Aula 20

Tratamento de colisões em Tabelas de dispersão

Algoritmos e Estruturas de Dados

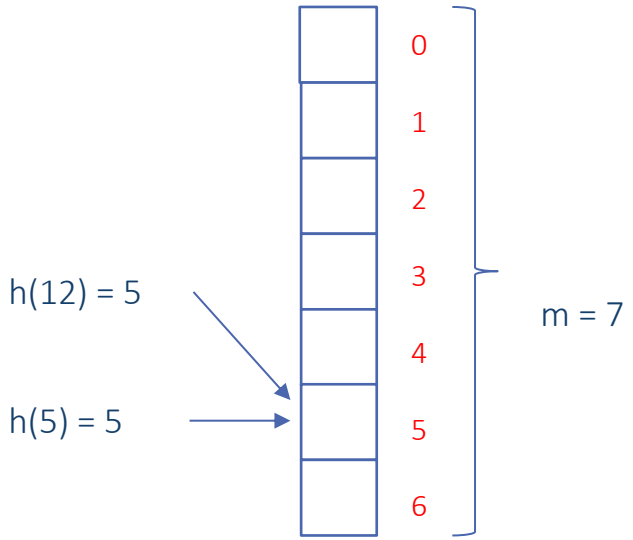
Tratamento de colisões

- O que fazer quando duas chaves diferentes têm o mesmo valor de hash?
- Tratamento de colisões
  - Várias soluções
  - Tabelas de encadeamento separado

    Separate chaining
  - Tabelas de endereçamento aberto

    Linear probing

    Double hashing



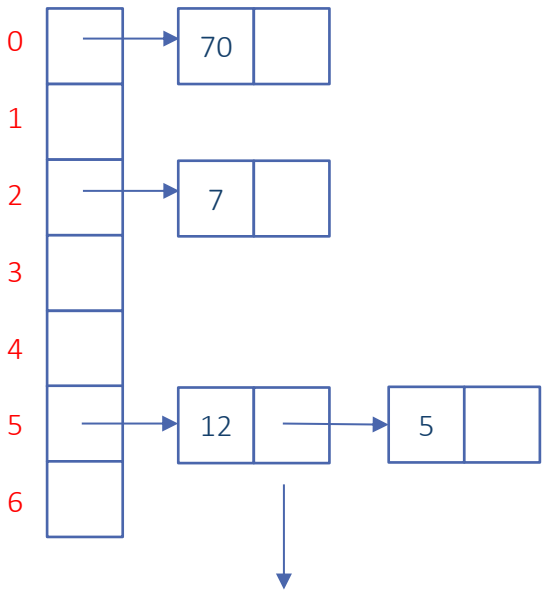
Tabelas de encadeamento separado

Separate chaining

Tabelas de encadeamento separado

- Guardar chaves com mesmo valor de hash dentro do mesmo balde
  - Usar uma lista ligada
  - Ou um array
  - **Chaves dentro de cada encadeamento não estão ordenadas!**

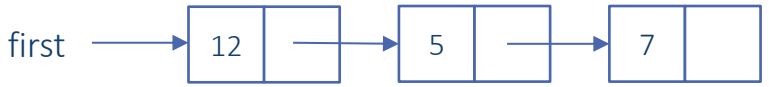
    Alguém sabe explicar pq?
  - Encadeamentos podem ser implementados como Lista de Pesquisa Sequencial



Cada balde tem o seu próprio encadeamento que pode conter várias chaves

Lista de Pesquisa Sequencial

```
public class SequentialSearchList<Key,Value> {

    private class Node{
        Key key;
        Value value;
        Node next;

        public Node(Key k, Value v, Node next)
        {
            this.key = k;
            this.value = v;
            this.next = next;
        }
    }

    private Node first;
    private int size = 0;
```

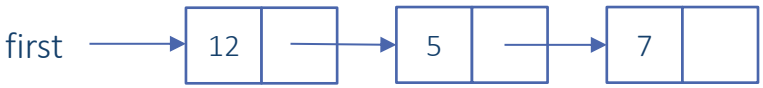


Para facilitar a legibilidade, apenas é mostrada a chave em cada nó

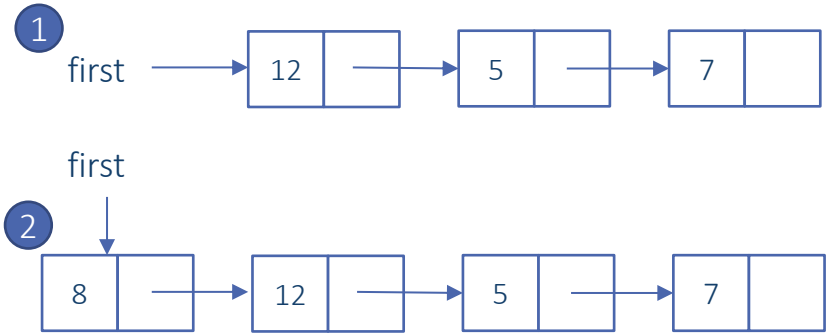
Lista de Pesquisa Sequencial

```
public Value get(Key key)
{
    Node n = this.first;
    while(n != null)
    {
        if(key.equals(n.key)) return n.value;
        n = n.next;
    }
    return null;
}
```

get(5)



put(8,10)

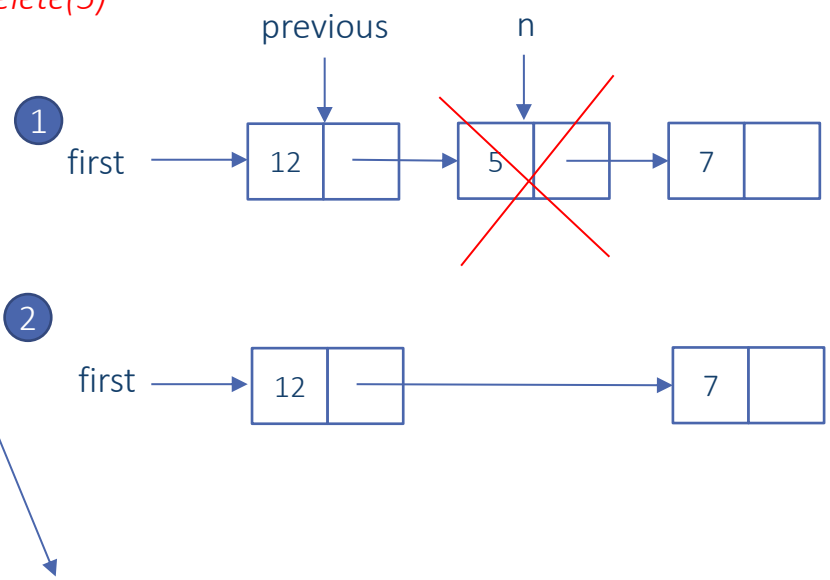



```
public void put(Key key, Value value)
{
    Node n = this.first;
    while(n != null)
    {
        if(key.equals(n.key))
        {
            //this is an update of an existing key
            n.value = value;
            return;
        }
        n = n.next;
    }
    //key does not exist, create a new one
    this.first = new Node(key,value,this.first);
    this.size++;
}
```

Lista de Pesquisa Sequencial

```
public void delete(Key key)
{
    if(this.first == null) return;
    if(key.equals(this.first.key))
    {
        this.first = this.first.next;
        this.size--;
        return;
    }

    Node previous = this.first;
    Node n = this.first.next;
    while(n != null)
    {
        if(key.equals(n.key))
        {
            previous.next = n.next;
            size--;
            return;
        }
        previous = n;
        n = n.next;
    }
}
```

delete(5)



caso especial, o primeiro nó contém a chave a ser removida

Tabela de encadeamento separado

```
public class SeparateChainingHashTable<Key, Value> {

    private static final int DEFAULT_M = 997;

    private int m;
    private int size;
    private SequentialSearchList<Key,Value>[] buckets;

    public SeparateChainingHashTable()
    {
        this(DEFAULT_M);
    }

    @SuppressWarnings("unchecked")
    public SeparateChainingHashTable(int tableSize)
    {
        this.m = tableSize;
        this.buckets = (SequentialSearchList<Key,Value>[]) new SequentialSearchList[this.m];
    }
```

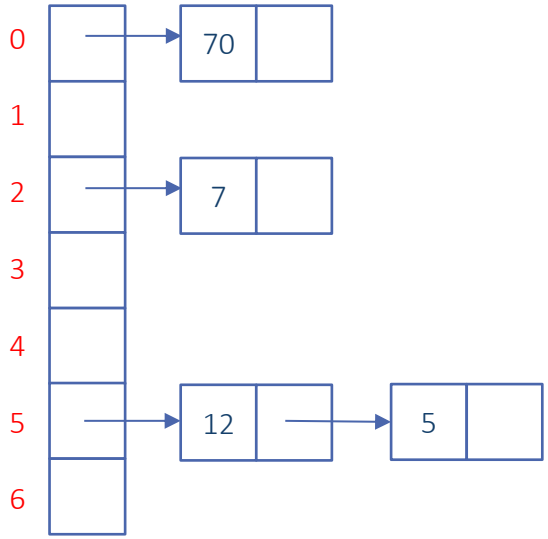


Tabela de encadeamento separado

```
private int hash(Key k)
{
    return (k.hashCode() & 0x7fffffff) % this.m;
}

public Value get(Key k)
{
    return this.buckets[hash(k)].get(k);
}

public void put(Key k, Value v)
{
     SequentialSearchList<Key,Value> l = this.buckets[hash(k)];
     int initialSize = l.size();
     l.put(k,v);
     this.size += l.size() - initialSize;
}
```

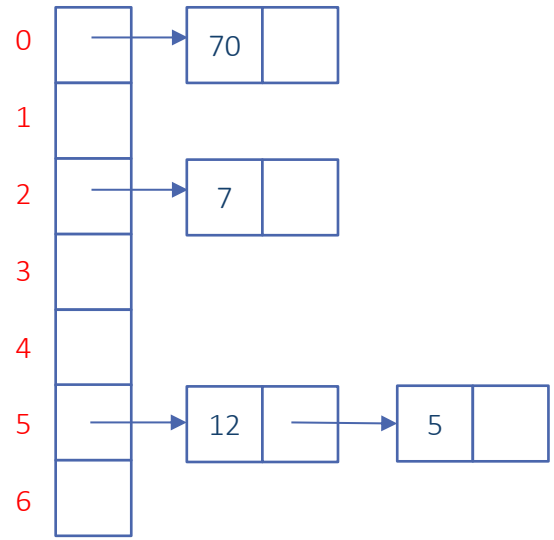
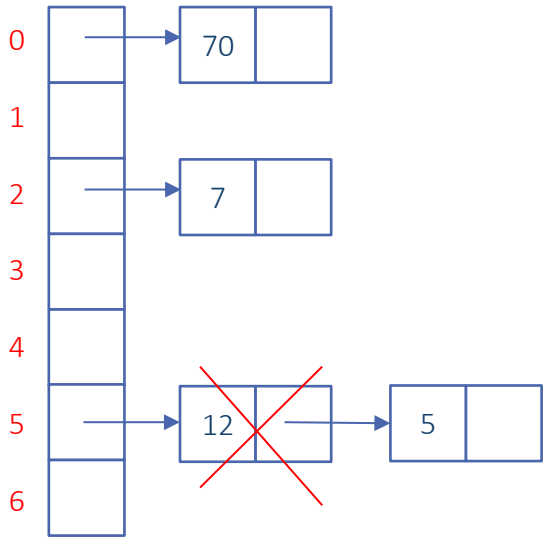


Tabela de encadeamento separado

```
public void delete(Key k)
{
    SequentialSearchList<Key,Value> l = this.buckets[hash(k)];
    int initialSize = l.size();
    l.delete(k);
    this.size += l.size() - initialSize;
}
```

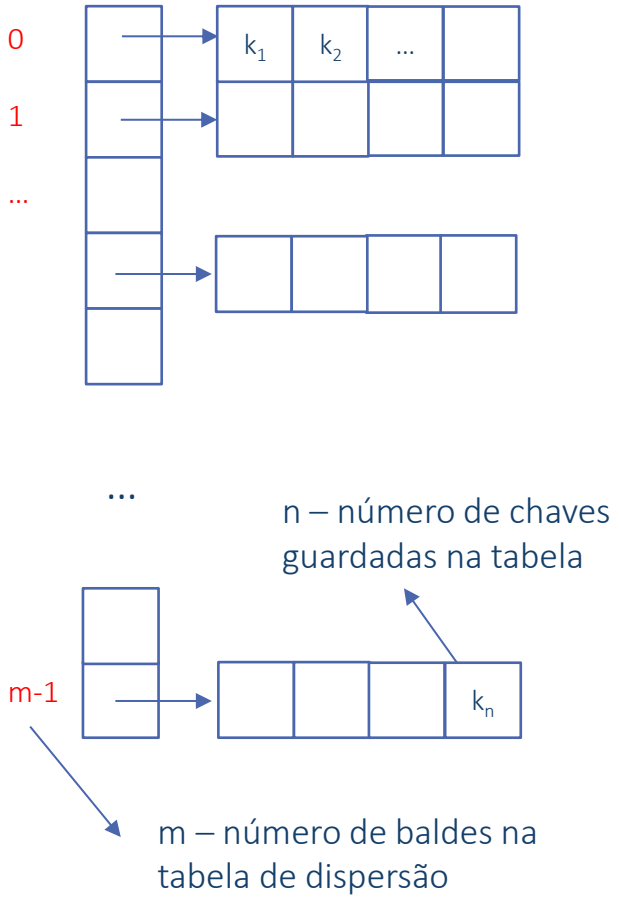


Complexidade Temporal

- Assumindo que a função de hash distribui as chaves pelos índices de forma uniforme:

O número de chaves em cada encadeamento é aproximadamente $\frac{n}{m}$ com probabilidade ~1

Exemplo:
Para m = $\frac{n}{4}$
Número médio de chaves por enquadeamento = $\frac{n}{\frac{n}{4}} = 4$



Complexidade Temporal

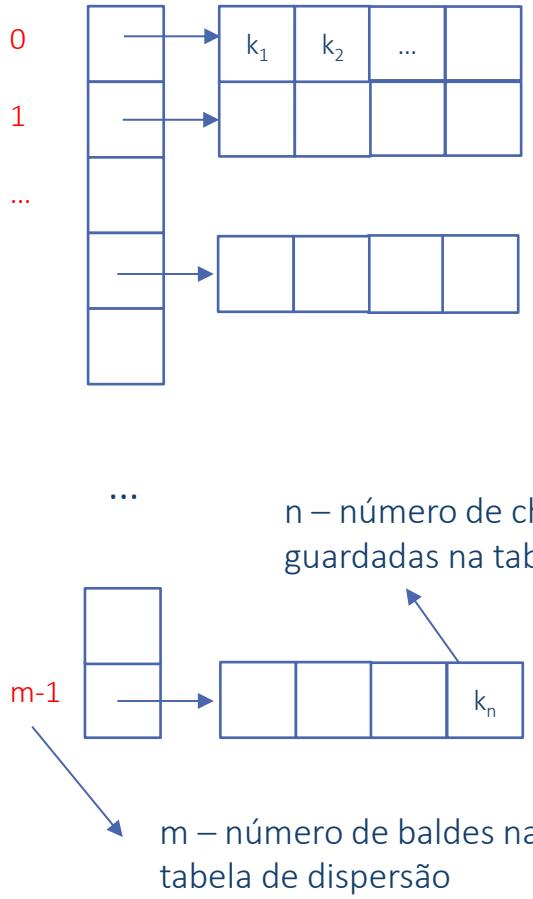
- operações de pesquisa e inserção numa tabela de encadeamento separado
  - 1) usar a função de hash para determinar o balde

    ~1
  - 2) procurar a chave no encadeamento

    Número de comparações:

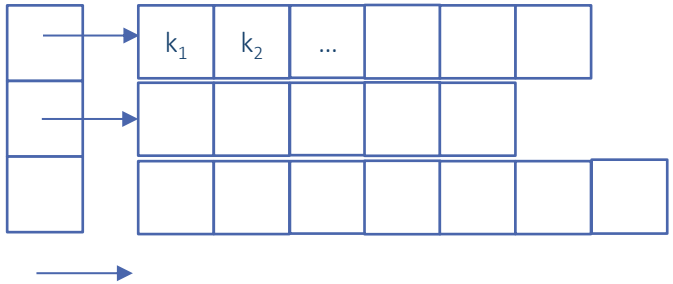
    pior caso - $\sim \frac{n}{m}$

    Caso médio $\frac{1+2+\cdots+\frac{n}{m}}{\frac{n}{m}} = \frac{\frac{\frac{n}{m}}{2}(1+\frac{n}{m})}{\frac{n}{m}} = \frac{1+\frac{n}{m}}{2}$

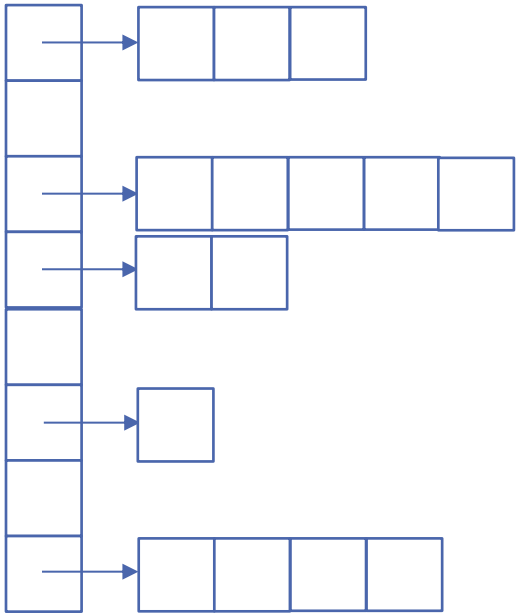


Complexidade Temporal

- Se m for mt pequeno
  - Encadeamentos muito longos



- Se m for grande
  - Muitos encadeamentos vazios (estamos a desperdiçar demasiada memória)



Complexidade Temporal

Para m = n/4

$T_{get}(n)$=

$T_{chain.get}(n))$ =

~(n/m) =

~4

O(1) → Complexidade temporal assimtótica constante!

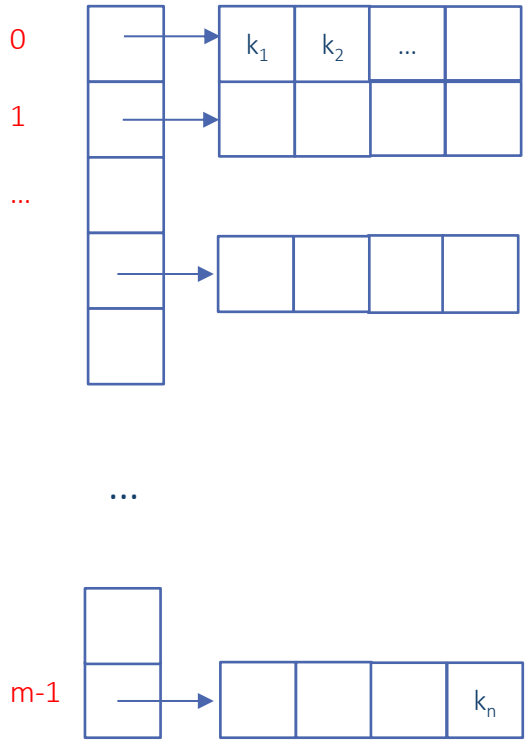


Tabela com tamanho fixo

- Quando se conheçe à partida a grandeza da quantidade de elementos que vamos guardar na tabela

  n

  - Então podemos criar uma tabela de dispersão com tamanho fixo

    m ~ n/4

    m – preferencialmente um número primo (se a função de hash for demasiado simples e não conseguir distribuir de forma uniforme)

Tabela com tamanho variável

- Quando não se sabe quantos elementos vão ser guardados na tabela
  - Para uma maior eficiência
  - O tamanho da tabela deve mudar com o número de chaves inseridas

    Dobrar o tamanho da tabela de dispersão quando n/m >= 8

    Reduzir para metade o tamanho quando n/m <= 2

    Isto requer que todas as chaves têm que voltar a ser introduzidas

    Embora hashCode() não mude, o resultado da função hash() irá mudar.

Complexidade Temporal

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
Lista Sequencial	n	n/2	n	n
Array Pesquisa Binária	n	$\log_2 n$	2n	$\log_2 n$
Árvore Pesquisa Binária	$1.39 \log_2 n$	$1.39 \log_2 n$	n	n
Árvore 2-3	$c \log_2 n$	$c \log_2 n$	$c \log_2 n$	$c \log_2 n$
Árvore Red-Black	$\log_2 n$	$\log_2 n$	$2 \log_2 n$	$2 \log_2 n$
Tabela de encadeamentos separados	**4**	**2.5**	**n** **4-8**	**n** **4-8**

Escolhendo para tamanho da tabela n/4

Isto só acontece quando a função de hash não é boa a dispersar as chaves

Não é difícil implementar boas funções de hash, portanto podemos ignorar este caso, e considerar os tempos para o caso médio

Tabelas de endereçamento aberto

Com exploração linear

Linear Probing

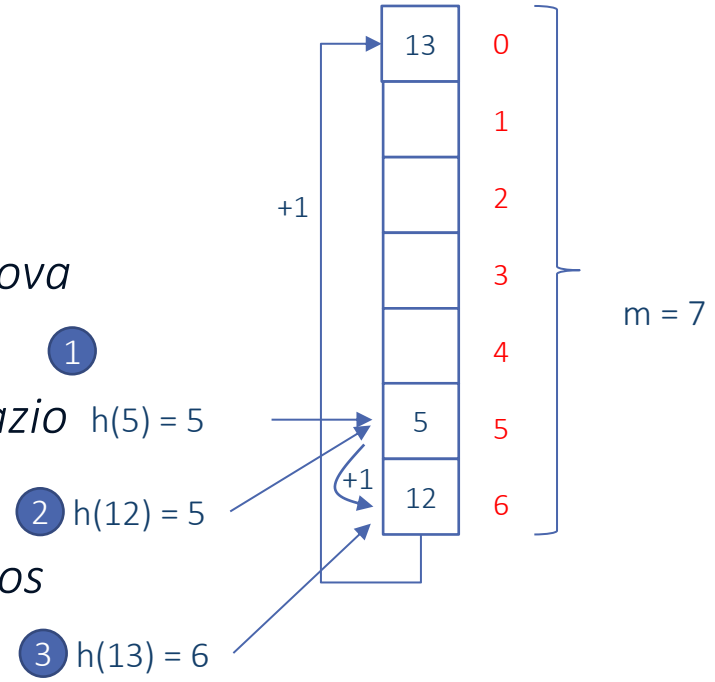
Tabelas de endereçamento aberto c/ exploração linear

- Ideia:
  - Usar posições vazias na tabela de dispersão para ajudar a resolver colisões

    Quando existe uma colisão para uma nova chave k_2

    Colocar a chave k_2 no próximo índice vazio

    Se chegarmos ao fim da tabela, voltamos ao início



Exploração linear (linear probing)
Quando existe colisão, esta técnica vai explorar as próximas posições vazias de forma linear
i=(i+1) mod m

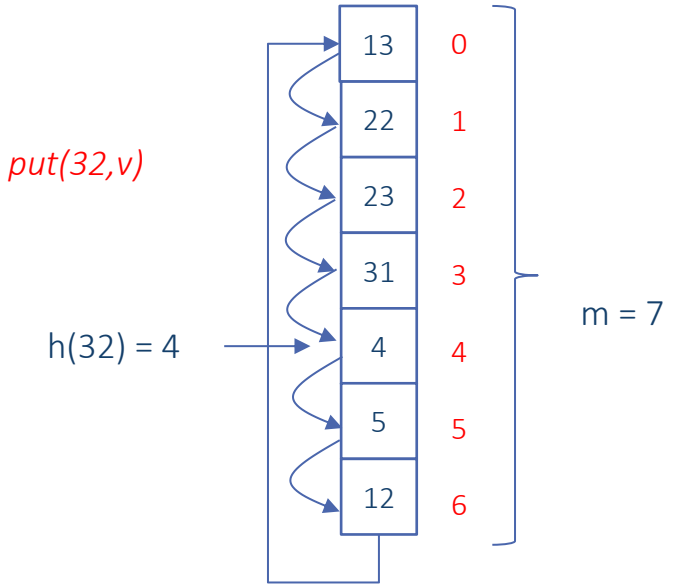
Tabelas de endereçamento aberto c/ exploração linear

- Para tabela de endereçamento aberto funcionar
  - m > n

  - Se m < n,

    tabela fica completamente cheia

    ciclo infinito a inserir chave m+1



Implementação Linear Probing

```
public class OpenAdressingHashTable<Key, Value> {
    private static int[] primes = {
            17, 37, 79, 163, 331, 673, 1361, 2729, 5471, 10949, 21911,
              43853, 87719, 175447, 350899, 701819, 1403641, 2807303,
              5614657, 11229331, 22458671, 44917381, 89834777, 179669557};
    private int m;
    private int primeIndex;
    private int size;
    private float loadFactor;

    private Key[] keys;
    private Value[] values;

    @SuppressWarnings("unchecked")
    private OpenAdressingHashTable(int primeIndex)
    {
        this.primeIndex = primeIndex;
        this.m = this.primes[primeIndex];
        this.size = 0; this.loadFactor = 0;
        this.keys = (Key[]) new Object[this.m];
        this.values = (Value[]) new Object[this.m];
    }

    public OpenAdressingHashTable(){this(0);}
```

array com valores primos que iremos usar para os vários tamanhos possíveis da tabela de dispersão

o valor primo actual que estamos a usar para o tamanho

Pesquisa

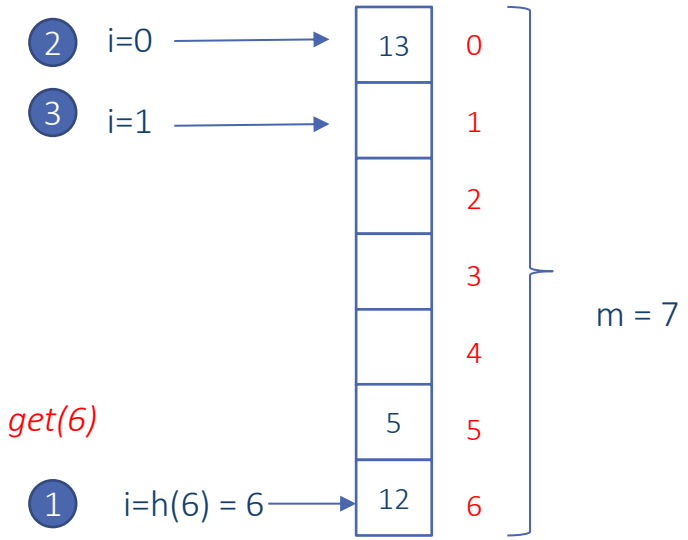
- a) usar função de hash para determinar índice inicial pesquisa
  - i = hash(k)
- b) pesquisa
  - Se keys[i] = null

    Não existe a chave, retornar null
  - Se keys[i] = k

    Encontrámos a chave, retornar valor
  - Se keys[i] ≠ k

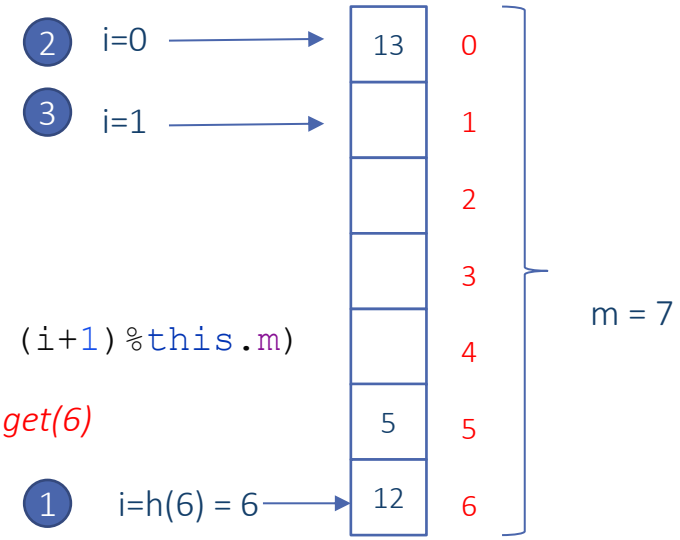
    Não encontrámos (mas pode existir)

    repetir b) para posição seguinte



Pesquisa

```
private int hash(Key k)
{
    return (k.hashCode() & 0x7fffffff) % this.m;
}

public Value get(Key k)
{
    for(int i = hash(k); this.keys[i] != null; i = (i+1)%this.m)
    {
        //key was found, return its value
        if(this.keys[i].equals(k))
        {
            return this.values[i];
        }
    }
    return null;
}
```



Inserção

Factor de carga = size/m

- Semelhante a pesquisa
  - a) se factor de carga >= 50%, redimensionar
  - b) determinar índice inicial

    i=hash(k)
  - c) inserção

    Se keys[i] = null

    Encontrámos o ponto de inserção, inserir

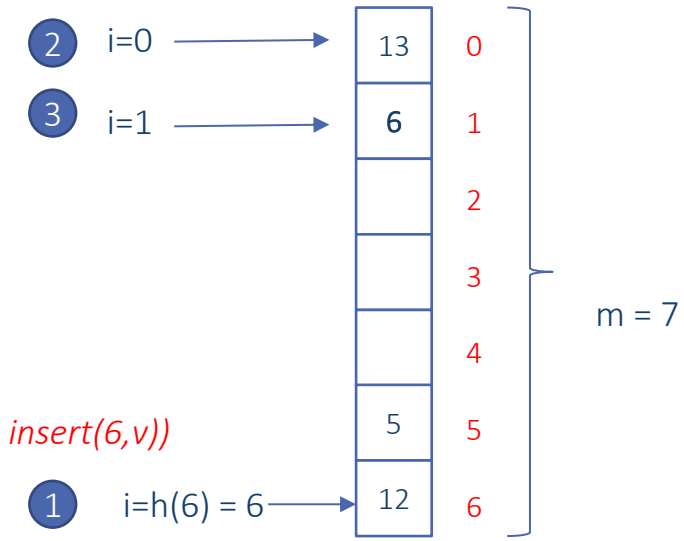
    Se keys[i] = k

    Encontrámos a chave, é um update

    Se keys[i] ≠ k

    Posição ocupada

    repetir c) para posição seguinte

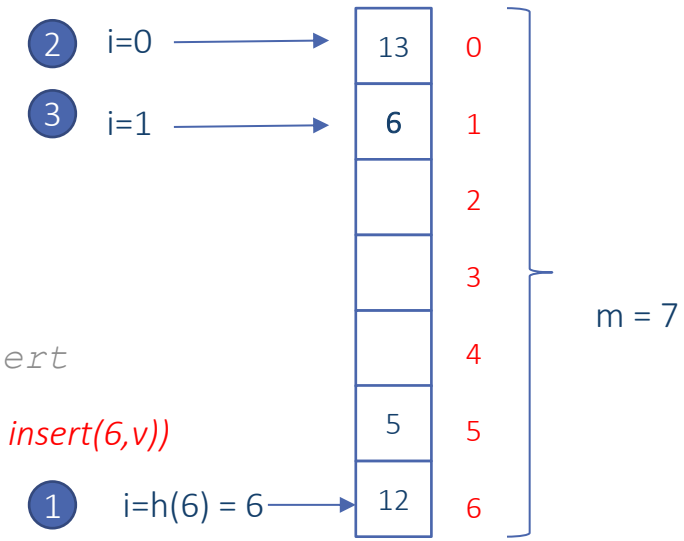


Inserção

Factor de carga = size/m

```
public void put(Key k, Value v)
{
    if(this.loadFactor >= 0.5f)
    {
        resize(this.primeIndex+1);
    }

    int i = hash(k);
    for(; this.keys[i] != null; i = (i+1)% this.m)
    {
        //key was found, update its value
        if(this.keys[i].equals(k))
        {
            this.values[i] = v;
            return;
        }
    }

    //we've found the right insertion position, insert
    this.keys[i] = k;
    this.values[i] = v;
    this.size++;
    this.loadFactor = this.size/this.m;
}
```

i é declarado fora do *for*, pois precisamos dele mesmo depois do *for* terminar



Redimensionar

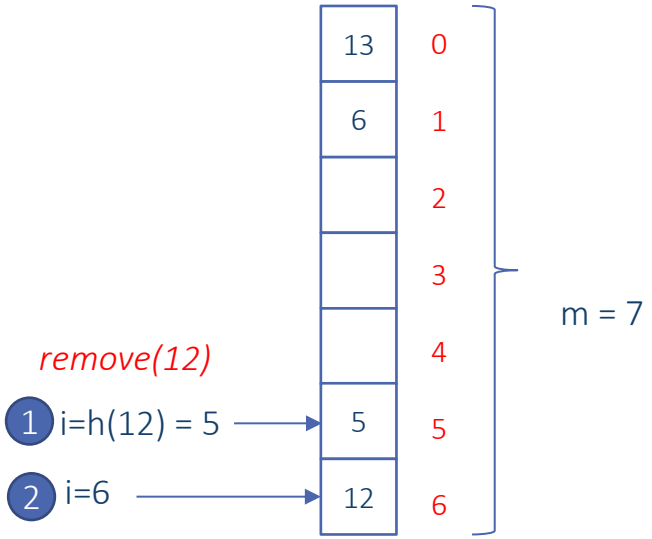
```
private void resize(int primeIndex)
{
    //if invalid size do not resize;
    if(primeIndex < 0 || primeIndex >= primes.length) return;

    this.primeIndex = primeIndex;
    OpenAdressingHashTable<Key,Value> aux =
         new OpenAdressingHashTable<Key,Value>(this.primeIndex);
    //place all existing keys in new table
    for(int i = 0; i < this.m; i++)
    {
        if(keys[i] != null) aux.put(keys[i],values[i]);
    }

    this.keys = aux.keys;
    this.values = aux.values;
    this.m = aux.m;
    this.loadFactor = this.size/this.m;
}
```

Remover chaves

- Remoção de chaves é mais complexa
  - Não podemos remover apenas a chave

    Ex. Se removermos a chave 12, deixamos de conseguir aceder à chave 6

    Chave 6 foi lá colocada por as posições anteriores estarem ocupadas
  - As chaves seguintes que sejam contínuas têm que ser reinseridas

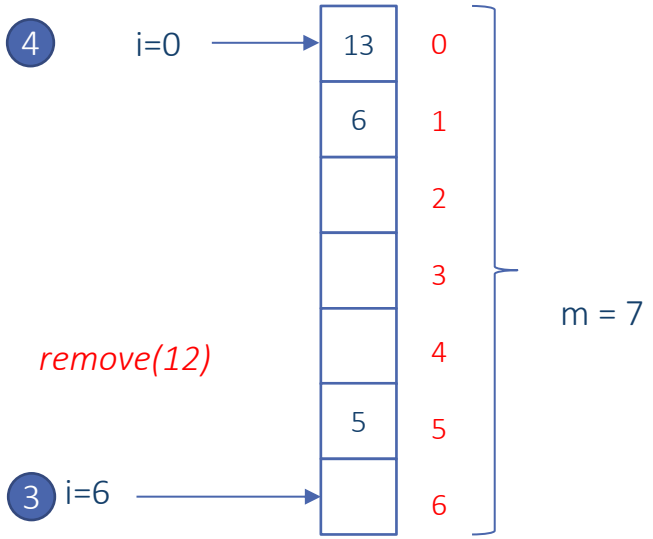


Remover chaves

- Remoção de chaves é mais complexa
  - Não podemos remover apenas a chave

    Ex. Se removermos a chave 12, deixamos de conseguir aceder à chave 6

    Chave 6 foi lá colocada por as posições anteriores estarem ocupadas
  - As chaves seguintes que sejam contínuas têm que ser reinseridas

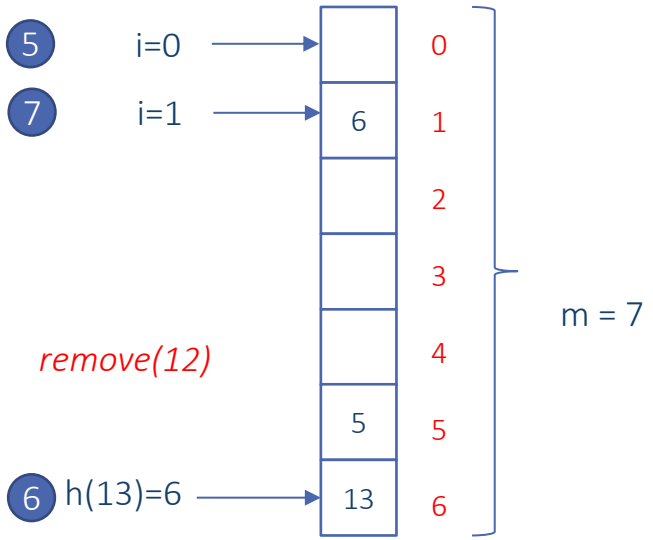


Remover chaves

- Remoção de chaves é mais complexa
  - Não podemos remover apenas a chave

    Ex. Se removermos a chave 12, deixamos de conseguir aceder à chave 6

    Chave 6 foi lá colocada por as posições anteriores estarem ocupadas
  - As chaves seguintes que sejam contínuas têm que ser reinseridas



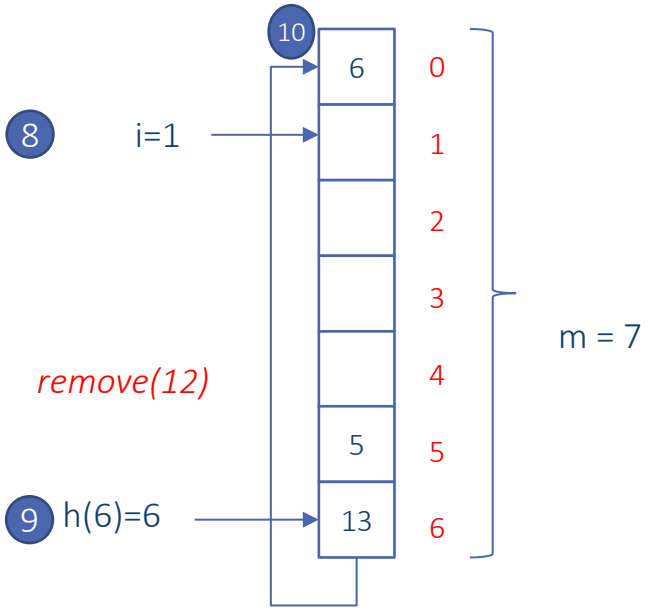
Remover chaves

- Remoção de chaves é mais complexa
  - Não podemos remover apenas a chave

    Ex. Se removermos a chave 12, deixamos de conseguir aceder à chave 6

    Chave 6 foi lá colocada por as posições anteriores estarem ocupadas
  - As chaves seguintes que sejam contínuas têm que ser reinseridas

    Se encontrarmos uma posição vazia podemos parar a reinserção



Remover Chaves

```
private void delete(Key k)
{
    int i = hash(k);
    while(true)
    {
        //no key to delete, return
        if(this.keys[i] == null) return;
        //if key was found, exit the loop
        if(this.keys[i].equals(k)) break;
        i = (i+1)%this.m;
    }

    //delete the key and value
    this.keys[i] = null;
    this.values[i] = null;
    this.size--;

    //we need to reenter any subsequent keys
    i = (i+1)%this.m;

    while(this.keys[i] != null)
    {
        Key auxKey = this.keys[i];
        Value auxValue = this.values[i];
        //remove from previous position
        this.keys[i] = null;
        this.values[i] = null;
        //temporarily reduce size,
        //next put will increment it
        this.size--;
        //add the key and value again
        this.put(auxKey,auxValue);
        i = (i+1)%this.m;
    }

    this.loadFactor = this.size/this.m;

    if(this.loadFactor < 0.125f)
        resize(this.primeIndex-1);
}
```

este ciclo procura a chave a remover

este ciclo reintroduz as chaves à "direita" da chave removida

Complexidade Temporal

- Assumindo que a função de hash distribui uniformemente as chaves
- Número médio de comparações para uma tabela com factor de carga α
  - $\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ no caso de search hit
  - $\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ no caso de search miss/insert

Complexidade Temporal

- Número médio de comparações para uma tabela com factor de carga α
  - $\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ no caso de search hit
  - $\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ no caso de search miss/insert
- Para α=50% (=) α=1/2
  - ~ 3/2 = 1.5 no caso de search hit
  - ~ 5/2 = 2.5 no caso de search miss/insert

Complexidade Temporal

	Caso Médio		Pior Caso	
	Inserção	Pesquisa	Inserção	Pesquisa
Árvore Red-Black	$\log_2 n$	$\log_2 n$	$2 \log_2 n$	$2 \log_2 n$
Tabela de encadeamentos separados	4* 2**	2.5* 1.5**	n 4-8* 2-4**	n 4-8* 2-4**
Tabela de endereçamento aberto c/ exploração linear	**2.5*** **	**1.5*** **	**n 2-4*** **	**n 2-4*** **

Observação: Uma tabelas de encadeamentos separados (TES) embora seja bastante eficiente para m=n/2, costuma usar-se mais com m=n/4 porque a TES tende a desperdiçar mais memória.
Uma TES com m = n/2 gasta 4*n bytes a mais do que uma TEA com fator de carga 50%. Uma TES com m=n/4 gasta 2*n bytes a mais.
Portanto, se memória não for importante, e estivermos dispostos a usar m=n/2, a eficiência temporal da TES é muito boa.

Isto só acontece quando a função de hash não é boa a dispersar as chaves

Portanto podemos ignorar este caso, e considerar ~2.5 no pior caso.

* assumindo m = n/4
* assumindo m = n/2
**assumindo um fator de carga de 50%