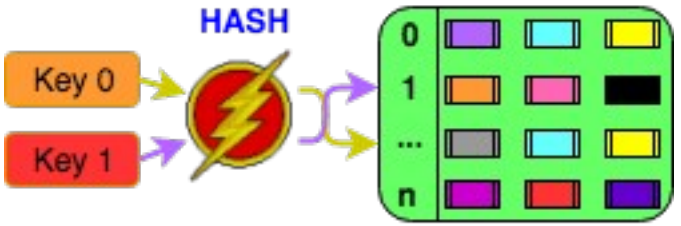
Pesquisa e Ordenação de Dados

Unidade 5.2:

Tabelas Hash



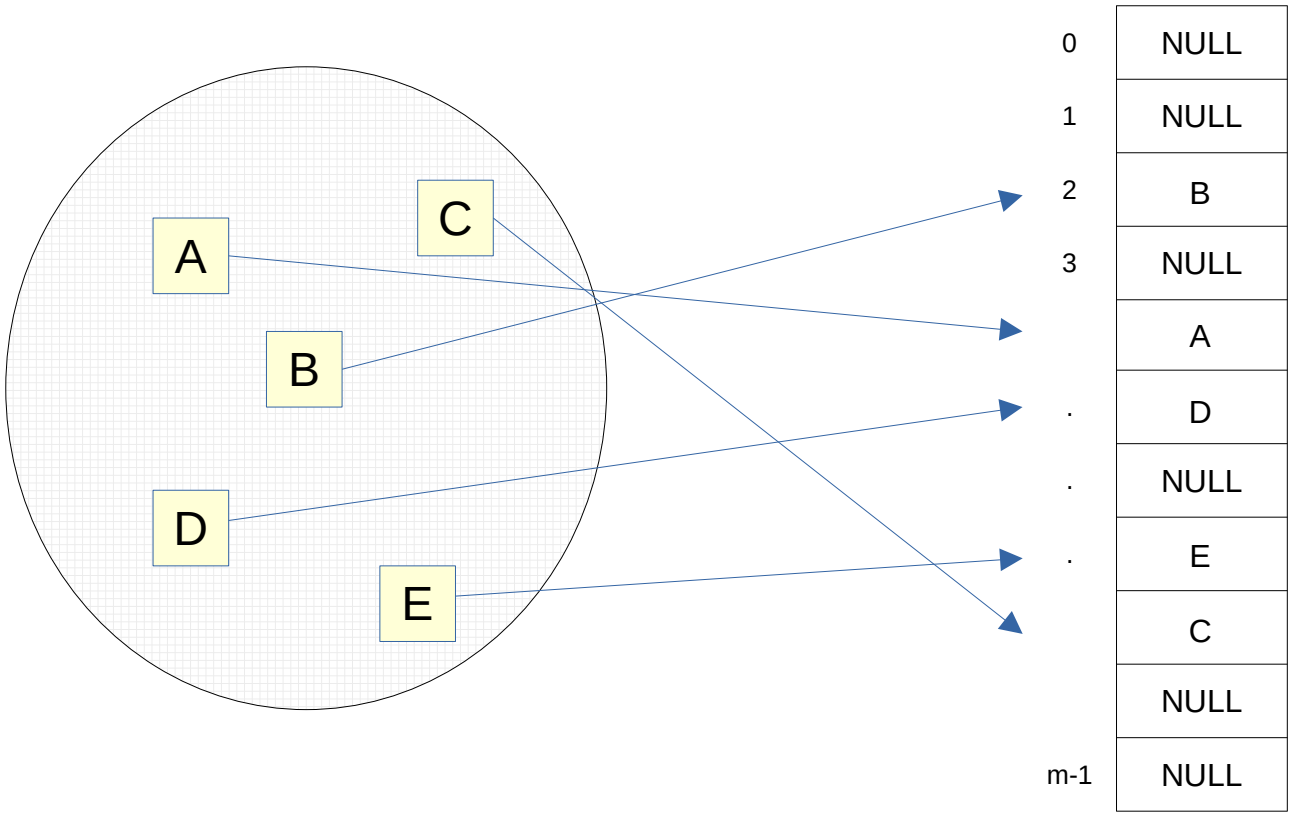
Tabelas Hash

- Diversos métodos de busca vistos funcionam através de comparações de chaves.
  - Busca binária: requer que os dados estejam ordenados
    - Custo do melhor caso da ordenação: O(n log n)
    - Custo da busca: O(log n)
- Tabelas Hash: permitem acesso direto ao elemento procurado, sem comparações de chaves e sem necessidade de ordenação
  - Custo O(1)

Tabelas Hash

- Tabela de Dispersão ou Tabela de Espalhamento
  - Estrutura de dados capaz de armazenar uma ou mais chaves (e seus valores associados) em um vetor
    - **Chave**: parte da informação que compõe o elemento a ser armazenado
- Utiliza uma **função** para definir posição de cada chave na tabela (ou seja, espalhar os dados pela tabela)
- Suporta as mesmas operações que as listas sequenciais (inserção, remoção, busca), porém, de forma mais eficiente.
- Elementos ficam dispostos de forma não ordenada

Tabelas Hash

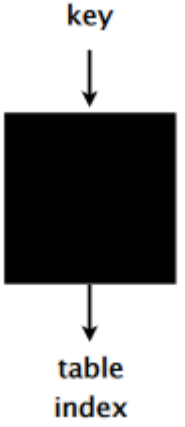


Tabelas Hash

- A implementação de uma tabela hash considera o mapeamento do conjunto de N chaves em um **vetor** de tamanho M > N.
  - Cada posição do vetor é também chamada de *bucket* ou *slot*.
- **Função de hashing**
  - transforma cada chave em um inteiro equivalente a um dos índices da tabela hash.
    - h(chave) → índice
  - usamos então este índice para armazenar/buscar o elemento no vetor.

Função de hashing

- A função de hash executa a transformação do valor de uma chave em um índice de vetor, por meio da aplicação de operações aritméticas e/ou lógicas.
- Os valores das chaves podem ser numéricos, alfabéticos ou alfanuméricos (a função irá converter o que não é número).
- Portanto, cada chave deve ser mapeada para um inteiro entre 0 e M-1 (para uso como índice do vetor de M posições).



Função de hash

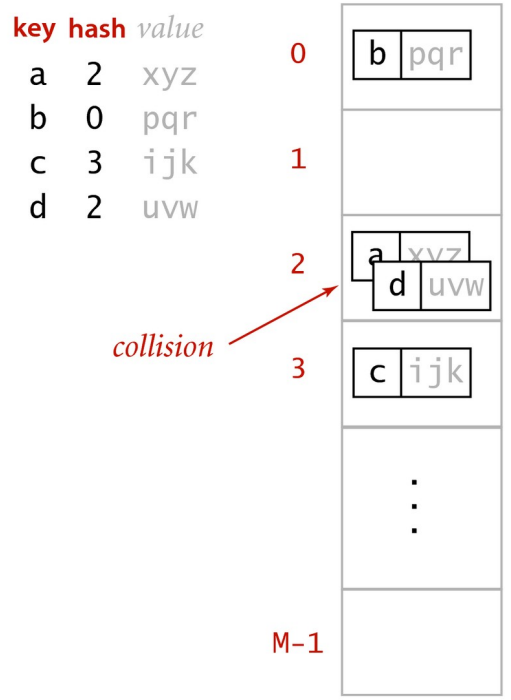
- Uma função hash possui 2 requisitos:
  - deve ser eficientemente computável, pois será utilizada em todas as operações sobre a tabela hash;
  - cada índice de tabela é igualmente provável de ser obtido. Ou seja, há uma boa probabilidade de "espalhamento" das informações na tabela.

Função de hash

- Uma função hash geralmente é formada por 2 partes:
  - criação do valor de hash: consiste em transformar uma chave k em um número inteiro, ainda que não esteja no intervalo [0, M-1];
  - função de compressão: transforma o número inteiro em um valor que esteja no intervalo [0, M-1].
- Idealmente, M deve ser um número primo, o que fará com que haja menos colisões.
  - É muito comum o uso do método da divisão: k mod M.

Colisões

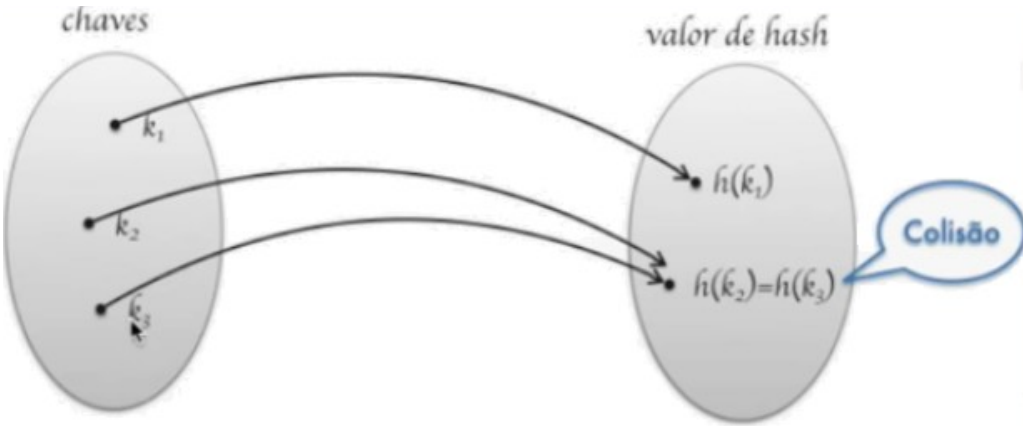
- Uma colisão ocorre quando a função de hash gera o mesmo valor para 2 ou mais chaves.
- Possíveis causas:
  - o número de chaves a armazenar é maior do que o tamanho da tabela;
  - a função de hash utilizada não produz uma boa distribuição (espalhamento).



Hashing: the crux of the problem

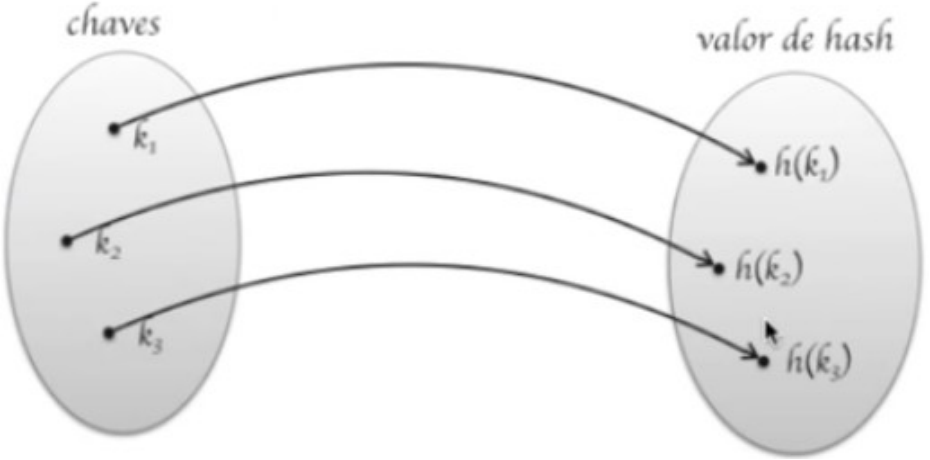
Colisões

- É sempre preferível evitar, pois degrada o desempenho.
  - Se todas as chaves colidem, o desempenho da busca pode cair para O(n).
- Quanto melhor a função, melhor a dispersão e menor a probabilidade de colisões.



Colisões

- Hashing perfeito
  - para cada chave diferente, é obtido um valor de hash diferente.
    - situação muito específica, como quando todas as chaves são previamente conhecidas



Colisões

- Exemplo: inserir um conjunto de chaves em uma tabela hash de tamanho 11, usando como função de hash h(k) = k mod M.

```
chaves = {7, 17, 36, 100, 106, 205}

h(k) = k mod M
```

  - h(7) = 7
  - h(17) = 6
  - h(36) = 3
  - h(100) = 1
  - h(106) = 7 (colisão)
  - h(205) = 7 (colisão)

Tratamento de Colisões

- Endereçamento aberto (*open addressing*)
- Encadeamento separado (*separate chaining*)

Endereçamento Aberto

- Todas as chaves são adicionadas à própria tabela, sem nenhuma estrutura de dados auxiliar.
- Em caso de colisão, é necessário procurar uma nova posição para a chave a ser inserida.
  - sondagem linear
  - sondagem quadrática
  - hashing duplo
- Vantagem: recuperação mais rápida (dados estão no próprio vetor). Não utiliza ponteiros.
- Desvantagem: custo extra de calcular a posição. Busca pode se tornar O(n) quando todas as chaves colidem.

Endereçamento Aberto

- **Sondagem linear**
  - Consiste em procurar a próxima posição livre do vetor.
  - Para isso, aplicamos uma função de hash $h_j(k)$ que contém a função $h(k)$.
    - O valor de j será 0, 1, 2, ….
  - A função é aplicada sucessivamente até que algum j resulte em um valor correspondente a uma posição livre.

$$h_j(k) = (h(k) + j) \bmod M$$

- Assim, enquanto não há colisão, $h_j(k)$ vai resultar o próprio $h(k)$ (pois j é zero).
  - A partir do momento em que há colisão, incrementamos j e tentamos alocar o valor. Se ainda há colisão, incrementamos j novamente e assim por diante, até que encontremos uma posição livre.
- Desvantagem: quando há muita colisão, acaba gerando um agrupamento consecutivo de dados (*cluster* primário), o que degrada o desempenho da inserção e da busca.

Endereçamento Aberto

- **Sondagem linear**

```
chaves = {7, 17, 36, 100, 106, 205}
h(k) = k mod M
hj(k) = (h(k) + j) mod M
```

  - $h_0(106) = (h(106) + 0) \bmod 11 = (7 + 0) \bmod 11 = 7$ (colisão)
  - $h_1(106) = (h(106) + 1) \bmod 11 = (7 + 1) \bmod 11 = 8 \bmod 11 = 8$ (nova posição)

  - $h_0(205) = (h(205) + 0) \bmod 11 = (7 + 0) \bmod 11 = 7$ (colisão)
  - $h_1(205) = (h(205) + 1) \bmod 11 = (7 + 1) \bmod 11 = 8 \bmod 11 = 8$ (colisão)
  - $h_2(205) = (h(205) + 2) \bmod 11 = (7 + 2) \bmod 11 = 9 \bmod 11 = 9$ (nova posição)

0	
1	100
2	
3	36
4	
5	
6	17
7	7
8	106
9	205
10	

Endereçamento Aberto

- **Sondagem quadrática**
  - A sondagem quadrática utiliza um valor de j^2 para evitar a formação dos *clusters* primários. A posição da chave na tabela é dada pela função de *hash*:

$$h_j(k) = (h(k) + j^2) \bmod M$$

  - Este método também cria um padrão de agrupamento chamado de agrupamento secundário, no qual o conjunto de posições ocupadas "salta" no arranjo de maneira pré-determinada (por causa j^2), o que também prejudica o desempenho das principais operações em tabelas hash.

Endereçamento Aberto

- **Sondagem quadrática**

  $chaves = \{7, 17, 36, 100, 106, 205\}$

  $h(k) = k \bmod M$

  $h_j(k) = (h(k) + j^2) \bmod M$

  - $h_0(106) = (h(106) + 0^2) \bmod 11 = (7 + 0) \bmod 11 = 7$ (colisão)
  - $h_1(106) = (h(106) + 1^2) \bmod 11 = (7 + 1) \bmod 11 = 8 \bmod 11 = 8$ (nova posição)

  - $h_0(205) = (h(205) + 0^2) \bmod 11 = (7 + 0) \bmod 11 = 7$ (colisão)
  - $h_1(205) = (h(205) + 1^2) \bmod 11 = (7 + 1) \bmod 11 = 8 \bmod 11 = 8$ (colisão)
  - $h_2(205) = (h(205) + 2^2) \bmod 11 = (7 + 4) \bmod 11 = 11 \bmod 11 = 0$ (nova posição)

0	205
1	100
2	
3	36
4	
5	
6	17
7	7
8	106
9	
10	

Endereçamento Aberto

- **Hashing duplo**
  - utiliza uma segunda função de hash para gerar um resultado diferente no que se refere ao tamanho do passo.

$$h_j(k) = (h(k) + j * h'(k)) \bmod M$$

  - A função de hash secundária $h'(k)$ não deve ser a mesma função de hash primária e também nunca deve produzir zero (do contrário, não haverá passo).

Endereçamento Aberto

- **Hashing duplo**

  `chaves = {7, 17, 36, 100, 106, 205}`

  `h(k) = k mod M`

  $h_j(k) = (h(k) + j * h'(k)) \bmod M$

  `h'(k) = q – (k mod q),` onde `q` é primeiro número primo menor que o tamanho do vetor (no caso, 7).

- Vamos calcular `h'(k)` para os dados que tiveram colisão:
  - h’(106) = 7 – (106 mod 7) = 7 – 1 = 6
  - h’(205) = 7 – (205 mod 7) = 7 – 2 = 5

0	
1	100
2	
3	36
4	
5	
6	17
7	7
8	
9	
10	

Endereçamento Aberto

- **Hashing duplo**
- Agora, calculamos $h_j(k)$ para os dois valores que tiveram colisão:
  - $h_0(106) = (h(106) + 0 * h'(106)) \bmod 11 = (7 + 0*6) \bmod 11 = 7 \bmod 11$ (colisão)
  - $h_1(106) = (h(106) + 1*h'(106)) \bmod 11 = (7 + 1*6) \bmod 11 = 13 \bmod 11 = 2$ (nova posição)
  - $h_0(205) = (h(205) + 0*h'(205)) \bmod 11 = (7 + 0*5) \bmod 11 = 7$ (colisão)
  - $h_1(205) = (h(205) + 1*h'(205)) \bmod 11 = (7 + 1*5) \bmod 11 = 12 \bmod 11 = 1$ (colisão)
  - $h_2(205) = (h(205) + 2*h'(205)) \bmod 11 = (7 + 2*5) \bmod 11 = 17 \bmod 11 = 6$ (colisão)
  - $h_3(205) = (h(205) + 3*h'(205)) \bmod 11 = (7 + 3*5) \bmod 11 = 22 \bmod 11 = 0$ (nova posição)

0	205
1	100
2	106
3	36
4	
5	
6	17
7	7
8	
9	
10	

Operações com Endereçamento Aberto

- **Inserção**: calcular a posição $h_j(k)$ e, se esta estiver ocupada, verificar a posição $h_1(k)$, $h_2(k)$, e assim por diante.
  - Melhor caso: O(1), quando na primeira tentativa a posição já está livre.
  - Pior caso: O(n), quando todas as chaves são mapeadas para posições ocupadas.

Operações com Endereçamento Aberto

- **Busca**: calcula-se o valor de $h_0(k)$. Se a chave não for encontrada nesta posição, realizamos novas tentativas de localizá-la em $h_1(k)$, $h_2(k)$, etc. A busca termina quando:
  - a chave é encontrada;
  - uma posição vazia (NULL) é encontrada;
  - o número de sondagens realizadas é igual ao tamanho da tabela ($j=m$)

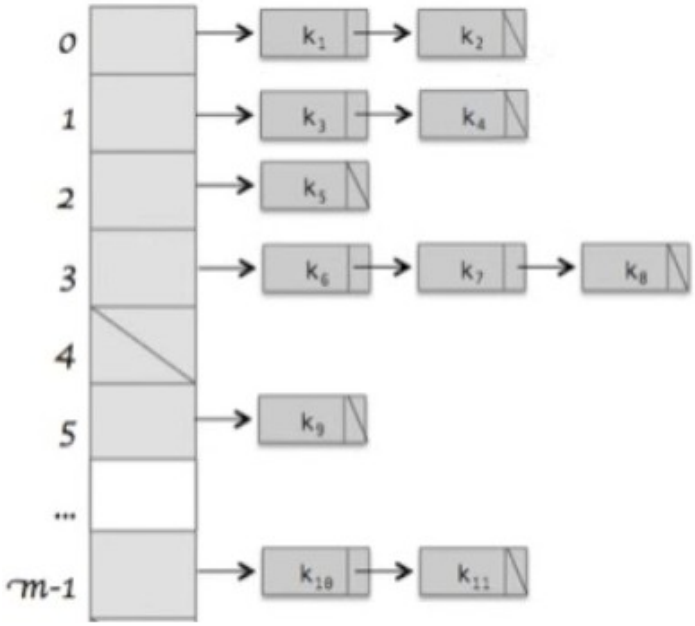
  - Melhor caso: O(1), que ocorre quando a chave está na posição $h_0(k)$.
  - Pior caso: O(m), pois terá que testar todas as posições até encontrar a chave.

Operações com Endereçamento Aberto

- **Remoção**: calcula-se o valor de hash $h_j(k)$ para identificar a primeira posição do vetor onde o registro com a chave a ser removida poderia se encontrar.
  - Se a posição estiver vazia, termina a busca.
  - Se a posição estiver ocupada por outro registro (colisão), procura a próxima posição;
  - Se a posição estiver ocupada pelo registro a ser removido:
    - marca a posição como excluída (não usar null para não prejudicar a busca das demais chaves que colidiram)
  - Melhor caso: O(1), quando encontra na primeira tentativa
  - Pior caso: O(m), pois pode ser necessário testar todas as posições para encontrar a chave.

Encadeamento Separado

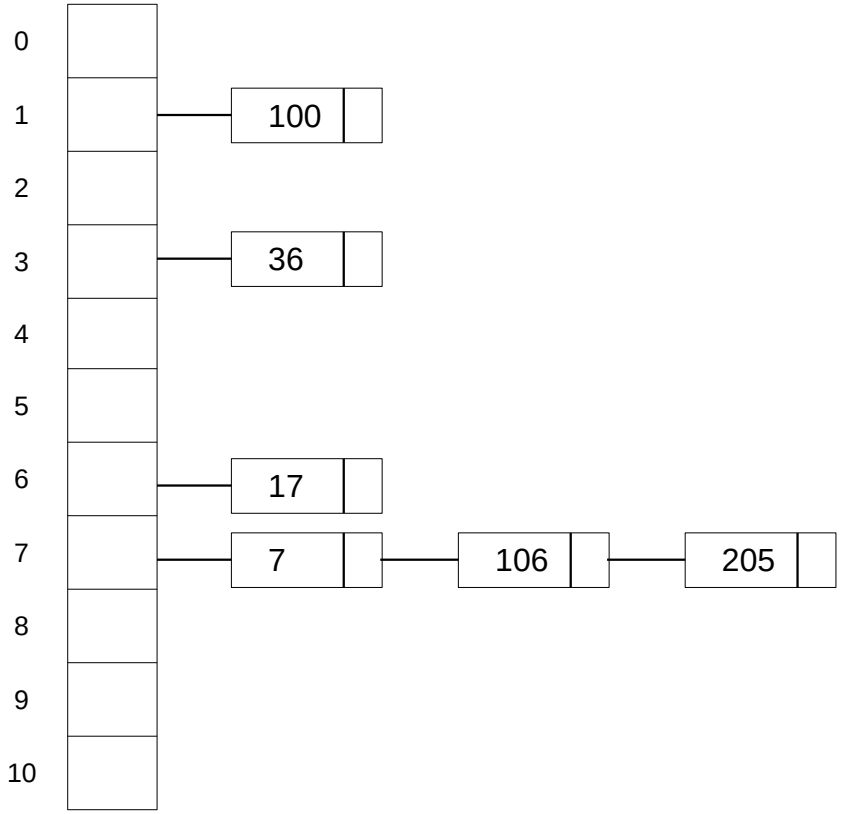
- Hashing aberto
- Cada posição da tabela aponta para o início de uma lista encadeada.
- Todas as chaves que colidem são armazenadas na respectiva lista encadeada (no início ou ao final).
  - Cada valor armazenado deve, portanto, possuir um ponteiro para o próximo elemento da lista.
- Necessita de memória adicional (o que não ocorre no endereçamento aberto).



Encadeamento Separado

- `chaves = {7, 17, 36, 100, 106, 205}`
- `h(k) = k mod M`

- Temos que:
  - h(7) = 7
  - h(17) = 6
  - h(36) = 3
  - h(100) = 1
  - h(106) = 7 (colisão)
  - h(205) = 7 (colisão)



Operações com Encadeamento Separado

- **Inserção**: a chave é inserida no final (ou início) da lista que se encontra na posição h(k)
  - Melhor caso / Pior caso: sempre O(1) (se a lista não for ordenada).

Operações com Encadeamento Separado

- **Busca**: calcular h(k) e buscar a chave na lista encadeada cujo início que se encontra na posição tabela[h(k)].
  - Melhor caso: O(1), quando buscamos a chave que se encontra na primeira posição da lista.
  - Pior caso: O(n), quando todas as N chaves se encontram na mesma posição do vetor da tabela hash, ou seja, em uma única lista encadeada, e ainda, quando a chave procurada se encontra na última posição da lista encadeada.

Operações com Encadeamento Separado

- **Remoção**: semelhante à busca, pois primeiro precisamos buscar a chave para depois remover.
  - Melhor caso: O(1), quando o elemento a ser removido é o primeiro da lista encadeada.
  - Pior caso: assim como na busca, quando todas as chaves estão em uma única lista encadeada e a chave buscada está na última posição. Seria necessário que percorrer as n posições até encontrar a chave e removê-la, o que leva O(n).