# Aula 12
# Dicionários
## Tabelas de dispersão

*Programação II, 2018-2019*

*v1.4, 17-05-2018*

DETI, Universidade de Aveiro

# Sumário

# Sumário

# Colecções de dados: o que vimos até agora

- `LinkedList`
  - `addFirst()`, `addLast()`, `removeFirst()`, `first()`, ...
- `SortedList`
  - `insert()`, `remove()`, `first()`, ...
- `Stack`
  - `push()`, `pop()`, `top()`, ...
- `Queue`
  - `in()`, `out()`, `peek()`, ...
- `KeyValueList` (implementa um **dicionário**)
  - `set()`, `get()`, `remove()`, ...

# Colecções de dados: o que vimos até agora

- `LinkedList`
  - `addFirst()`, `addLast()`, `removeFirst()`, `first()`, ...
- `SortedList`
  - `insert()`, `remove()`, `first()`, ...
- `Stack`
  - `push()`, `pop()`, `top()`, ...
- `Queue`
  - `in()`, `out()`, `peek()`, ...
- `KeyValueList` (implementa um **dicionário**)
  - `set()`, `get()`, `remove()`, ...

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.

1. Vectores
2. Listas Ligadas
3. Dicionários

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Colecções de dados: o que vimos até agora

- Analisámos a sua eficiência em termos de **espaço** de memória e **tempo** de execução.
  1. Vectores
     - Espaço: $O(n)$ (proporcional ao número de elementos).
     - Tempo (acesso por índice): $O(1)$ (constante).
     - Tempo (procura por valor): $O(n)$.
     - Tempo (inserção no fim): $O(1)$.
     - Tempo (procura em vector ordenado): $O(\log n)$.
     - Tempo (inserção por ordem): $O(n)$.
  2. Listas Ligadas
     - Espaço: $O(n)$.
     - Tempo (acesso, procura): $O(n)$.
     - Tempo (inserção): $O(1)$.
  3. Dicionários
     - Eficiência depende da implementação.
     - No caso de implementação na forma de lista de pares chave-valor (aula anterior), a eficiência é similar à das listas.
     - Vamos agora ver implementações eficientes do conceito de dicionário.

# Dicionários: problema

- Uma empresa pretende aceder à informação de cada empregado usando como chave o respectivo *Número de Identificação de Segurança Social* (*NISS*).
  - O NISS tem 11 dígitos.
  - A empresa só tem algumas centenas ou milhares de empregados.
  - Como garantir tempo de acesso $O(1)$?
- Implementação em **lista de pares chave-valor**.
  - Não suporta a complexidade pretendida.
- Poderíamos usar o NISS como índice num **vector** de empregados.
  - Teria que ser um vector com dimensão $10^{11}$ e índices entre 0 e 99 999 999 999.
  - Só iríamos utilizar uma pequeníssima percentagem das entradas do vector!
  - Conclusão: para termos tempo $O(1)$, teríamos de desperdiçar muito espaço de memória.

# Dicionários: problema

- Uma empresa pretende aceder à informação de cada empregado usando como chave o respectivo *Número de Identificação de Segurança Social* (*NISS*).
  - O NISS tem 11 dígitos.
  - A empresa só tem algumas centenas ou milhares de empregados.
  - Como garantir tempo de acesso $O(1)$?
- Implementação em **lista de pares chave-valor**.
  - Não suporta a complexidade pretendida.
- Poderíamos usar o NISS como índice num **vector** de empregados.
  - Teria que ser um vector com dimensão $10^{11}$ e índices entre 0 e 99 999 999 999.
  - Só iríamos utilizar uma pequeníssima percentagem das entradas do vector!
  - Conclusão: para termos tempo $O(1)$, teríamos de desperdiçar muito espaço de memória.

# Dicionários: problema

- Uma empresa pretende aceder à informação de cada empregado usando como chave o respectivo *Número de Identificação de Segurança Social* (*NISS*).
  - O NISS tem 11 dígitos.
  - A empresa só tem algumas centenas ou milhares de empregados.
  - Como garantir tempo de acesso $O(1)$?
- Implementação em **lista de pares chave-valor**.
  - Não suporta a complexidade pretendida.
- Poderíamos usar o NISS como índice num **vector** de empregados.
  - Teria que ser um vector com dimensão $10^{11}$ e índices entre 0 e 99 999 999 999.
  - Só iríamos utilizar uma pequeníssima percentagem das entradas do vector!
  - Conclusão: para termos tempo $O(1)$, teríamos de desperdiçar muito espaço de memória.

# Dicionários: problema

- Uma empresa pretende aceder à informação de cada empregado usando como chave o respectivo *Número de Identificação de Segurança Social* (*NISS*).
  - O NISS tem 11 dígitos.
  - A empresa só tem algumas centenas ou milhares de empregados.
  - Como garantir tempo de acesso $O(1)$?
- Implementação em **lista de pares chave-valor**.
  - Não suporta a complexidade pretendida.
- Poderíamos usar o NISS como índice num **vector** de empregados.
  - Teria que ser um vector com dimensão $10^{11}$ e índices entre 0 e 99 999 999 999.
  - Só iríamos utilizar uma pequeníssima percentagem das entradas do vector!
  - Conclusão: para termos tempo $O(1)$, teríamos de desperdiçar muito espaço de memória.

# Dicionários: como optimizar?





- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de $O(n)$ para $O(\log n)$.
  - Neste caso, as listas transformam-se em árvores binárias (aula 19).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - É não para o número total de chaves possíveis!
    - No exemplo dado, o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.
  - Ou vector.

# Dicionários: como optimizar?

- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de $O(n)$ para $O(\log n)$.
  - Neste caso, as listas transformam-se em árvores binárias (aula 13).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - E não para o número total de chaves possíveis!
    - No exemplo dado: o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.

    do vector.

# Dicionários: como optimizar?

- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de $O(n)$ para $O(\log n)$.
  - Neste caso, as listas transformam-se em árvores binárias (aula 13).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - E não para o número total de chaves possíveis!
    - No exemplo dado: o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.

    do vector.

# Dicionários: como optimizar?

- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de $O(n)$ para $O(\log n)$.
  - Neste caso, as listas transformam-se em árvores binárias (aula 13).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - E não para o número total de chaves possíveis!
    - No exemplo dado: o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.

    do vector.

# Dicionários: como optimizar?

- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de $O(n)$ para $O(\log n)$.
  - Neste caso, as listas transformam-se em árvores binárias (aula 13).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - E não para o número total de chaves possíveis!
    - No exemplo dado: o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.

  do vector.

# Dicionários: como optimizar?

- Lista de pares chave-valor.
  - Se cada nó passar a apontar para dois nós, em vez de apenas um, o tempo de acesso por chave pode reduzir-se de $O(n)$ para $O(\log n)$.
  - Neste caso, as listas transformam-se em árvores binárias (aula 13).
- Vector.
  - O vector é dimensionado tendo em conta uma previsão do número médio ou máximo de pares chave-valor a armazenar.
    - E não para o número total de chaves possíveis!
    - No exemplo dado: o número de empregados é uma fracção ínfima de todos os inscritos na Segurança Social.

    do vector.

# Dicionários: implementação usando vector





- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feito pela chamada função de dispersão (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como tabelas de dispersão (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

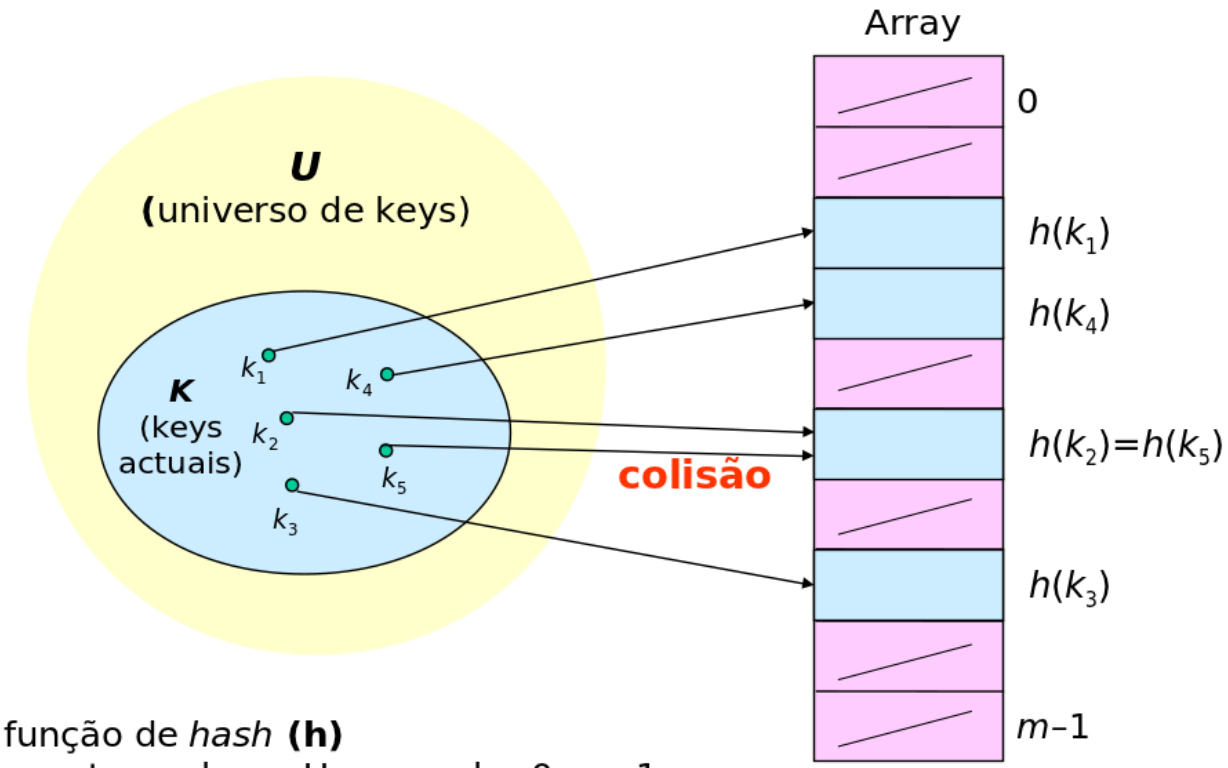
# Dicionários: implementação usando vector

- Objectivo: desempenho com o melhor dos "dois mundos":
  - Tempo de acesso / procura por chave: $O(1)$, como nos vectores.
  - Tempo de inserção: $O(1)$, como nas listas não ordenadas.
  - Espaço: $O(n)$, onde $n$ é o número de pares armazenados.
- Para cada chave a inserir ou procurar, **calcula-se** o índice correspondente no vector.
  - O mapeamento das chaves para índices válidos do vector é feita pela chamada **função de dispersão** (*hash function*).
  - A função de dispersão é determinística: dada a mesma chave, devolve sempre o mesmo índice.
  - Várias chaves podem ser mapeadas no mesmo índice.
  - Mas convém que as chaves fiquem bem distribuídas (dispersas) pelos índices do vector.
  - Dicionários implementados em vector com função de dispersão são conhecidos como **tabelas de dispersão** (*hash tables*).

# Tabelas de dispersão





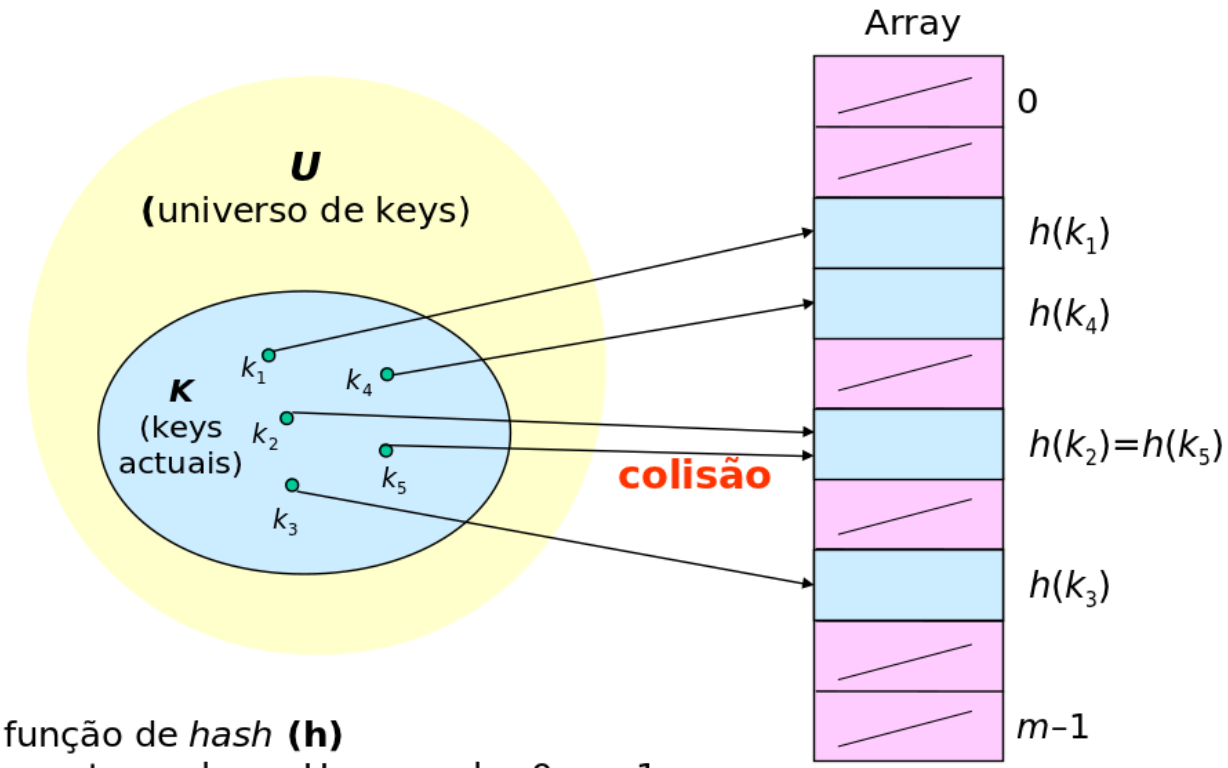



A função de *hash* **(h)**
converte qualquer U num valor 0...m-1

# Tabelas de dispersão








A função de *hash* **(h)**
converte qualquer U num valor 0...m-1

# Módulo *HashTable* (tabela de dispersão)





- Nome do módulo:
  - HashTable
- Serviços:
  - HashTable(n): construtor;
  - get(key): devolve o elemento associado à chave dada;
  - set(key, elem): actualiza o elemento associado à chave k, caso esta exista, ou insere o novo par (k, e);
  - remove(key): remove a chave dada bem como o elemento associado;
  - contains(key): tabela contem a chave dada;
  - isEmpty(): tabela vazia;
  - size(): número de associações;
  - clear(): limpa a tabela;
  - keys(): devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Módulo *HashTable* (tabela de dispersão)

- Nome do módulo:
  - `HashTable`
- Serviços:
  - `HashTable(n)`: construtor;
  - `get(key)`: devolve o elemento associado à chave dada
  - `set(key, elem)`: actualiza o elemento associado à chave `k`, caso esta exista, ou insere o novo par `(k, e)`
  - `remove(key)`: remove a chave dada bem como o elemento associado
  - `contains(key)`: tabela contém a chave dada
  - `isEmpty()`: tabela vazia
  - `size()`: número de associações;
  - `clear()`: limpa a tabela;
  - `keys()`: devolve um vector com todas as chaves existentes.

# Tabelas de dispersão: Funções de *Hash*





- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:
  - Função de Compressão (*m* é a dimensão do vector)
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisões: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:
    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)
    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*

- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*





- Funções de *Hash* (duas partes):
  - Cálculo do *hash code*:

    `chave` $\longrightarrow$ `inteiro`
  - Função de Compressão ($m$ é a dimensão do vector)

    `inteiro` $\longrightarrow$ `inteiro` $[0, m-1]$
- $h(k)$ é o valor de *hash* da chave $k$.
- Problema:
  - Colisão: chaves distintas podem produzir o mesmo valor de *hash* (i.e. mesmo índice do vector)!

# Tabelas de dispersão: Funções de *Hash*





- A escolha de uma "boa" função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma "boa" função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)$...

# Tabelas de dispersão: Funções de *Hash*

- A escolha de uma “boa” função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma “boa” função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)$...

# Tabelas de dispersão: Funções de *Hash*

- A escolha de uma “boa” função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma “boa” função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)\ldots$

# Tabelas de dispersão: Funções de *Hash*

- A escolha de uma "boa" função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma "boa" função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)$...

# Tabelas de dispersão: Funções de *Hash*

- A escolha de uma "boa" função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma "boa" função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)\ldots$

# Tabelas de dispersão: Funções de *Hash*

- A escolha de uma "boa" função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma "boa" função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)$...

# Tabelas de dispersão: Funções de *Hash*

- A escolha de uma "boa" função de *hash* deve minimizar o número de colisões.
  - O desempenho da tabela de dispersão depende da capacidade da função de *hash* para distribuir uniformemente as chaves pelos índices do vector.
- A escolha de uma "boa" função de *hash* pode ter em consideração o tipo dos dados que serão utilizados:
  - Uma análise estatística da distribuição das chaves pode ser considerada.
- O valor de *hash* deve ser independente de qualquer padrão que exista nos dados (chaves).
- Vamos ver vários exemplos de $h(k)$...

# Funções de *hash*: aproximações





1. Método da divisão:
   - Este método usa o resto da divisão inteira:
     $$h(k) = k \,\%\, m$$
   - Se $m$ é par, então
     $$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{impar} & \text{se } k \text{ é impar} \end{cases}$$
   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*.
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$h(k) = k \,\%\, m$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações





1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: aproximações

1. Método da divisão:
   - Este método usa o resto da divisão inteira:

$$\boxed{h(k) = k \,\%\, m}$$

   - Se $m$ é par, então

$$h(k) = \begin{cases} \text{par} & \text{se } k \text{ é par} \\ \text{ímpar} & \text{se } k \text{ é ímpar} \end{cases}$$

   - Outra má opção é $m = 2^p$ ($h(k)$ serão os $p$ bits menos significativos).
   - Para este método utilizar um valor primo para $m$ é uma escolha razoável.
2. Método da multiplicação:
   - Pode fazer uso dos operadores de *bit shift*
   - Exemplo: $h(k) = (k << 3) + (k >> 28) + 33$

# Funções de *hash*: Exemplo para chaves tipo `String`

```
private int hashstring(String str, int tablesize)
{
  int len=str.length();
  long hash=0;
  char[] buffer=str.toCharArray();

  int c=0;
  for (int i=0; i < len; i++)
  {
    c = buffer[i]+33;
    hash = ((hash<<3) + (hash>>28) + c);
  }

  hash = hash % tablesize;
  return (int) (hash>=0 ? hash : hash + tablesize);
}
```

- Todos os objectos em `Java` têm uma função de dispersão, `hashCode()`, que devolve um inteiro.
- Vamos utilizar esta função nas nossas tabelas de dispersão.

# Funções de *hash*: Exemplo para chaves tipo `String`

```
private int hashstring(String str, int tablesize)
{
  int len=str.length();
  long hash=0;
  char[] buffer=str.toCharArray();

  int c=0;
  for (int i=0; i < len; i++)
  {
    c = buffer[i]+33;
    hash = ((hash<<3) + (hash>>28) + c);
  }

  hash = hash % tablesize;
  return (int) (hash>=0 ? hash : hash + tablesize);
}
```

- Todos os objectos em `Java` têm uma função de dispersão, `hashCode()`, que devolve um inteiro.
- Vamos utilizar esta função nas nossas tabelas de dispersão.

# Funções de *hash*: Exemplo para chaves tipo `String`

```
private int hashstring(String str, int tablesize)
{
  int len=str.length();
  long hash=0;
  char[] buffer=str.toCharArray();

  int c=0;
  for (int i=0; i < len; i++)
  {
    c = buffer[i]+33;
    hash = ((hash<<3) + (hash>>28) + c);
  }

  hash = hash % tablesize;
  return (int) (hash>=0 ? hash : hash + tablesize);
}
```

- Todos os objectos em `Java` têm uma função de dispersão, `hashCode()`, que devolve um inteiro.
- Vamos utilizar esta função nas nossas tabelas de dispersão.

# Funções de *hash*: Exemplo para chaves tipo `String`

```
private int hashstring(String str, int tablesize)
{
  int len=str.length();
  long hash=0;
  char[] buffer=str.toCharArray();

  int c=0;
  for (int i=0; i < len; i++)
  {
    c = buffer[i]+33;
    hash = ((hash<<3) + (hash>>28) + c);
  }

  hash = hash % tablesize;
  return (int) (hash>=0 ? hash : hash + tablesize);
}
```

- Todos os objectos em `Java` têm uma função de dispersão, `hashCode()`, que devolve um inteiro.
- Vamos utilizar esta função nas nossas tabelas de dispersão.

# Tabelas de dispersão: Factor de Carga





- O factor de carga (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$).
- Dimensionamento de $\alpha$:
  - um valor alto de $\alpha$ significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de $\alpha$ significa que temos muito espaço desperdiçado;
  - valor recomendado para $\alpha$: entre 50% e 80%.

# Tabelas de dispersão: Factor de Carga

- O factor de carga (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$).
- Dimensionamento de $\alpha$:
  - um valor alto de $\alpha$ significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de $\alpha$ significa que temos muito espaço desperdiçado;
  - valor recomendado para $\alpha$: entre 50% e 80%.

# Tabelas de dispersão: Factor de Carga

- O factor de carga (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$).
- Dimensionamento de $\alpha$:
  - um valor alto de $\alpha$ significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de $\alpha$ significa que temos muito espaço desperdiçado;
  - valor recomendado para $\alpha$: entre 50% e 80%.

# Tabelas de dispersão: Factor de Carga

- O factor de carga (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$).
- Dimensionamento de $\alpha$:
  - um valor alto de $\alpha$ significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de $\alpha$ significa que temos muito espaço desperdiçado;
  - valor recomendado para $\alpha$: entre 50% e 80%.

# Tabelas de dispersão: Factor de Carga

- O factor de carga (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$).
- Dimensionamento de $\alpha$:
  - um valor alto de $\alpha$ significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de $\alpha$ significa que temos muito espaço desperdiçado;
  - valor recomendado para $\alpha$: entre 50% e 80%.

# Tabelas de dispersão: Factor de Carga

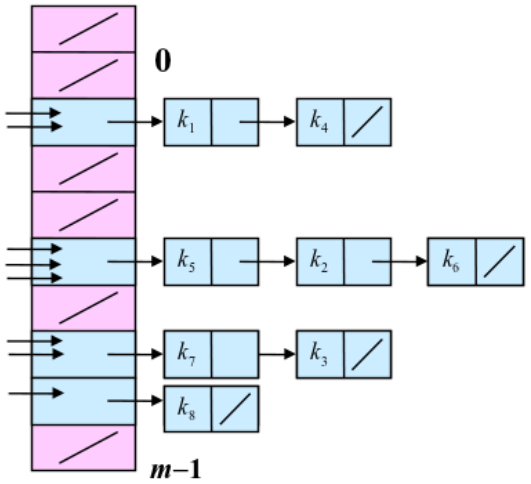




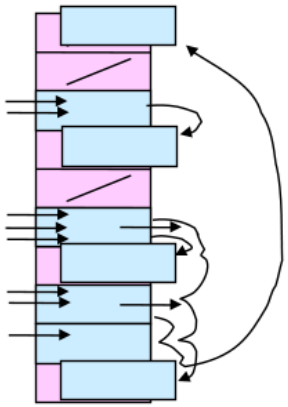
- O factor de carga (*load factor*) é o número de elementos na tabela dividido pelo tamanho da tabela ($\alpha = \frac{n}{m}$).
- Dimensionamento de $\alpha$:
  - um valor alto de $\alpha$ significa que vamos ter maior probabilidade de colisões;
  - um valor baixo de $\alpha$ significa que temos muito espaço desperdiçado;
  - valor recomendado para $\alpha$: entre 50% e 80%.

# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

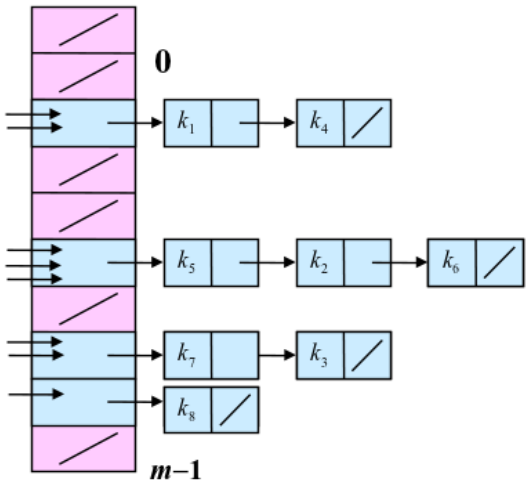



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento constituído para encontrar uma posição livre a armazenar ali;
   - O vector é tratado como circular.

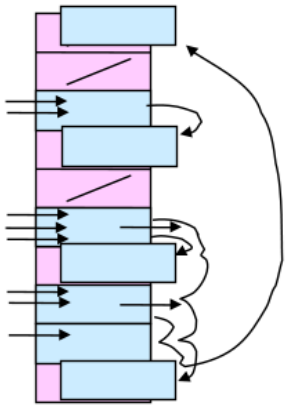


# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

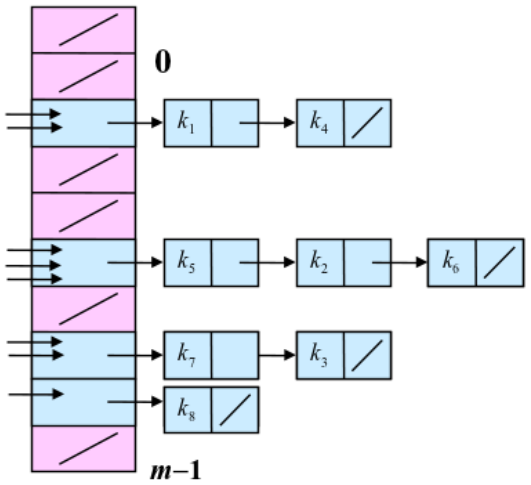



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.

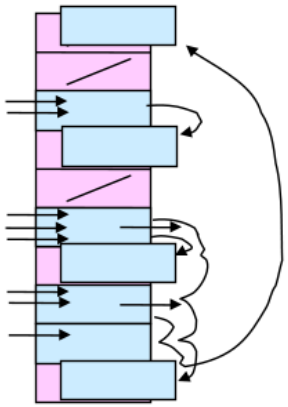


# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

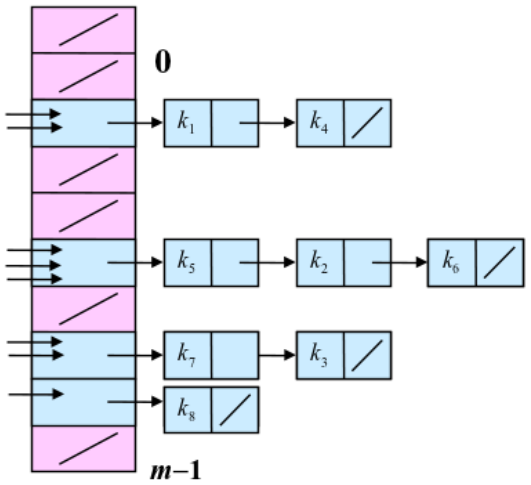



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.

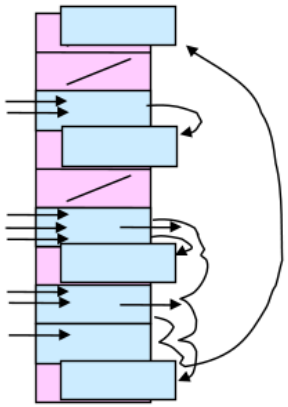


# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

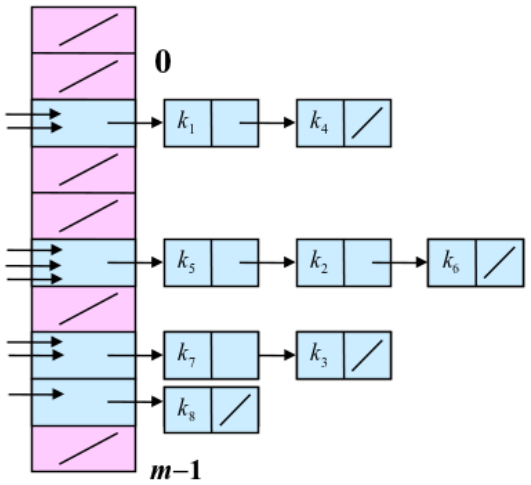



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.

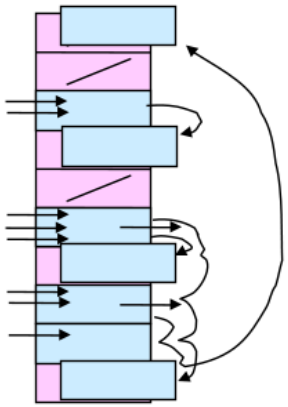


# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

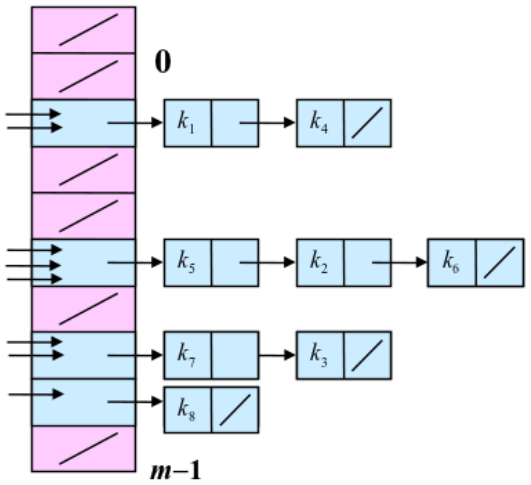



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.

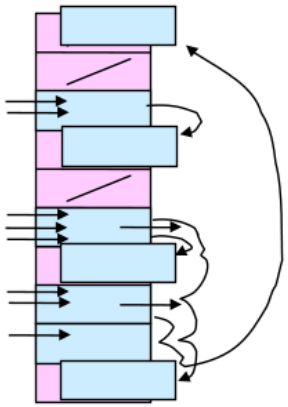


# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

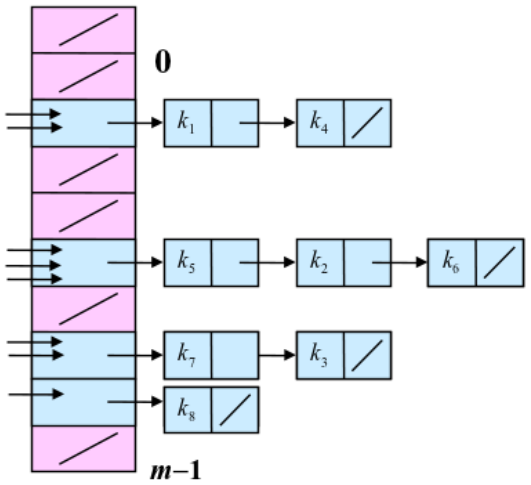



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.

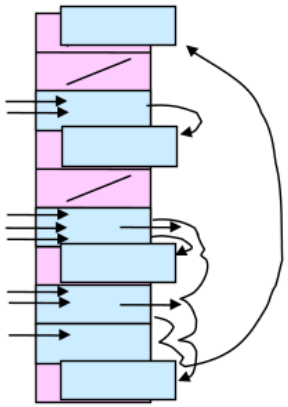


# Resolução do Problema das Colisões

1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.

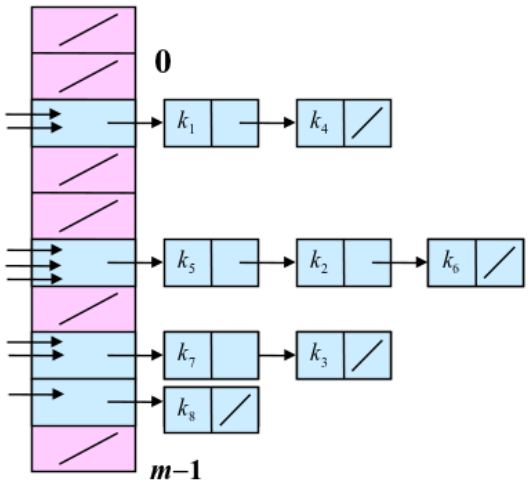



2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.

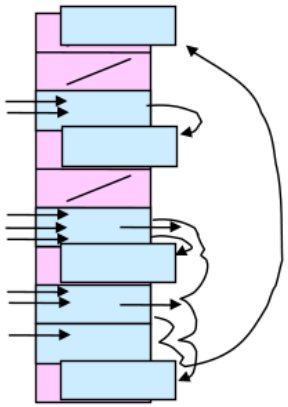


# Resolução do Problema das Colisões

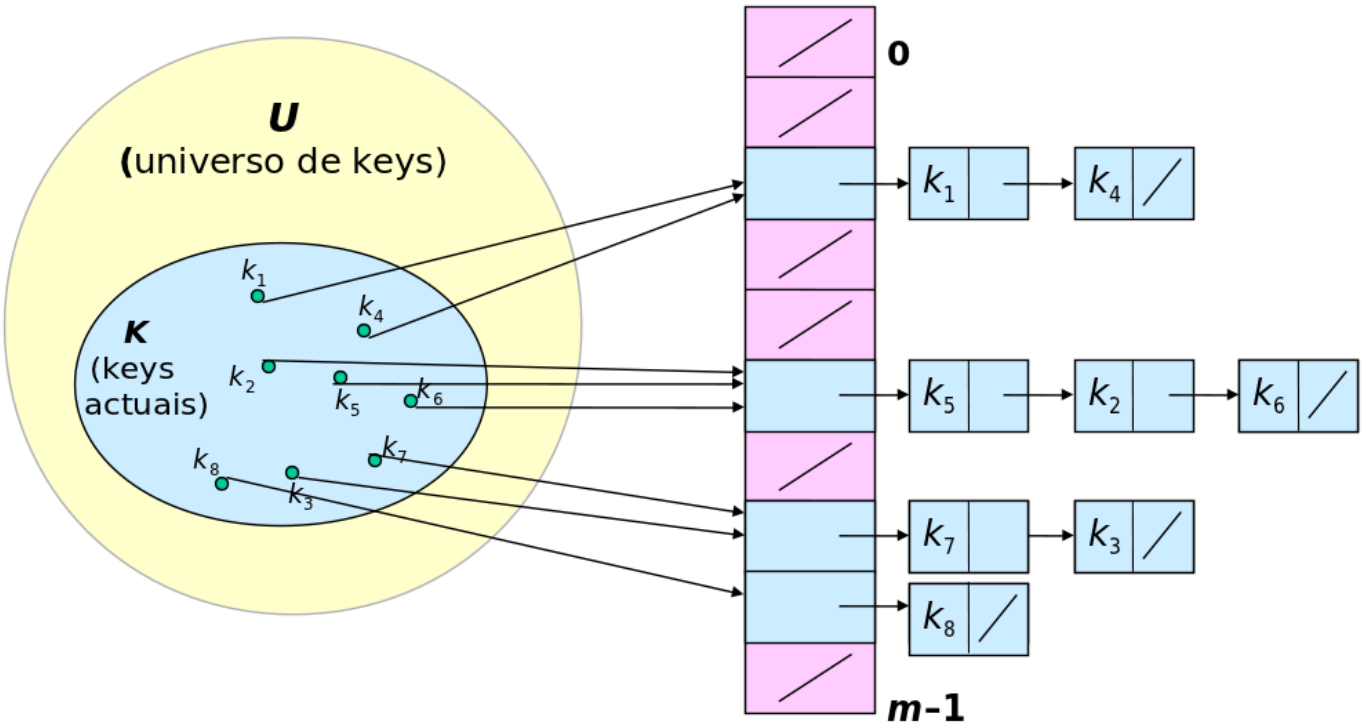
1. *Tabela de dispersão com encadeamento externo (Separate Chaining / Closed Addressing Hash Table)*
   - Múltiplos pares chaves-valor associados a um mesmo índice;
   - Cada entrada do vector contém uma lista ligada de pares chave-valor.



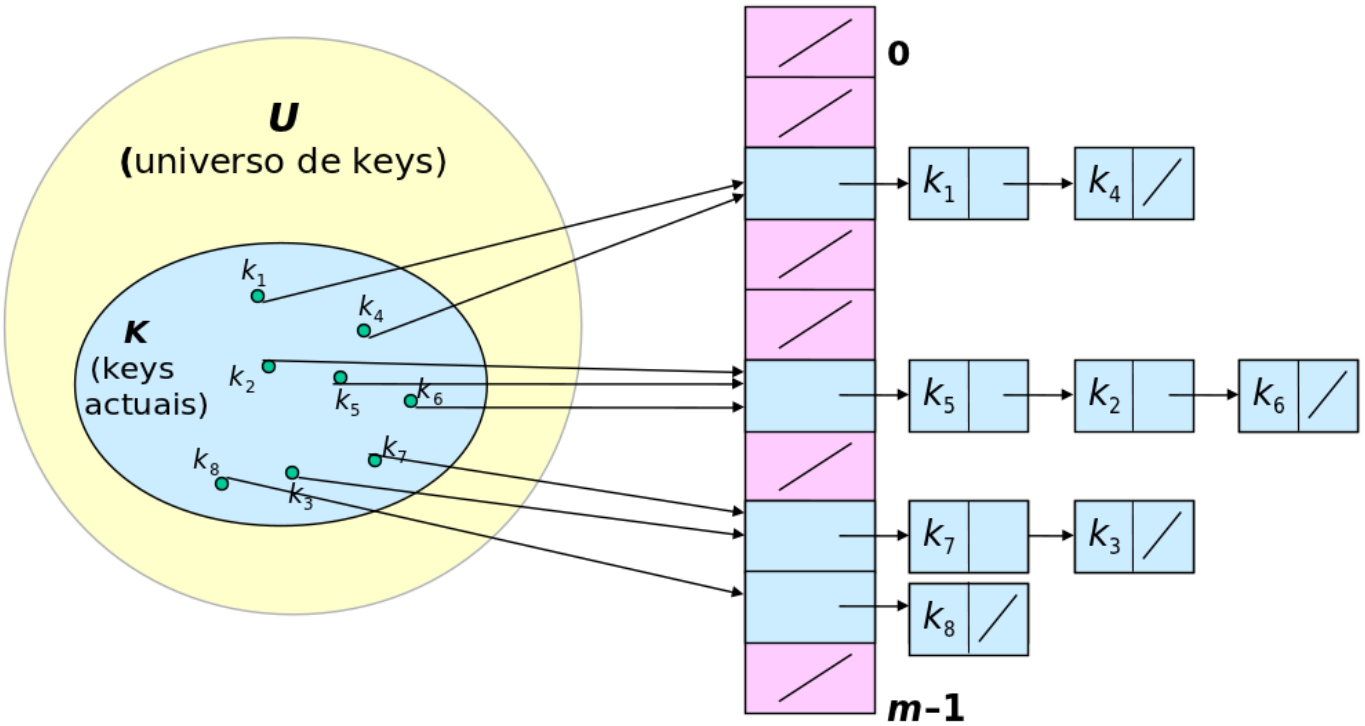

2. *Tabela de dispersão com encadeamento interno (Open Addressing Hash Table)*
   - No máximo, um par chave-valor em cada posição do vector;
   - No caso de colisão, segue-se um procedimento consistente para encontrar uma posição livre e armazenar aí;
   - O vector é tratado como circular.



# Tabela de dispersão com encadeamento externo

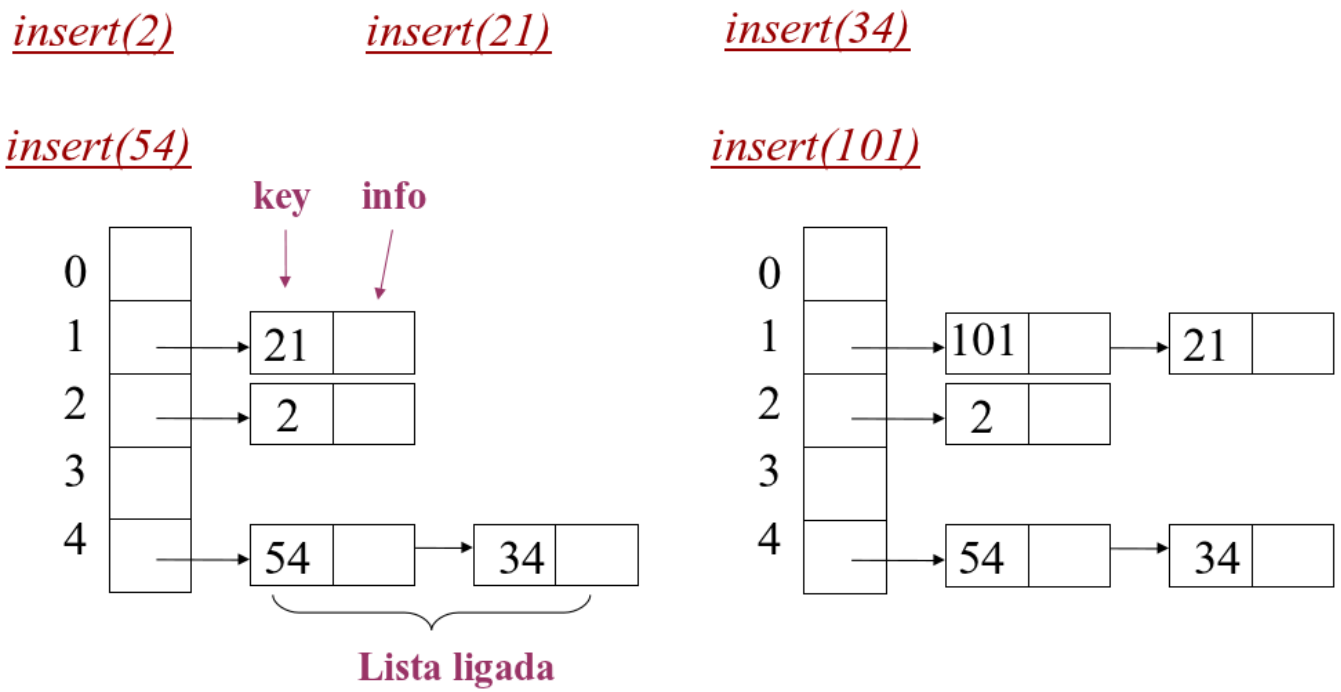


# Tabela de dispersão com encadeamento externo



# *Tabela de dispersão com encadeamento externo*: exemplo

- $h(k) = k \,\%\, m$ com $m = 5$ e $k \in [0; 999]$



# Tabela de dispersão com encadeamento externo

- Complexidade Temporal:
  - Inserção: $O(1)$
  - Pesquisa: proporcional ao comprimento máximo da lista ligada
  - Remoção: o mesmo que a pesquisa
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# Tabela de dispersão com encadeamento externo

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# Tabela de dispersão com encadeamento externo

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# *Tabela de dispersão com encadeamento externo*

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# Tabela de dispersão com encadeamento externo

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# Tabela de dispersão com encadeamento externo

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# *Tabela de dispersão com encadeamento externo*

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# *Tabela de dispersão com encadeamento externo*

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# *Tabela de dispersão com encadeamento externo*

- Complexidade Temporal:
  - **Inserção**: $O(1)$
    - tempo de cálculo da $h(k)$ + tempo de inserção no início da lista ligada.
  - **Pesquisa**: proporcional ao comprimento máximo da lista ligada.
  - **Remoção**: o mesmo que a pesquisa.
- Não esquecendo que . . . uma má função de *hash* pode comprometer todo o desempenho da tabela de dispersão!

# *Tabela de dispersão com encadeamento externo*: esqueleto

```
public class HashTable<E> {

   public HashTable(int n) {
      array = (KeyValueList<E>[])new KeyValueList[n];
      for(int i = 0; i < array.length; i++)
         array[i] = new KeyValueList<E>();
   }
   public E get(String k) {
      assert contains(k) : "Key does not exist";
      ... ...
   }
   public void set(String k, E e) {
      ... ...
      assert contains(k) && get(k).equals(e);
   }
   public void remove(String k) {
      assert contains(k) : "Key does not exist";
      ... ...
      assert !contains(k) : "Key still exists";
   }
   public boolean contains(String k) { ... }
   public String[] keys() { ... }
   public int size() { ... }
   public boolean isEmpty() { ... }

   private KeyValueList<E>[] array;
   private int size = 0;
}
```

# *Tabela de dispersão com encadeamento externo*: esqueleto

```
public class HashTable<E> {

   public HashTable(int n) {
      array = (KeyValueList<E>[])new KeyValueList[n];
      for(int i = 0; i < array.length; i++)
         array[i] = new KeyValueList<E>();
   }
   public E get(String k) {
      assert contains(k) : "Key does not exist";
      ... ...
   }
   public void set(String k, E e) {
      ... ...
      assert contains(k) && get(k).equals(e);
   }
   public void remove(String k) {
      assert contains(k) : "Key does not exist";
      ... ...
      assert !contains(k) : "Key still exists";
   }
   public boolean contains(String k) { ... }
   public String[] keys() { ... }
   public int size() { ... }
   public boolean isEmpty() { ... }

   private KeyValueList<E>[] array;
   private int size = 0;
}
```

# *Tabela de dispersão com encadeamento externo*: `set` & `get`

```
public class HashTable<E> {
   ...
   public E get(String key)
   {
      assert contains(key);

      int pos = hashFcn(key);
      return array[pos].get(key);
   }

   public void set(String key, E elem)
   {
      int pos = hashFcn(key);
      boolean newelem = array[pos].set(key, elem);
      if (newelem) size++;

      assert contains(key) && get(key).equals(elem);
   }
   ...
}
```

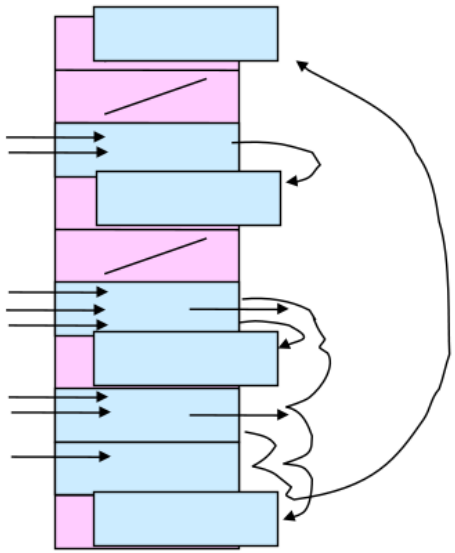
# *Tabela de dispersão com encadeamento externo*: `set` & `get`

```
public class HashTable<E> {
   ...
   public E get(String key)
   {
      assert contains(key);

      int pos = hashFcn(key);
      return array[pos].get(key);
   }

   public void set(String key, E elem)
   {
      int pos = hashFcn(key);
      boolean newelem = array[pos].set(key, elem);
      if (newelem) size++;

      assert contains(key) && get(key).equals(elem);
   }
   ...
}
```

# Tabela de dispersão com encadeamento interno



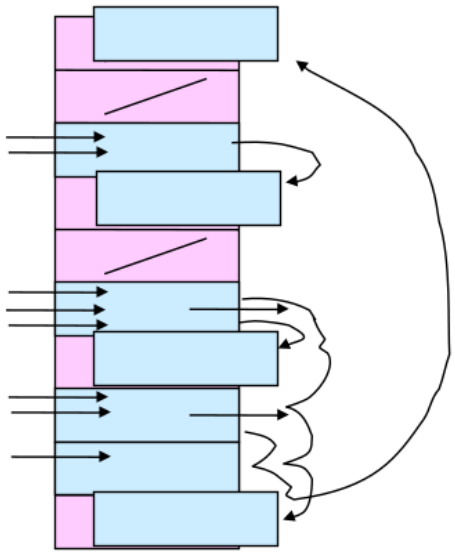
- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.
- Resolução de Colisões:
  - $h_0 = h(k)$
  - se posição $h_0$ ocupada, então tentar:
  - $h_{i+1} = (h_i + c) \% n$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, ...).



# Tabela de dispersão com encadeamento interno



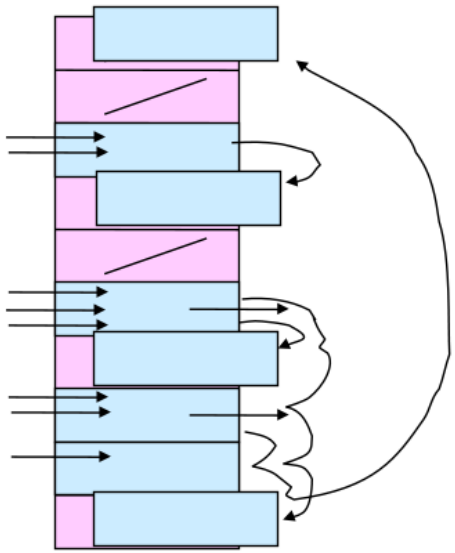
- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.
- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, ...).



# Tabela de dispersão com encadeamento interno



- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.
- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, ...).

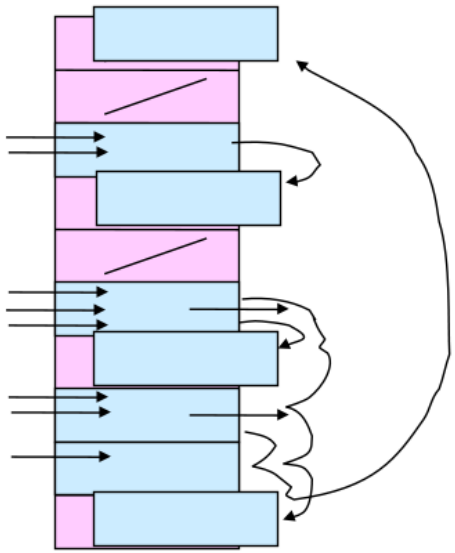


# Tabela de dispersão com encadeamento interno



- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, ...).



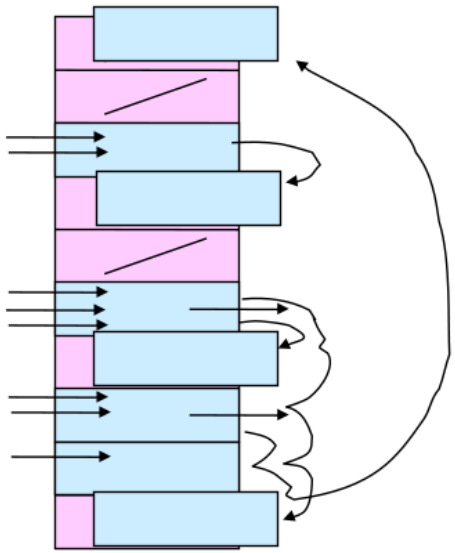
# Tabela de dispersão com encadeamento interno





- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, . . . ).



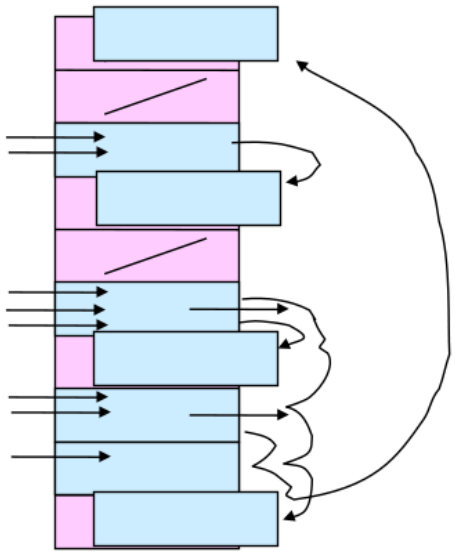
# Tabela de dispersão com encadeamento interno





- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, . . . ).

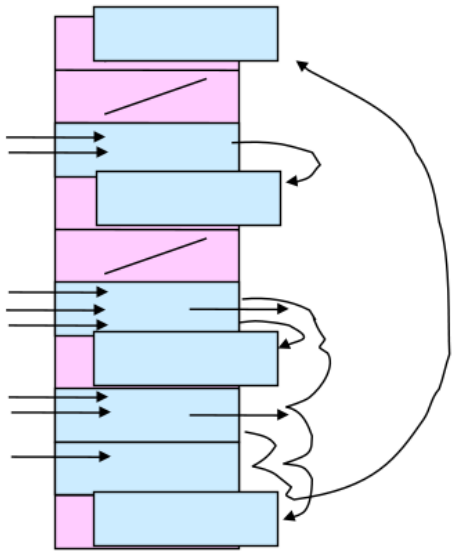


# Tabela de dispersão com encadeamento interno



- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, . . . ).

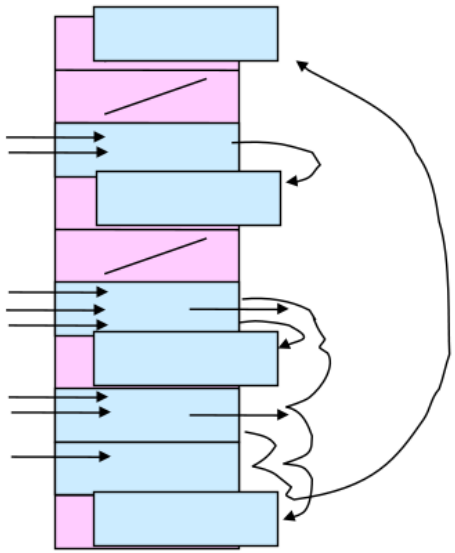


# Tabela de dispersão com encadeamento interno



- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, . . . ).

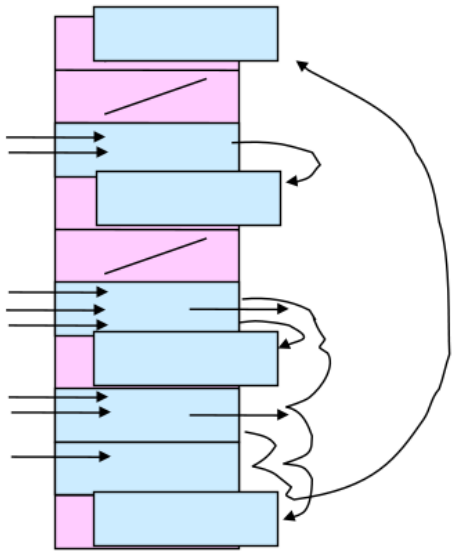


# Tabela de dispersão com encadeamento interno



- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, . . . ).

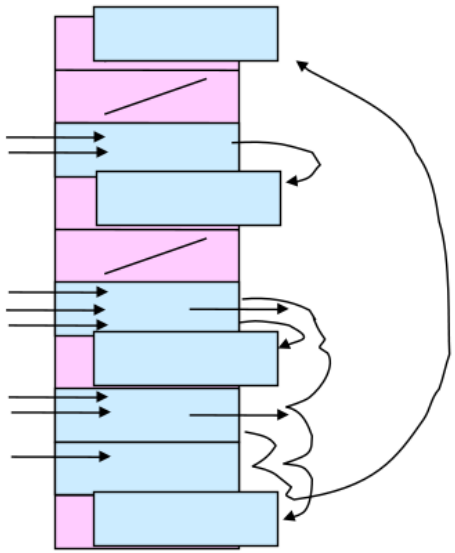


# Tabela de dispersão com encadeamento interno



- No mínimo, o tamanho da tabela tem de ser igual ao número máximo de elementos a armazenar.
- É usual sobredimensionar-se a tabela de forma a manter $\alpha < 0.7$:
  - O objectivo é minimizar o tempo despendido com a resolução das colisões.

- Resolução de Colisões:
  - $i_0 = h(k)$
  - se posição $i_j$ ocupada, então tentar:
  - $i_{j+1} = (i_j + c) \% m$
  - e repetir até encontrar uma posição livre.
  - o valor $c$ pode ser constante (pesquisa linear), ou seguir outra estratégia (quadrática, . . . ).



# *Tabela de dispersão com encadeamento interno*: exemplo



- $h(k) = k \,\%\, m$ com $m = 5$ e $k \in [0; 99]$

*insert(2)*

| | key | data |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | 2 | … |
| 3 | | |
| 4 | | |

*insert(21)*

| | key | data |
|---|---|---|
| 0 | | |
| 1 | 21 | … |
| 2 | 2 | … |
| 3 | | |
| 4 | | |

*insert(34)*

| | key | data |
|---|---|---|
| 0 | | |
| 1 | 21 | … |
| 2 | 2 | … |
| 3 | | |
| 4 | 34 | … |

*insert(54)*

| | key | data |
|---|---|---|
| 0 | 54 | … |
| 1 | 21 | … |
| 2 | 2 | … |
| 3 | | |
| 4 | 34 | … |

***Colisão:*** índice #4

***(4 + 1) mod 5 = 0***

# *Tabela de dispersão com encadeamento interno*: exemplo



- $h(k) = k \,\%\, m$ com $m = 5$ e $k \in [0; 99]$

*insert(2)*

| | key | data |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | 2 | … |
| 3 | | |
| 4 | | |

*insert(21)*

| | key | data |
|---|---|---|
| 0 | | |
| 1 | 21 | … |
| 2 | 2 | … |
| 3 | | |
| 4 | | |

*insert(34)*

| | key | data |
|---|---|---|
| 0 | | |
| 1 | 21 | … |
| 2 | 2 | … |
| 3 | | |
| 4 | 34 | … |

*insert(54)*

| | key | data |
|---|---|---|
| 0 | 54 | … |
| 1 | 21 | … |
| 2 | 2 | … |
| 3 | | |
| 4 | 34 | … |

***Colisão:*** índice #4

***(4 + 1) mod 5 = 0***

# *Encadeamento externo versus interno*

- Tabela de dispersão com encadeamento externo:
  - Não tem limite rígido do número de elementos.
  - Desempenho degrada suavemente à medida que o factor de carga aumenta.
  - Não desperdiça memória com dados que ainda não existem.
- Tabela de dispersão com encadeamento interno:
  - Não precisa de guardar apontadores de uns elementos para os outros.
  - Não perde tempo a alocar nós sempre que chega um novo elemento.
  - Toda a memória é alocada no início. Não requer alocação dinâmica.
  - Especialmente adequado quando os elementos são de pequena dimensão.
- Na prática, e para a maior parte das situações, estas diferenças são marginais.