1. **O que é uma Tabela *Hash*?**

A busca linear peca em um grave aspecto. Embora seja fácil de implementar, não é muito eficiente, visto o tempo gasto ao percorrer uma lista sequencial no pior caso será O(log n). Infelizmente, as árvores binárias de busca só conseguem ser implementadas em sequências cujas chaves sejam distribuídas de forma ordenada, logo, não resolveriam esse problema.

Para solucionar o caso foi criado um algoritmo não baseado em comparação que provê operações de busca mais eficientes – as Tabelas *Hash*. Python, por exemplo, utiliza-se desta técnica para sua implementação de dicionários.

As tabelas de dispersão (*hash*) consistem em um tipo de estrutura de dados que associam chaves de pesquisa a valores, tendo por objetivo fazer buscas rápidas e obter os valores desejados a partir de chaves. Desta forma, cada posição da tabela possuirá um índice associado a uma chave e esta chave possuirá um valor.

1. ***Hashing***

*Hashing* é o processo de mapear uma chave de busca para um número limitado de índices de uma lista com o objetivo de prover acesso direto a estas chaves, de forma rápida e concisa. As chaves são armazenadas em uma outra lista chamada tabela *hash* (ou de dispersão) e uma função de dispersão é associada a esta tabela. A função converte (mapeia) as chaves de busca em entradas específicas na tabela.

Tomemos como base as chaves de busca a seguir e uma tabela de dispersão T que contenha M = 13 elementos.

**765, 431, 96, 142, 579, 226, 903, 388**

Poderemos então definir uma função simples de dispersão h(.) que mapeie as chaves em entradas da tabela *hash*:

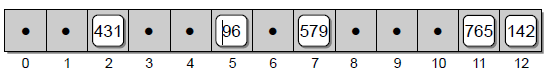
***h(chave) = chave % M***

O resto da divisão da chave pelo tamanho da tabela gerará sempre um valor válido entre os índices da tabela de dispersão. Desta forma, para adicionar chaves à tabela *hash*, aplicamos a função de dispersão às chaves desejadas, obtendo assim a posição em que elas deverão ser armazenadas. Por exemplo, aplicando a função à chave 765, obteremos o resultado 11, indicando que ela deverá ser armazenada na posição 11 da tabela de dispersão. Repetindo o procedimento com as próximas chaves também obteremos suas posições:

**h(431) => 2 h(96) => 5 h(142) => 12 h(579) => 7**

Observe que os resultados são valores de índice existentes na tabela de dispersão , não apresentando nenhum que não seja abrangido pela mesma. Abaixo, a forma como ficaria a tabela *hash* após a inserção das chaves citadas em suas posições obtidas pela formula de

dispersão.

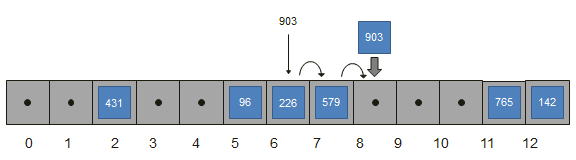


1. **Colisões**

Quando, após o processo de *Hashing,* duas chaves resultarem em um mesmo índice haverá uma colisão. Ou seja, a função de dispersão tentará relacionar duas chaves ao mesmo índice, resultando em um conflito. Quanto mais cheia uma tabela *hash* estiver, mais colisões haverão. Nestes casos, como proceder?

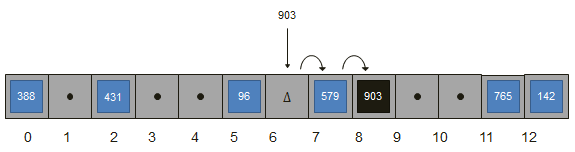
**3.1 – Sondagem Linear**

A solução mais utilizada chama-se Sondagem Linear, e consiste, basicamente, em tentar inserir a chave e, obtendo um conflito, saltar até o próximo índice vazio para então realizar a inserção.

No exemplo abaixo, tenta-se inserir a chave 903, que dividida por 13 terá índice 6, este, já ocupado pela chave 226. Desta forma, são realizados dois saltos até encontrar o próximo índice vazio e então inserir.

Durante uma busca, o mesmo procedimento será utilizado. Caso se esteja procurando a chave 903, a função *hash* será aplicada e, encontrando um valor diferente do pesquisado para aquele índice, serão realizados saltos entre os índices ocupados até encontra-lo. Caso se chegue até o fim da lista, a mesma será percorrida do início até a posição de busca iniciada. Não encontrando a chave, None será retornado. Para a situação em que se encontre um índice vazio, seja ele o inicial ou outro durante a varredura, None também será retornado. Mas e se o índice tiver sido removido?

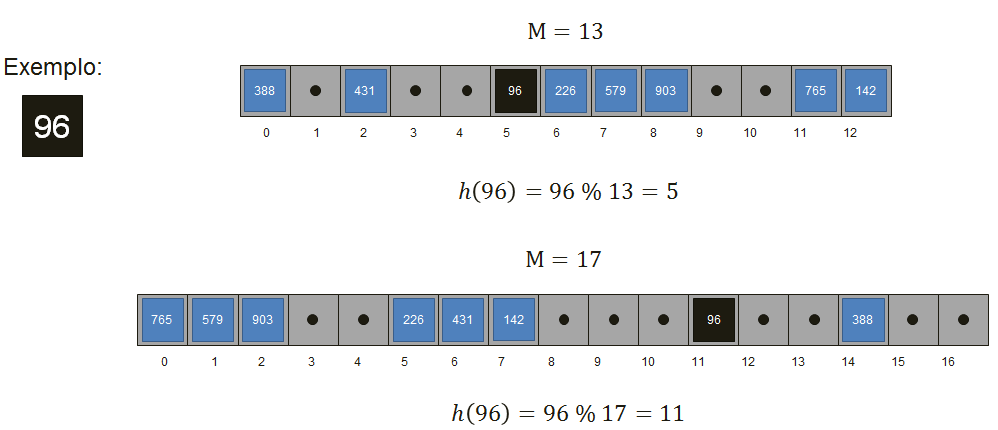
A remoção funciona da mesma forma que a busca e a inserção, ou seja, é realizada uma divisão entre a chave e o tamanho da tabela de dispersão, seu resto será o índice usado, neste caso, removido. Suponha, porém, que houve uma colisão na inserção como no exemplo acima e que logo após, foi removida a chave 226. Ao tentarmos remover a chave 903, a busca se deparará imediatamente com um índice vazio. Para resolver este problema, durante o processo de remoção é deixado um marcador específico no lugar da chave apagada e, durante a busca, se o índice contiver este marcador, será saltado para o próximo. Durante a inserção, porém, o marcador é substituído por qualquer chave que vá ocupar o índice.

Neste exemplo é realizada a remoção referida no parágrafo anterior. O delta representa um marcador deixado no lugar da chave previamente removida.

**3.2 – *Clustering* e *Rehashing***

Quanto mais chaves inseridas, mais colisões haverão, desta forma mais será aplicada a Sondagem Linear e mais preenchida a tabela de dispersão ficará. Um cluster é um aglomerado sequencial de chaves associadas a índices sem intervalos entre elas, porém, esta formação não é desejada, já que muitas chaves sem intervalos resultarão em um aumento do tempo de busca, o que é justamente o contrário do que desejamos ao aplicar o procedimento de *Hashing*. Além disso, ao inserirmos uma chave, mesmo que no primeiro índice de um cluster, ela será jogada para após a última posição do mesmo e, caso este seja o único espaço vazio entre este e outro cluster, geraremos um cluster ainda maior.

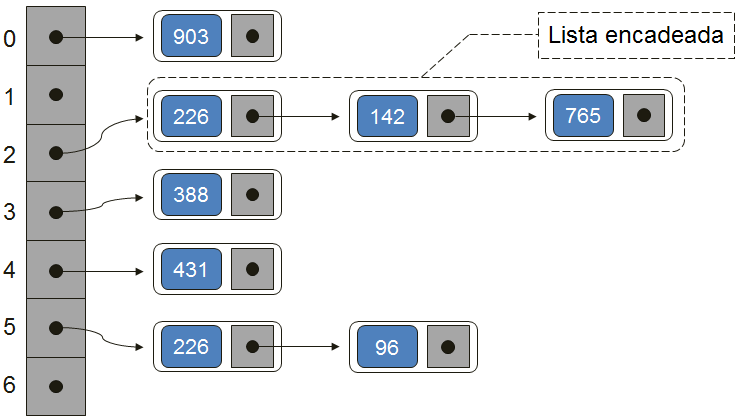
Para resolvermos este problema, utilizamos o procedimento conhecido como *Rehashing*, ou seja, redimensionamos o tamanho da tabela e redistribuímos seu conteúdo, aplicando novamente a função h(chave) = chave % M.

No exemplo abaixo redimensionamos a tabela que antes continha 13 índices para passar a conter 17, então, reaplicamos a função de *hash* para todas as chaves e as alocamos para seus respectivos lugares.

Diferente do exemplo, o recomendado é que se dobre a quantidade inicial de índices sempre que se vá realizar a função de *rehash*, bem como que se utilize sempre um número primo como tamanho da tabela. A primeira recomendação fará com que haja menores taxas de formação de clusters, a segunda, que existam menos divisores exatos para o tamanho da tabela, o que acarretará em menos índices conflituosos.

**3.3 – Encadeamento Separado**

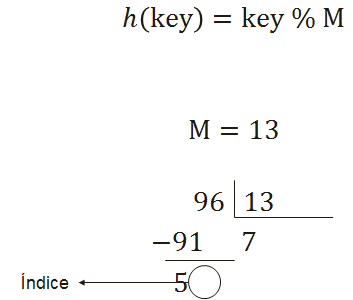
O encadeamento separado foi criado com a finalidade de eliminar todas as colisões de uma tabela de dispersão. Seu funcionamento consiste em unir a simples implementação de listas encadeadas às tabelas *hash*, de forma que, quando o resto da divisão de uma chave pelo tamanho da tabela resulta em um índice conflitante, será criada uma lista encadeada dentro daquele índice, com a primeira chave inserida referenciando a próxima, e assim sucessivamente

No exemplo abaixo foram adicionadas as chaves 142 e 765, e ambas deveriam ser alocadas ao índice 2. Deste modo, foi criada uma lista encadeada com a primeira chave do índice referenciando a próxima, em ordem de inserção.

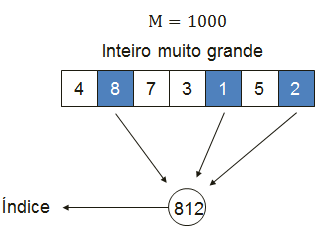
1. **Funções *Hash***

As funções de dispersão (*hash*) são responsáveis por gerar os índices em que serão inseridas as chaves. Todo o funcionamento das tabelas de dispersão depende delas, de forma que as mesmas têm de sempre apresentar computação simples, a fim de se obter resultados rápidos e índices não aleatórios, a fim de serem encontrados facilmente. Para existirem de forma eficiente, os tamanhos das tabelas devem ser números primos, como dito anteriormente e devem ser aplicadas apenas a inteiros, strings, ou mistos de inteiros e strings.

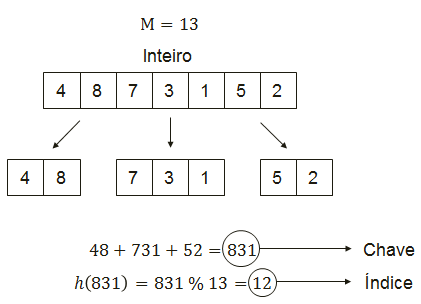
**4.1 – Função de Divisão**

É a função *hash* primordial, que foi largamente explorada até então. Consiste em realizar a divisão da chave pelo tamanho da tabela de dispersão e, a partir de seu resto, atribuir um índice à chave em questão.

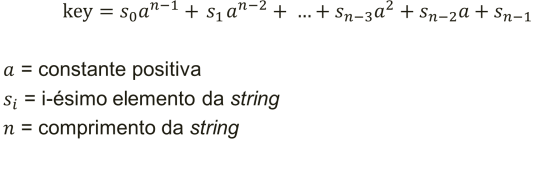
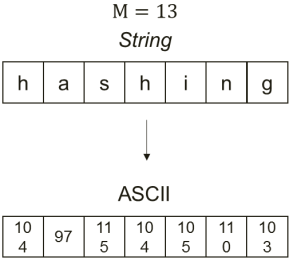
**4.2 – Função de Truncamento**

Quando se tem um inteiro muito maior que o tamanho da tabela, pode-se realizar a função de truncamento. Ela consiste em pegar partes deste inteiro e concatená-las diretamente no índice da tabela, como a seguir:

**4.3 – Função de Dobramento**

 Outra possibilidade para inteiros muito grandes é a função de dobramento. Ela consiste em repartir o inteiro e somar suas partes, gerando então uma chave que, após passada pela função de divisão, terá um índice a ser alocado.

**4.4 – *Hash* de *strings***

 Há duas formas de se realizar o *hash* de strings. A primeira consiste em converter cada caractere da string em sua representação em ASCII para então somar todos estes valores e gerar sua chave. A segunda consiste em aplicar as representações ASCII uma fórmula que gerará sua chave. Em ambas deverá ser aplicada a chave à função de divisão, obtendo seu índice.

