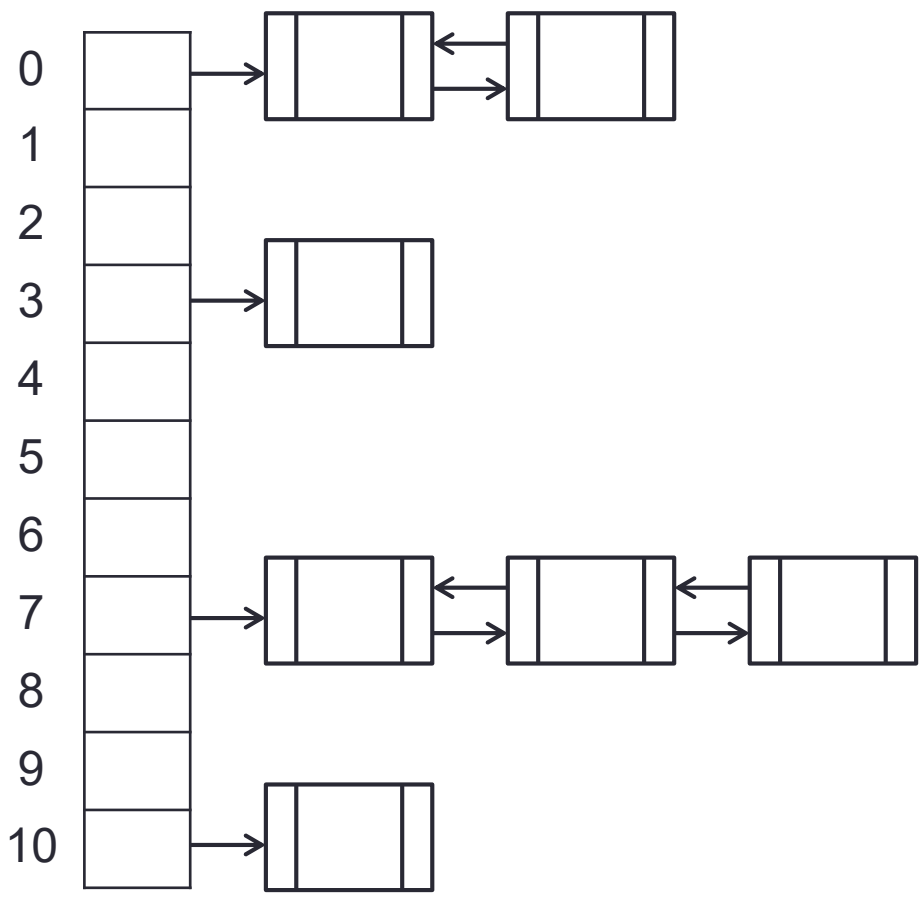
Tabela de dispersão aberta



A função de dispersão converte a chave que identifica um objeto a guardar na TD, num índice do vetor

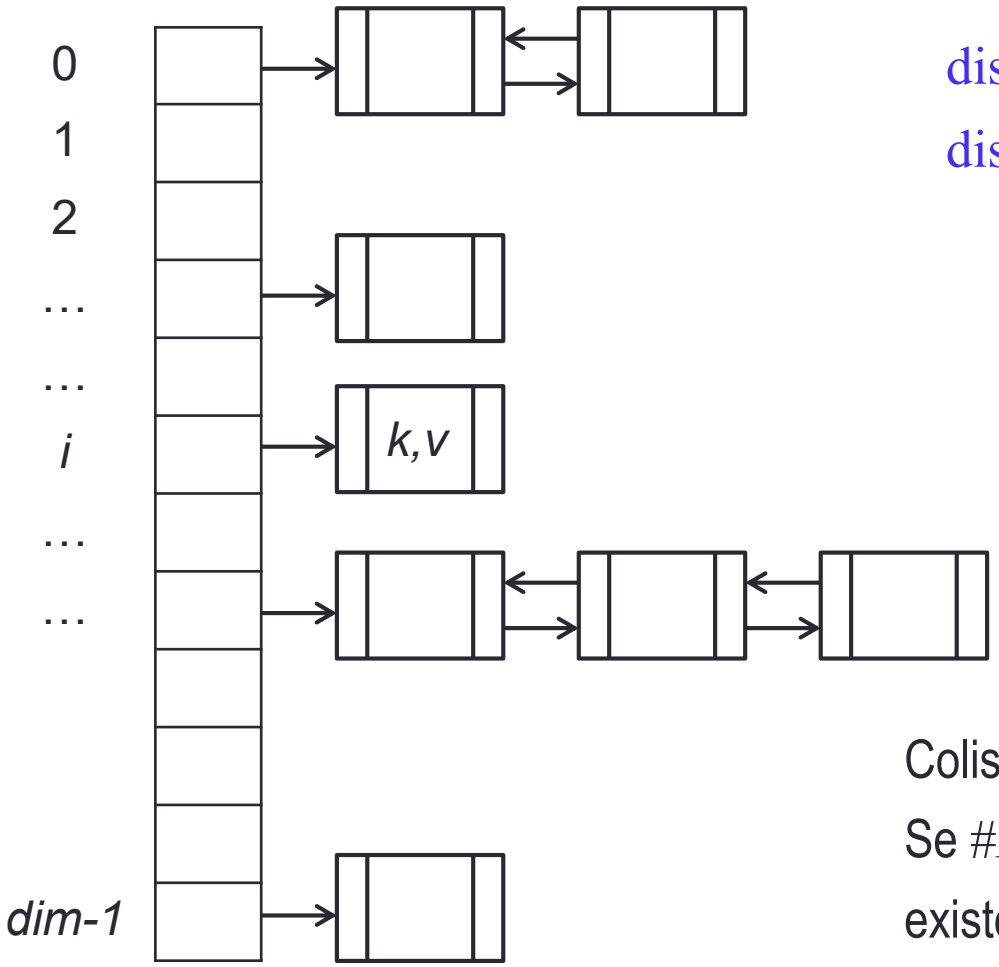
Quando se insere mais do que um objeto com a mesma dispersão, cria-se uma lista de colisões, no índice associado à dispersão

Tabela de dispersão aberta



- **`dim (11)`** – é a dimensão do vetor
- **K –** é o conjunto (domínio) das chaves que identificam os objetos a guardar
- ***n* (7) –** número de objetos guardados na tabela
- **$\lambda = n / dim$** deve ser sempre inferior a 1
- Complexidade ideal *O(1)*
- Se λ for inferior a 1 e a função de dispersão for apropriada, o preenchimento da tabela será esparso e a dimensão das listas curta.
- O acesso a um elemento estará perto de *O*(1)

Função de Dispersão



dispersão : $K \rightarrow \{ 0,1, 2, \ldots, dim\text{-}1\}$

dispersão (k) = i

Colisões:

Se $\#K > dim$,

existem $k1, k2 \in K$ tais que:

$k1 \neq k2$ e dispersão$(k1)$ = dispersão$(k2)$

Função de Dispersão

- Objetivos
  - Deve ser eficiente.
  - Deve distribuir as chaves uniformemente por todas as posições da tabela.
- Regras Práticas
  - A função de dispersão deve ser simples de calcular.
  - A dimensão da tabela (*dim*) deve ser um número primo.
  - Se as chaves forem grandes, deve-se considerar apenas uma parte, oriunda de vários pontos.

Cálculo da função de dispersão (2 passos)

- Passo 1: Cada chave sabe calcular o seu código de dispersão.

```
public int hashCode( );
```

- Passo 2: A tabela de dispersão converte o código de dispersão da chave num índice da tabela.

```
Math.abs( key.hashCode() ) % table.length
```

Exemplos de códigos de dispersão

- Se a chave é um número inteiro n o código será o próprio n.

- Se a chave é uma cadeia de caracteres $s_0 s_1 \ldots s_{n-1}$:
  - $s_0 + s_1 + \ldots + s_{n-1}$;
  - $(s_0 a^{n-1} + s_1 a^{n-2} + \ldots + s_{n-1})$ % b (onde a e b são primos)

Regra de Horner (1)

$$\text{códigoDisp}(s_0\, s_1 \ldots s_{n-1}) = (s_0\, a^{n-1} + s_1\, a^{n-2} + \ldots + s_{n-1}) \,\%\, b$$

$v \leftarrow 0;$

$v \leftarrow (v * a + s_0) \,\%\, b = (s_0) \,\%\, b$

$v \leftarrow (v * a + s_1) \,\%\, b = (s_0\, a + s_1) \,\%\, b$

$v \leftarrow (v * a + s_2) \,\%\, b = (s_0\, a^2 + s_1\, a + s_2) \,\%\, b$

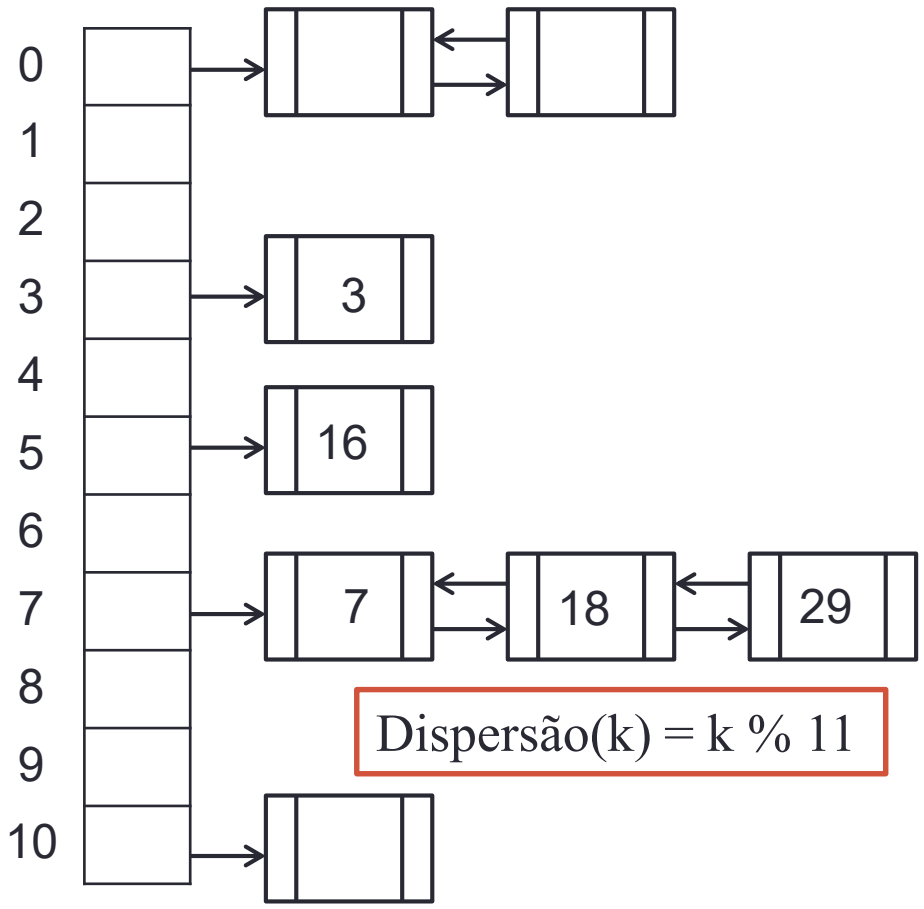
$v \leftarrow (v * a + s_3) \,\%\, b = (s_0\, a^3 + s_1\, a^2 + s_2\, a + s_3) \,\%\, b$

Regra de Horner (2)

$$\text{códigoDisp}(s_0 s_1 \ldots s_{n-1}) = (s_0 a^{n-1} + s_1 a^{n-2} + \ldots + s_{n-1}) \% b$$

```
public static int hash( String key ){

   int a = 127; // a is a prime number.
   int b = 2147483647; // b is a prime number.
   int hashCode = 0;

   for ( int i = 0; i < key.length(); i++ )
      hashCode = ( hashCode * a + key.charAt(i) ) % b;
   return hashCode;
}
```

Tabela de Dispersão Aberta (Separate Chaining)



- **Criar a TD:** Criar o vetor de listas ligadas (do tipo dicionário) e criar todas as listas de colisões (que podem ser simples ou duplas, ordenadas ou desordenadas).
- **Pesquisar k:** Calcular dispersão(k) e pesquisar k no dicionário associado à dispersão(k)
- **Inserir k,v:** Calcular dispersão(k), inserir k,v no dicionário associado à dispersão(k)
- **Remover k:** Calcular dispersão(k), remover k do dicionário associado à dispersão(k)

Fator de Ocupação da tabela de dispersão

- Sejam:
  - n - o número de entradas na tabela
  - dim – a dimensão da tabela
- Fator de Ocupação da tabela : $\lambda = \frac{n}{dim}$

(λé o comprimento médio das listas de colisões)

- λ deve ser sempre inferior a 1

Complexidades da Dispersão Aberta

Pesquisa	Melhor Caso	Pior Caso	Caso Esperado
Com sucesso	O(1)	O(n)	$O(1+\lambda)$
Sem sucesso	O(1)	O(n)	$O(1+\lambda)$

Classe Abstrata Tabela de Dispersão (1)

```
package dataStructures;

public abstract class HashTable<K,V> implements Dictionary<K,V>{

    // Default size of the hash table.
    public static final int DEFAULT_CAPACITY = 50;

    // Number of entries in the hash table.
    protected int currentSize;

    // Maximum number of entries.
    protected int maxSize;
```

Classe Abstrata Tabela de Dispersão (2)

```
// Public Static Methods

// Returns the hash code of the specified key,
// which is an integer in the range 0, ..., b-1.
public static int hash( String key ){

   int a = 127; // a is a prime number.
   int b = 2147483647; // b is a prime number.
   int hashCode = 0;

   for ( int i = 0; i < key.length(); i++ )
      hashCode = ( hashCode * a + key.charAt(i) ) % b;
   return hashCode;
}
```

Classe Abstrata Tabela de Dispersão (3)

```
// Protected Static Methods

// Returns a prime number that is not less than the
// specified number; or zero if all such primes are greater
// than Integer.MAX VALUE.
protected static int nextPrime( int number ){
   for ( int i = 0; i < PRIMES.length; i++ )
      if ( PRIMES[i] >= number )
         return PRIMES[i];
      return 0;
}

protected static final int[] PRIMES =
   { 11, 19, 31, 47, 73, …, 1431655751, 2147483647 };
```

Classe Abstrata Tabela de Dispersão (4)

```
// Public Instance Methods

// Returns true iff the dictionary contains no entries.

public boolean isEmpty( ){
   return currentSize == 0;
}

// Returns the number of entries in the dictionary.
public int size( ){
   return currentSize;
}
```

Classe Abstrata Tabela de Dispersão (5)

```
// If there is an entry in the dictionary whose key is the specified
// key, returns its value; otherwise, returns null.
public abstract V find( K key );

// If there is an entry in the dictionary whose key is the specified
// key, replaces its value by the specified value and returns the old
// value; otherwise, inserts the entry (key, value) and returns null.
public abstract V insert( K key, V value );

// If there is an entry in the dictionary whose key is the specified
// key, removes it from the dictionary and returns its value;
// otherwise, returns null.
public abstract V remove( K key );

// Returns an iterator of the entries in the dictionary.
public abstract Iterator<Entry<K,V>> iterator( );
```

Classe Abstrata Tabela de Dispersão (6)

```
    // Protected Instance Methods

    // Returns true iff the hash table cannot contain more entries.
    protected boolean isFull( ){
        return currentSize == maxSize;
    }

} // End of HashTable.
```

Classe Tabela de Dispersão Aberta (1)

```
package dataStructures;
   public class ChainedHashTable<K extends Comparable<K>, V>
      extends HashTable<K,V>{

      // The array of dictionaries.
      protected Dictionary<K,V>[] table;

      public ChainedHashTable( ){
         this(DEFAULT_CAPACITY);
      }
```

Em cada posição do vetor existirá uma ED que implementa o interface dicionário, para apoiar a implementação da Tabela de Dispersão

Classe Tabela de Dispersão Aberta (2)

```
@SuppressWarnings("unchecked")
public ChainedHashTable( int capacity ){

   int arraySize = HashTable.nextPrime((int) (1.1 * capacity));

   // Compiler gives a warning.
   table = (Dictionary<K,V>[]) new Dictionary[arraySize];

   for ( int i = 0; i < arraySize; i++ )
      table[i] = new OrderedDoubleList<K,V>();
   maxSize = capacity;
   currentSize = 0;
}
```

- A implementar de raiz na 2ª fase do trabalho: Uma lista duplamente ligada que implementa o interface dicionário.
- A inserção, remoção e pesquisa nesta lista terão em conta a ordenação definida por K, tipo comparável

Classe Tabela de Dispersão Aberta (3)

```
// Returns the hash value of the specified key.
protected int hash( K key ){
   return Math.abs( key.hashCode() ) % table.length;
}


// If there is an entry in the dictionary whose key is the
// specified key, returns its value; otherwise, returns null.
public V find( K key ){
   return table[ this.hash(key) ].find(key);
}
```

Determina índice da Tabela de dispersão (do vetor)

Efetua pesquisa pela chave key (tipo K) na lista de colisões existente no índice

Classe Tabela de Dispersão Aberta (4)

```
   // If there is an entry in the dictionary whose key is the
   // specified key, replaces its value by the specified value and
   // returns the old value; otherwise, inserts the entry
   // (key, value) and returns null.
   public V insert( K key, V value ){
      if ( this.isFull() )
         this.rehash();

      // Efectua-se a inserção.
      ..........................................
   }
   ..............................................
} // End of ChainedHashTable (incomplete).
```

Falta implementar ainda método de remoção e classe iterador da Tabela de Dispersão

Implementação TAD Biblioteca

Dicionário de Documentos em Tabela de Dispersão Aberta

LibraryClass - incompleta(1)

```
package library;
import dataStructures.*;

public class LibraryClass implements Library {

   // Library documents repository
   private Dictionary<String,Document> documents;

   // Library document returns
   private CopyReturns returns;
        ...
   public LibraryClass(int capacity){
      documents=new ChainedHashTable<String,Document>(capacity);
      returns=new CopyReturnsClass();
                  ...
   }
```

LibraryClass - incompleta(2)

```
// Add new Book to library
// Requires: documentCode does not identify an existing document
public void addNewBook(String title, String subject, String documentCode,
                       String publisher, String author, long ISBN)
                       throws ExistingDocException{
   Document d;
   if (documents.find(documentCode) != null)
     throw new ExistingDocException();
   else {
         d = new BookClass(title, subject, documentCode, publisher,
                           author, ISBN);
         documents.insert(documentCode, d);
        }
}
```

LibraryClass - incompleta(3)

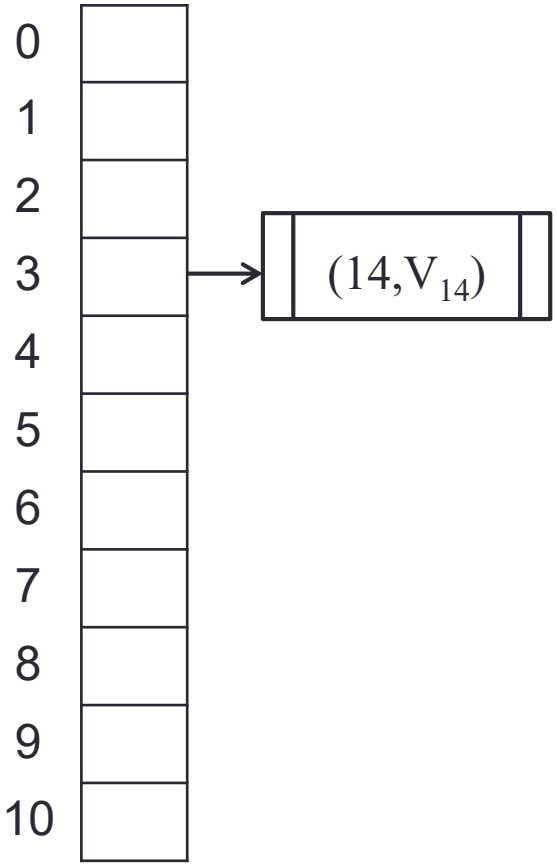
```
// Removes document identified by documentCode from library and returns
// removed document.
//Requires: documentCode identifies document in library
public Document removeDocument(String documentCode)
   throws NonExistingDocException{
   Document d = documents.remove(documentCode);
   if (d == null)
              throw new NonExistingDocException();
   else return d;
   }

// Returns document identified by documentCode from library
// Requires: documentCode identifies document in library
 public Document getDocument(String documentCode)
   throws NonExistingDocException{

   Document d = documents.find(documentCode);
   if (d == null) throw new NonExistingDocException();
   else
      return d;
}
```

Tabela de Dispersão Aberta - exemplo

Dispersão(k) = k % 11

1. Inserir $(14, V_{14})$
   Dispersão(14) = 3

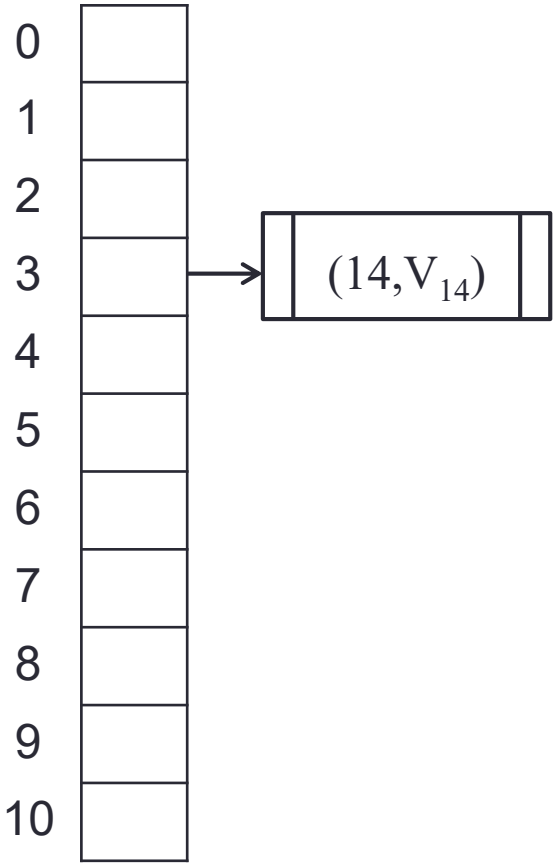
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(14,V_{14})$
   Dispersão(14) = 3

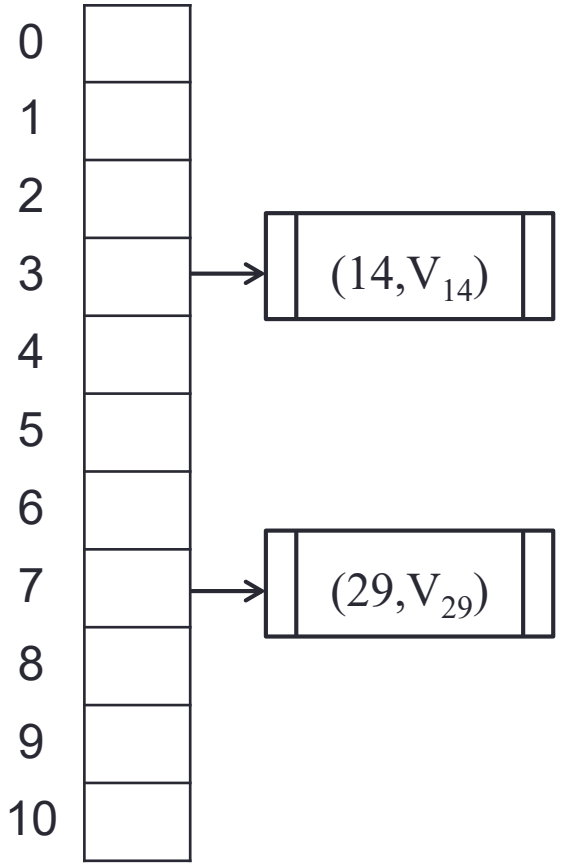
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(29,V_{29})$
   Dispersão(29) = 7

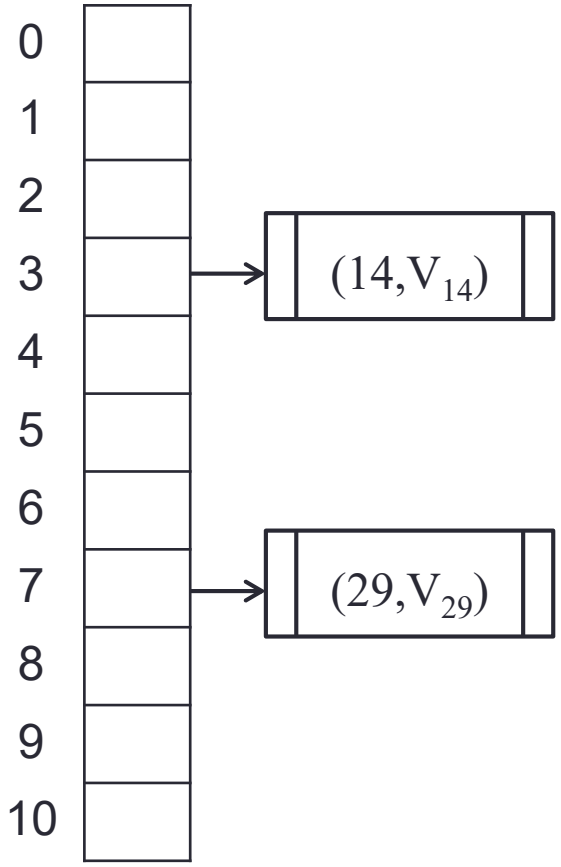
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(29,V_{29})$
   Dispersão(29) = 7

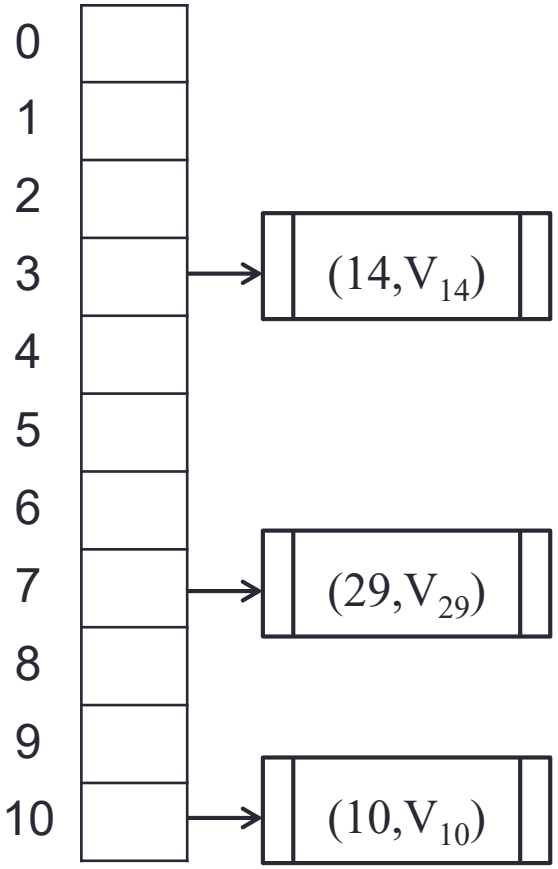
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(10,V_{10})$
   Dispersão(10) = 10

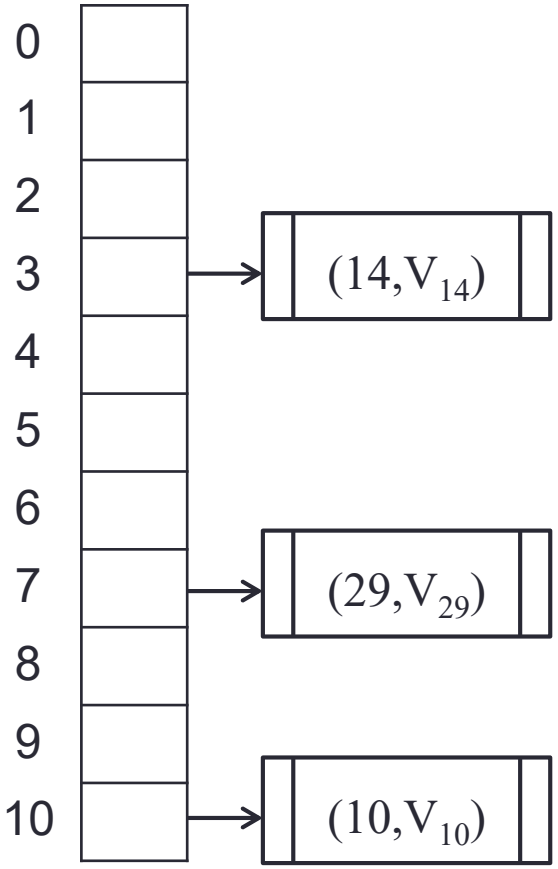
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(10,V_{10})$
   Dispersão(10) = 10

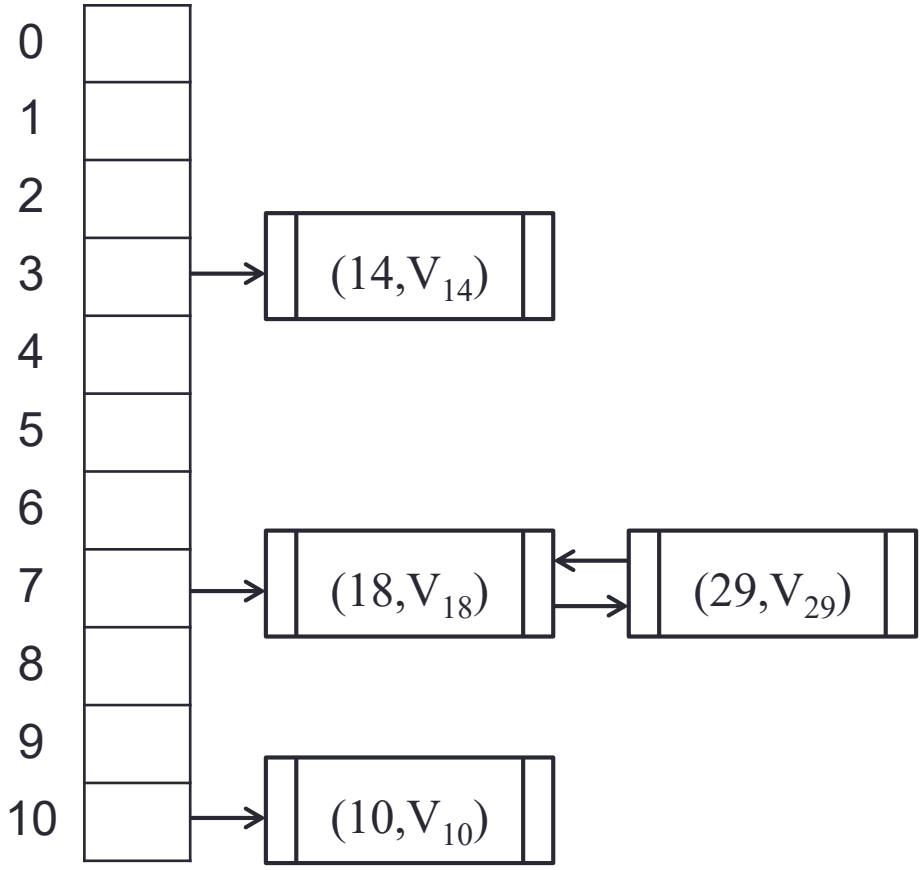
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(18,V_{18})$
   Dispersão(18) = 7

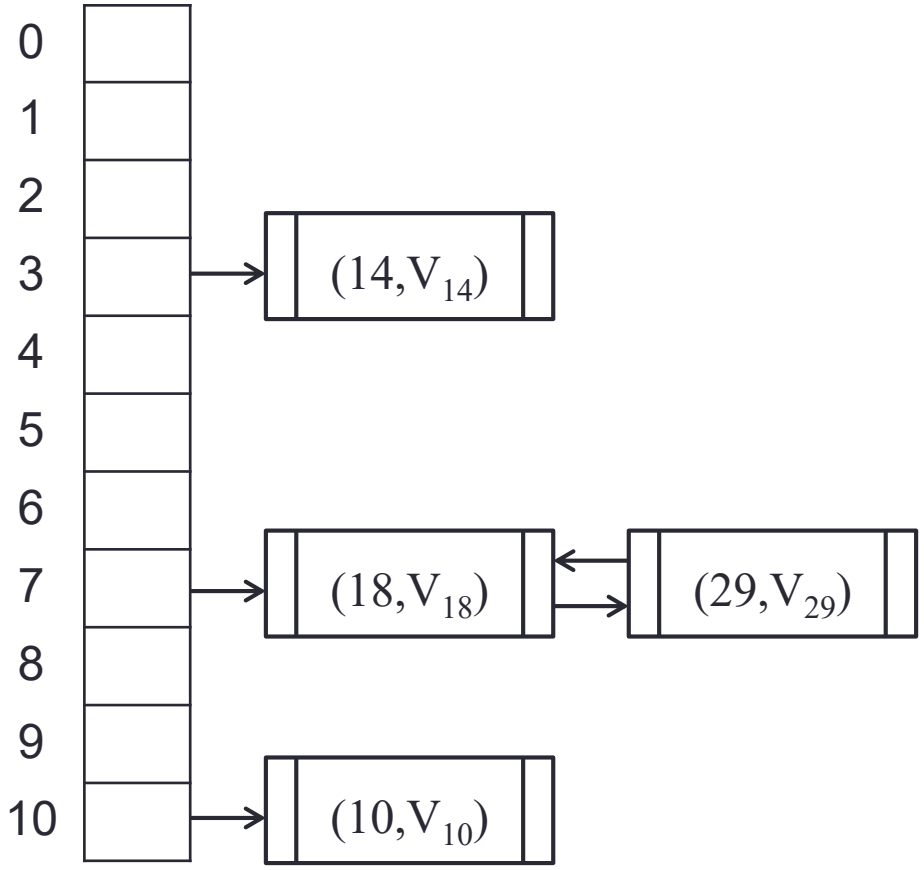
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(18,V_{18})$
   Dispersão(18) = 7

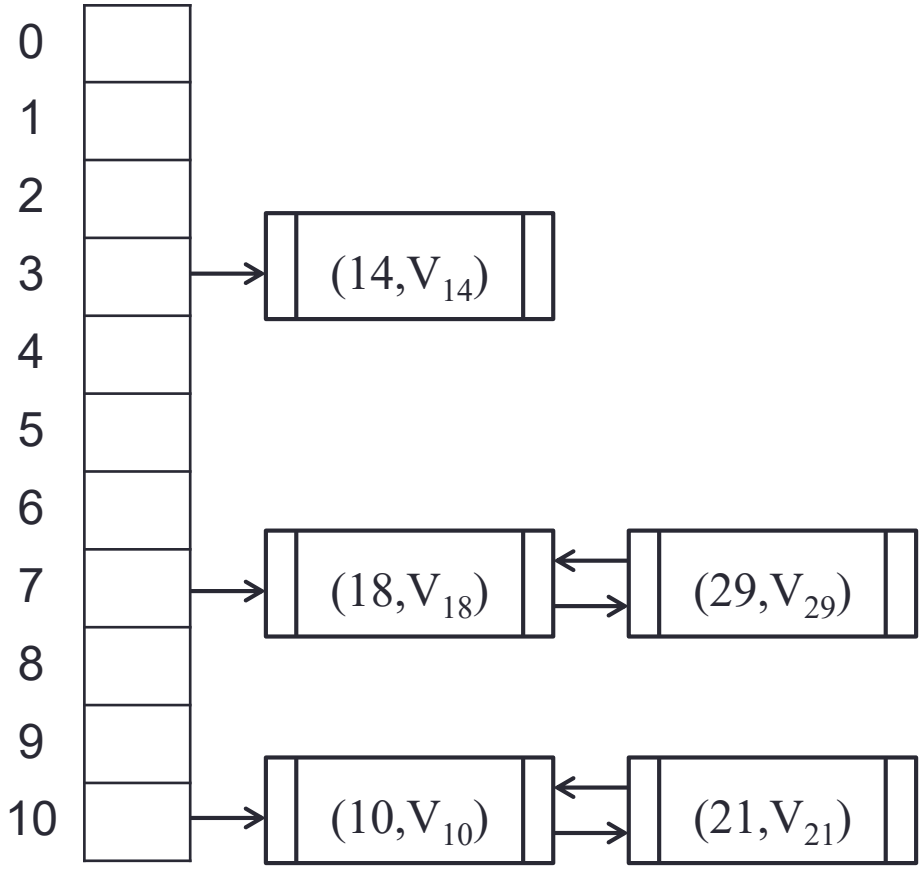
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(21,V_{21})$
   Dispersão(21) = 10

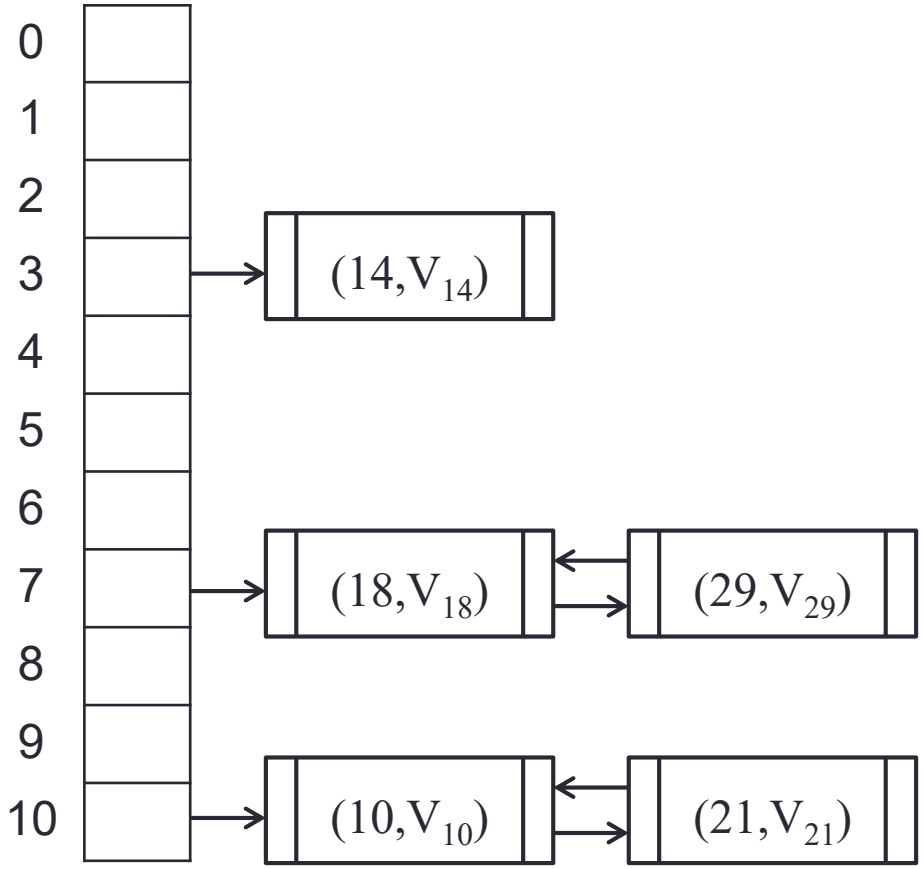
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Inserir $(21,V_{21})$
   Dispersão(21) = 10

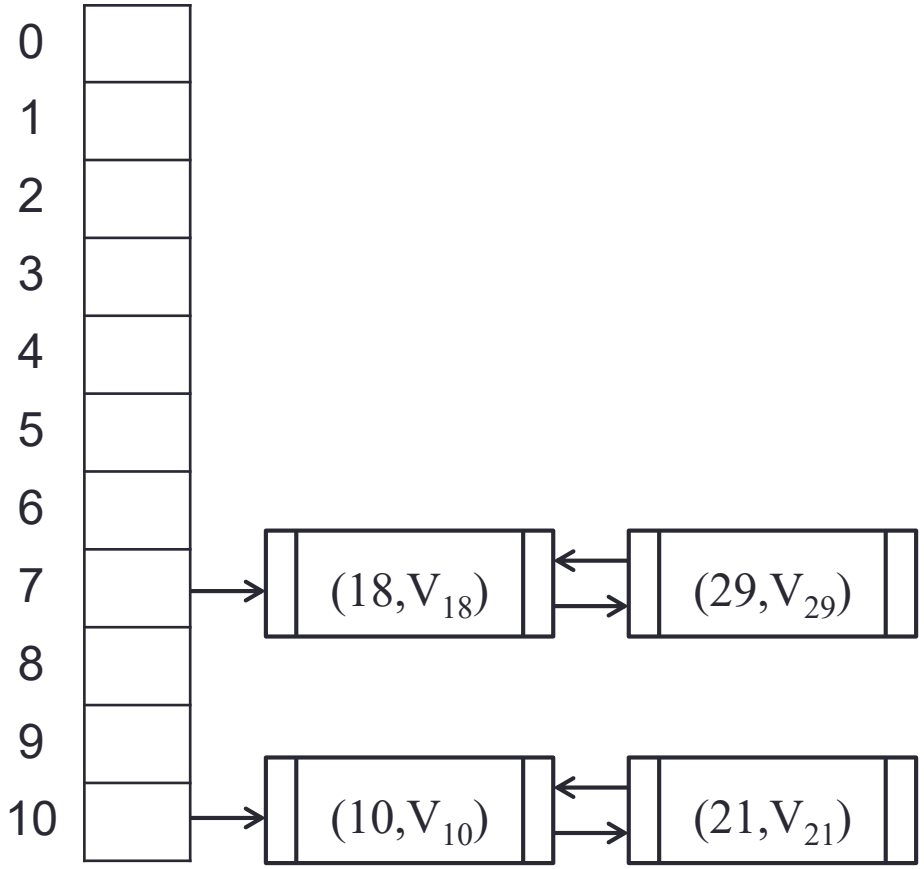
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Remover (14)
   Dispersão(14) = 3

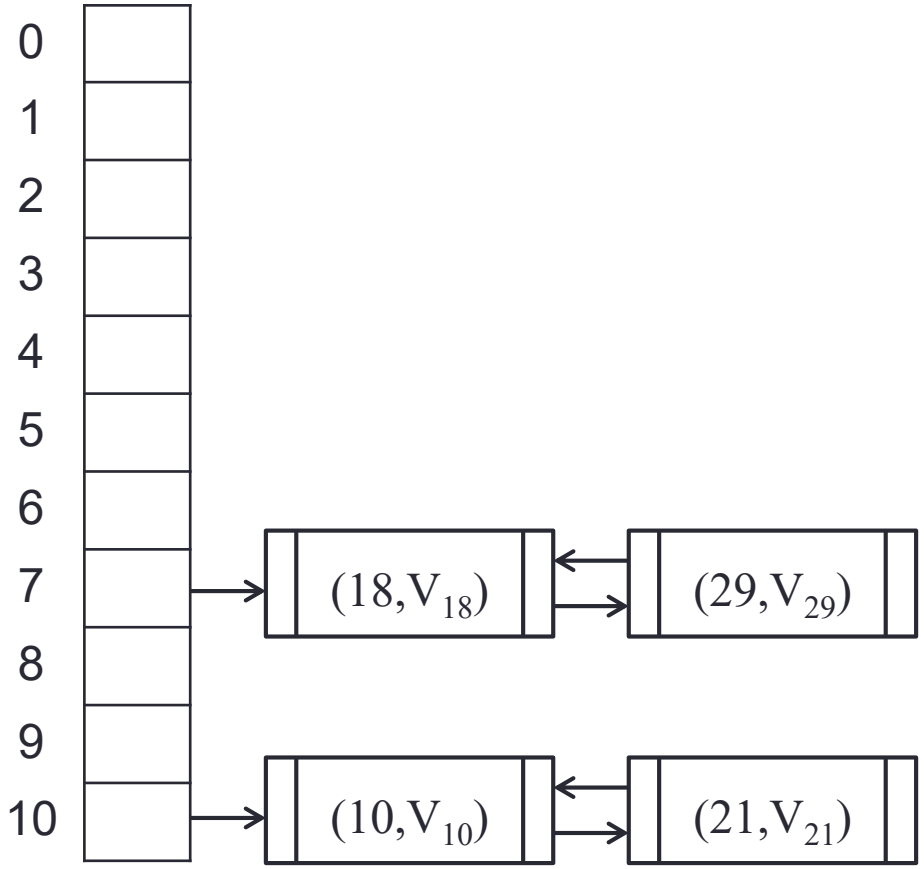
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Remover (14)
   Dispersão(14) = 3

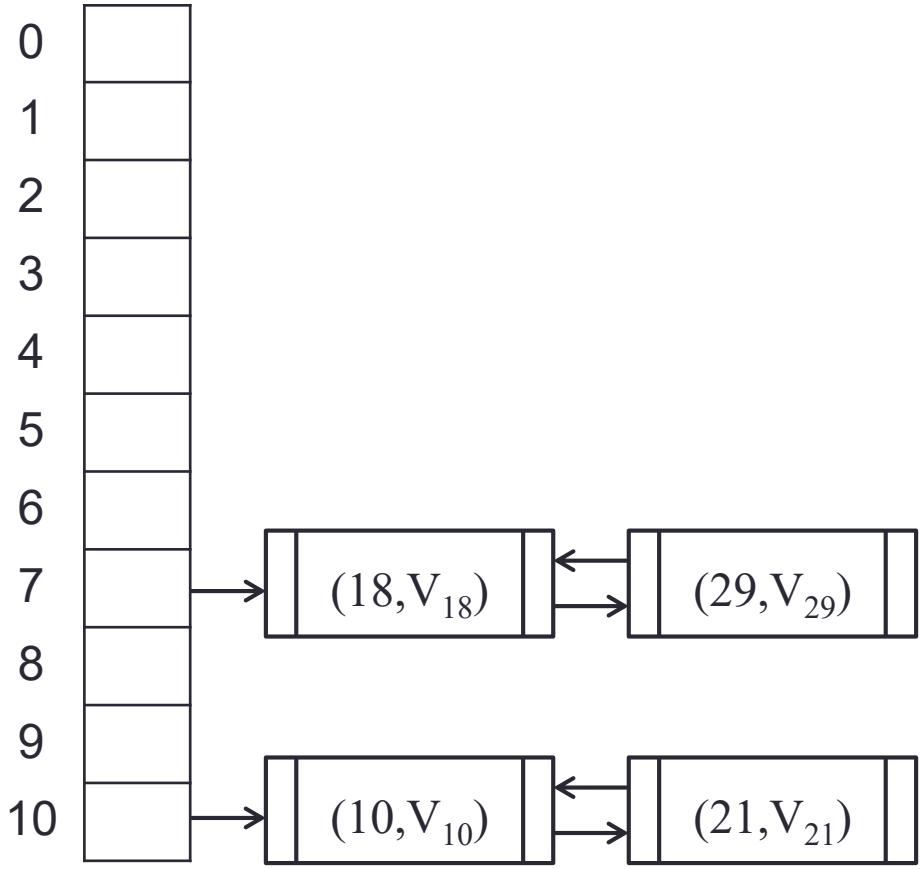
Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Remover (7)
   Dispersão(7) = 7

Tabela de Dispersão Aberta - exemplo



Dispersão(k) = k % 11

1. Remover (40)
   Dispersão(40) = 7

Dispersão Fechada (Open Addressing)

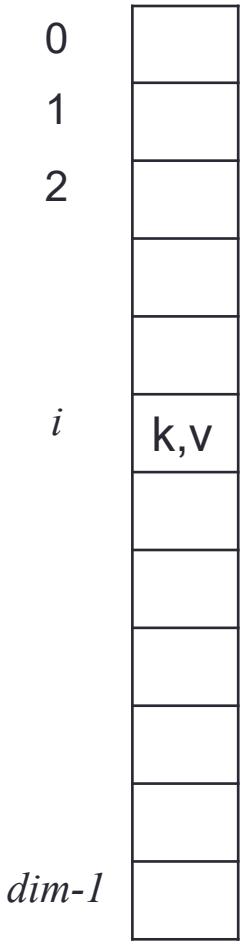


Tabela de Dispersão constituída apenas pelas posições contidas no vetor

$$\text{dispersão} : K \rightarrow \{ 0,1, 2, \ldots, dim\text{-}1\}$$

$$\text{dispersão}\ (k) = \text{i}$$

$$\text{dispersão}\ (k') = \text{i}$$

Na pesquisa da chave k', quando a posição da tabela indicada pela função de dispersão - dispersão(k') - já está ocupada com uma entrada cuja chave é diferente de k', recorre-se a uma segunda função - a função de sondagem - que indica outra posição da tabela.

Visitam-se as posições $p_0(k'), p_1(k'), p_2(k'), \ldots$ tais que:

$p_j(k') = (\ \text{dispersão}(k') + \text{sondagem}(j)\ )\ \%\ dim$

Tendo-se sempre sondagem(0) = 0

Sondagem Linear (Linear Probing)

Neste método a função de sondagem é representada por:

$$\text{sondagem}(j) = j$$

$$\text{sondagem}(0) = 0 \qquad p_0 = (\ \text{dispersão}(k)+0\ )\ \%\ dim$$
$$\text{sondagem}(1) = 1 \qquad p_1 = (\ \text{dispersão}(k)+1\ )\ \%\ dim$$
$$\text{sondagem}(2) = 2 \qquad p_2 = (\ \text{dispersão}(k)+2\ )\ \%\ dim$$
$$\text{sondagem}(3) = 3 \qquad p_3 = (\ \text{dispersão}(k)+3\ )\ \%\ dim$$

Primeira Posição: pos ← dispersão(k)

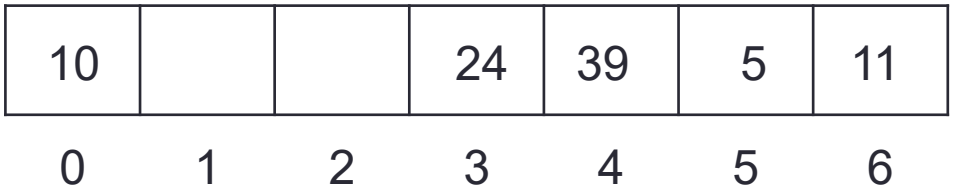
Posição Seguinte: pos ← (pos + 1) % dim

Sondagem Linear - exemplo

$dim = 7$

dispersão(n) = n % 7

1. dispersão(5) = 5
2. dispersão(24) = 3
3. dispersão(39) = 4
4. dispersão(11) = 4
5. dispersão(10) = 3



Sondagem Linear (Só pesquisas e inserções) (1)

```
// table is an array of entries.

// Returns the position where the entry with the specified key is,
// or the first empty position found, if no such entry exists.
protected int findPos( K key ){

   int pos = this.hash(key);

   while ( table[pos] != null && !table[pos].getKey().equals(key) )
      pos = ( pos + 1 ) % table.length;
   return pos;
}
```

Sondagem Linear (Só pesquisas e inserções) (2)

```
// If there is an entry in the dictionary whose key is the
// specified key, returns its value; otherwise, returns null.
public V find( K key ){

   int pos = this.findPos(key);

   if ( table[pos] == null )
      return null;
   else
      return table[pos].getValue();
}
```

Problemas da sondagem linear

- Existência de blocos contíguos de posições ocupadas de grandes dimensões (em geral, quando $\lambda \geq 0.5$).
- A sondagem é igual para chaves que colidem.

Dispersão Dupla (Double Hashing)

Neste método a função de sondagem é representada por:

$$\text{sondagem}(j) = j * \text{dispersão2}(k)$$

O incremento varia com a chave, é definido pela 2ª função de dispersão

$$p_0 = (\ \text{dispersão}(k)+0\ )\ \%\ dim$$
$$p_1 = (\ \text{dispersão}(k)+\text{dispersão2}(k)\ )\ \%\ dim$$
$$p_2 = (\ \text{dispersão}(k)+2\ \text{dispersão2}(k))\ \%\ dim$$
$$p_3 = (\ \text{dispersão}(k)+3\ \text{dispersão2}(k))\ \%\ dim$$

Primeira Posição: pos ← dispersão(k); inc ← dispersão2(k)

Posição Seguinte: pos ← (pos + inc) % *dim*

Dispersão Dupla - Segunda função de dispersão

- **Objetivos** da 2ª função de dispersão:
  - Não pode retornar zero.
  - Deve retornar números que sejam primos com dim.
  - Se dim for primo, basta garantir que os valores retornados variam entre 1 e dim - 1.

Dispersão fechada - Espaço

- Para garantir que o ciclo de pesquisa termina, é necessário assegurar que existem posições vazias.

Método de resolução de Colisões	λ ideal	λ máximo
Sondagem Linear	0.5	0.8
Dispersão Dupla	0.5	0.8

- Deve-se pré-dimensionar a tabela tendo em vista o fator de ocupação ideal, permitindo inserções apenas enquanto não for atingido o fator de ocupação máximo.

Dimensionamento da Dispersão Fechada Exemplo

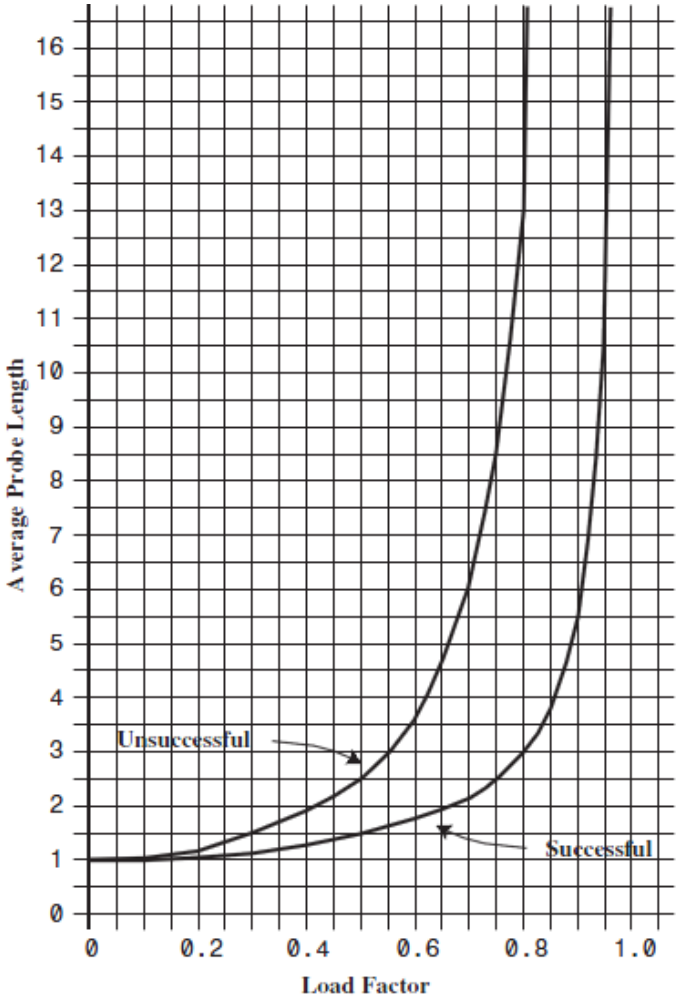
- Se o número esperado de entradas for 50
  - Dimensão da Tabela – 101 (número primo a seguir a 50/0.5)
  - Aceita-se a inserção de, no máximo, 80 entradas (0.8 * 101 = 80.8).

- Na inserção de 81ª entrada:
  - Efetua-se a redispersão – inserem-se as 80 entradas existentes numa tabela maior (que, em geral, tem o dobro da capacidade), e
  - Insere-se a nova entrada na nova tabela.

Complexidades da Sondagem Linear

Pesquisa	Melhor Caso	Pior Caso	Caso Esperado
Com sucesso	1	*n*	$\frac{1}{2}(1+\frac{1}{1-\lambda})$
Sem Sucesso	1	*n*	$\frac{1}{2}(1+\frac{1}{(1-\lambda)^2})$

Sendo *n* o número de entradas na tabela



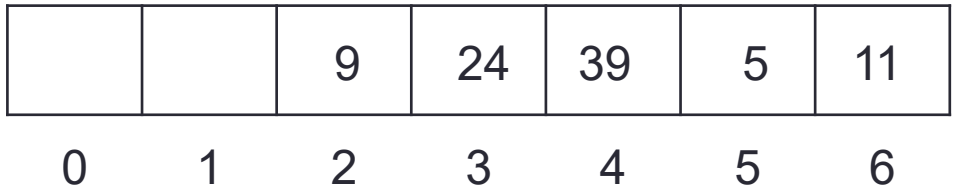
Fator de Ocupação

Pesquisa (caso esperado)	0.25	0.5	0.75	0.8	0.9
Com sucesso	1.2	1.5	2.5	3.0	5.5
Sem Sucesso	1.4	2.5	8.5	13.0	50.5

Remoção - Sondagem Linear

$dim = 7$ dispersão$(n) = n \% 7$

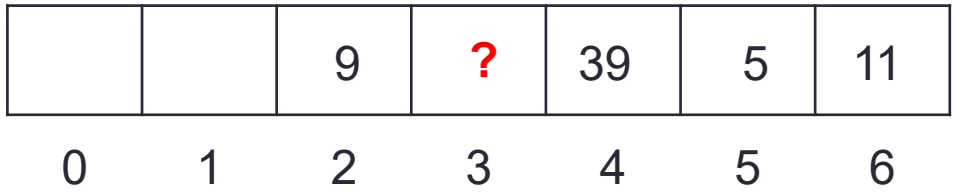
1.Remover(24)



Remoção - Sondagem Linear

$dim = 7$ dispersão$(n) = n \% 7$

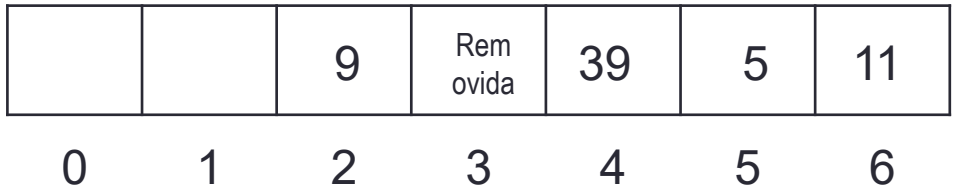
1.Remover(24)



Remoção - Sondagem Linear

$dim = 7$ dispersão$(n) = n \% 7$

1. Remover(24)



Remoção - Sondagem Linear

$dim = 7$

dispersão$(n) = n \% 7$

1. Remover(24)
2. Inserção 72, V
   dispersão(72) = 2

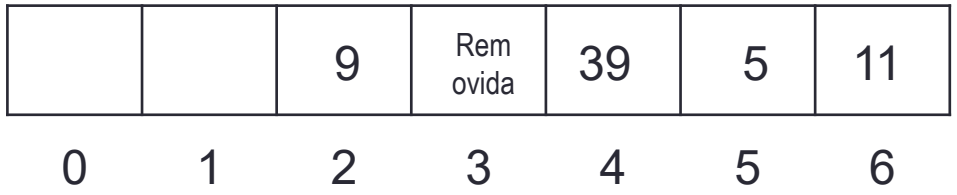
		9	Rem ovida	39	5	11
0	1	2	3	4	5	6

Remoção - Sondagem Linear

$dim = 7$

dispersão$(n) = n \% 7$

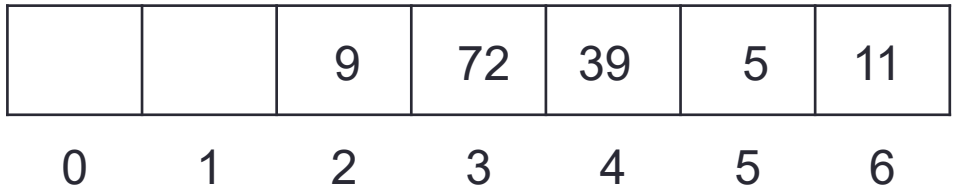
1. Remover(24)
2. Inserção 72, V
   dispersão(72) = 2



Remoção - Sondagem Linear

$dim = 7$ dispersão$(n) = n \% 7$

1. Remover(24)
2. Inserção 72, V
   dispersão(72) = 2



E se a chave 72 já existisse na tabela ?

- Tenho sempre de avançar no vetor até encontrar uma posição vazia, ou até encontrar a chave
- No entanto, a inserção, caso seja possível, deve ser feita na posição em estado "removida"

Sondagem linear com remoção(1)

- Estados de uma posição:
  - **Vazia** – Nunca foi preenchida
  - **Preenchida** – contém uma entrada
  - **Removida** – já conteve uma entrada, que foi removida e ainda não voltou a ser preenchida
- Pesquisar k: o ciclo de pesquisa termina quando se encontra k ou uma posição **Vazia**
- Remover k: o ciclo de pesquisa da chave para remoção termina quando se encontra k ou uma posição **Vazia**
  - Se se encontra k: Assinala-se a posição onde está k como sendo **Removida**

Remoção na sondagem linear (2)

- Inserir k,v: envolve dois ciclos de pesquisa antes de fazer a inserção
  - O **primeiro ciclo** termina quando se encontra k, uma posição **Vazia** ou uma posição **Removida**
    - **Vazia**: Insere-se k,v nessa posição
    - **Removida**: Guarda-se essa posição (`posição-se-inserir`) e executa-se o segundo ciclo
  - O segundo ciclo termina quando se encontra K ou uma posição **Vazia**
    - **Vazia**: Insere-se k,v em `posição-se-inserir`
- Este algoritmo resolve o problema, mas ao fim de algum tempo, a tabela terá várias posições **Removida**, o que vai afetar a performance
- Quando a remoção é importante, deve-se usar Dispersão Aberta

Comparação de tipos de dispersão

- Vantagens da Dispersão Aberta
  - Suporta a operação de remoção
  - Os diferentes conjuntos de chaves que colidem (as listas de colisões) não se misturam

- Vantagens da Dispersão Fechada
  - Gasta menos memória (alguns bytes por entrada)

Vantagens da Dispersão

- Caso esperado: complexidades de pesquisa e inserção (e de remoção na DA) são constantes

- A técnica é eficiente e “só” depende do fator de ocupação e da qualidade das funções (dispersão, sondagem (na DF))

Problemas da Dispersão

- Não é uma estrutura dinâmica
- Não suporta operações que se baseiem na relação de ordem entre as chaves: mínimo, máximo, percurso ordenado
- Pior Caso: As complexidades das operações de pesquisa e inserção (e de remoção, na DA) são lineares no número de entradas na tabela