Hashing

As árvores de busca binária representam uma eficiente maneira de inserir, encontrar e deletar dados armazenados em um computador. Hash Tables (Tabelas hash) representam uma alternativa às árvores de busca binária, onde elas também são utilizadas com o propósito de inserir, encontrar e deletar dados.

Uma hash table de tamanho n é um vetor de células.

- Cada célula do vetor pode está vazia, contendo um único dado armazenado ou pode conter uma lista de dados. (dependendo da técnica de resolução de colisões que veremos mais à frente).

- Os dados são inseridos, buscados e deletados por intermédio de uma função hash a qual mapeia os dados para uma célula da hash table.

- A chave de um dado é um identificador único. Chaves tendem a ser números naturais (ex. números seriais, CPF) ou string de caracteres (ex. nome de pessoas).

- Uma função hash é um mapeamento h : K → {0,1,...n-1} do conjunto de dados de dados para o conjunto de índices de células. A função hashing assim representa uma maneira de codificar chaves de dados em um conjunto de números naturais.

- Duas chaves K_1 e K_2 pertencentes à K, colidirão quando $h(K_1) = h(K_2)$. O tratamento destas colisões é o que determina os vários tipos de hash.

O mais interessante é que com hash, conseguimos atingir todo e qualquer registro que desejarmos com uma complexidade média de O(1), ou seja é constante.

Exemplo 1. Dado uma tabela hash de tamanho 6, seja K o conjunto dos primeiros nomes de pessoas (ignorando qualquer caractere não-alfabético) e defina h : K → {0,1,...4} onde.

$$h(k) = \sum \alpha(cn) \bmod 6,$$

onde cn é o enésimo caracter de cada nome e α(cn) é a ordem alfabética de cn. Aplique a função hash nos nomes: Pedro, Clara e Maria.

Aplicando a função hash acima no nome ANA, teríamos então:
1mod6 + 14mod6 + 1mod6 = 1 + 2 + 1 = 4
A N A

Propriedades de uma boa função hash h:

- h deverá distribuir uniformemente as chaves por todo o conjunto de células da tabela hash. (deverá ser um gerador pseudorandom o qual tem uma distribuição uniforme).

- h deverá ter baixa complexidade, pois tanto na inserção, busca e deleção teremos que computar a função hash.

- h irá trazer o mapeamento para um conjunto finito de valores de tamanho n(tamanho da tabela hash), o uso da função mod(n) pode ser aplicado para garantir que os valores encontrados estarão dentro dos limites da tabela.

Exemplo 2. Considere a função hash:

```
int hash(char * s) {
   int i = 1;
   int sum = 0;

   sum = s[0];
   while(s[i] != '\0')
      sum = 37*sum + s[i++];
}
```

Esta função é uma boa função hash?

Tratamento de colisões

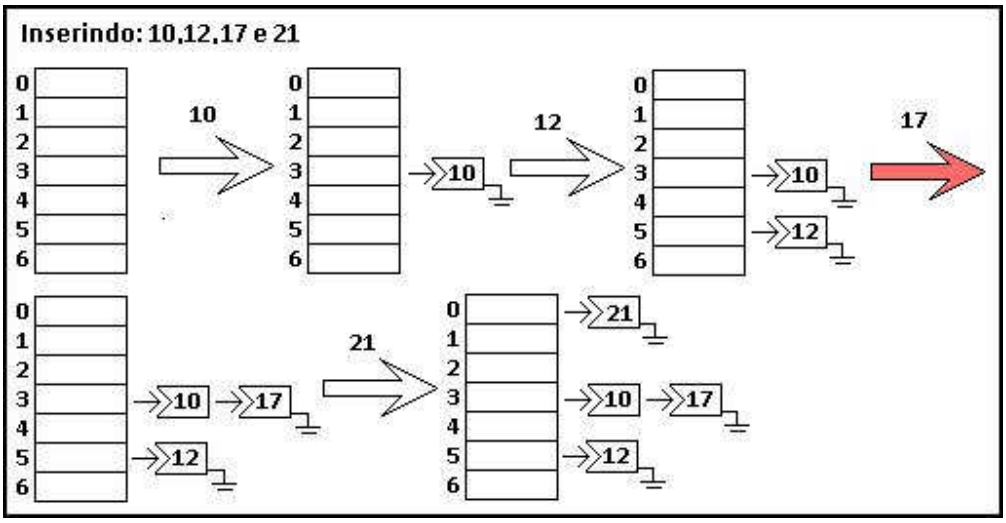
1. Encadeamento com listas separadas (*Separate chaining hashing)*

Cada célula da tabela hash consiste de uma lista encadeada de dados, em outras palavras, a célula n é a lista de todos os dados os quais tiveram como saída da função hash o valor n.

O encadeamento com listas separadas representa uma maneira simples de se tratar às colisões e pode ser eficiente desde que a lista não fique demasiadamente grande.

O fator de carga de uma tabela hash é definido como: $\lambda = m/n$ onde m é o números de itens a serem armazenados na tabela hash e n é o tamanho da tabela hash.

Exemplo: Função hash utilizada será: h(k)= Kmod(7)



No exemplo acima ficou explanado como é feita a inserção de dados utilizando a técnica de lista encadeada, as operações de busca e deleção são triviais. Caso você deseje resgatar os dados de sua tabela hash, basta aplicar a função hash na chave do dado que deseja encontrar e a função te retornará a lista encadeada onde o dado deverá está.

2. Encadeamento com alocação estática de espaços "listas misturadas" (*Coalesced hashing)*

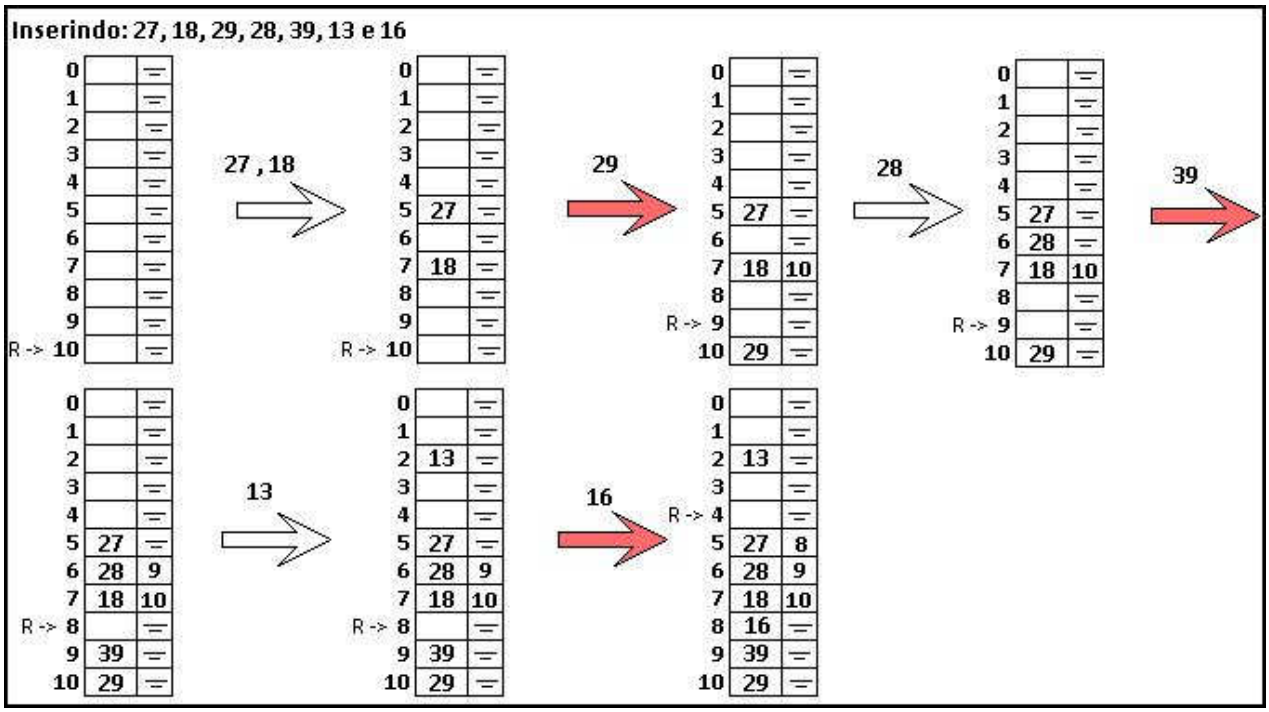
Esta técnica de resolução de colisões utiliza ponteiros para conectar os elementos de que possuem o mesmo valor aplicando-se a eles uma função hash.

Diferente da técnica acima se tem agora ponteiros pertencentes à tabela hash. Inicialmente todos estes ponteiros estão apontando para NULL.

Tem se também a presença de um ponteiro R, é um ponteiro onde aponta para uma posição vazia dentro da tabela hash, seu uso será entendido quando se mostrar um exemplo com o uso desta técnica.

Vamos apresentar um exemplo para facilitar o entendimento e depois faremos algumas considerações sobre a técnica.

Exemplo: Função hash utilizada será: h(k)= Kmod(11)



Os números 27 e 18 foram inseridos na tabela sem nenhum problema, quando tentamos inserir o 29 ocorreu uma colisão, então pela técnica, colocamos o 29 na posição apontada pelo ponteiro R e na posição inicial onde o 29 deveria ter ficado, fazemos com que o ponteiro aponte para a onde colocamos o 29. O mesmo processo ocorre com os outros números que foram inseridos.

O processo de busca por algum elemento se dar pela seguinte forma, nós devemos começar nossa busca pela célula retornada pela função hash aplicada na chave do elemento que desejamos encontrar, a busca finalizará quando encontramos uma célula com o ponteiro apontando para NULL ou quando encontrarmos o elemento que nós estamos procurando. Obviamente então, uma busca sem sucesso levará mais tempo que uma busca com sucesso com o uso desta técnica.

Note que o número de buscas para se encontrar um dado elemento na tabela hash, se dá em função da ordem na qual os elementos foram inseridos, os elementos inseridos primeiros serão sempre mais rápidos de se encontrarem do que os últimos.

O processo de deleção nesta técnica não e trivial, uma vez que se simplesmente remover os elementos que se deseja, pode acarretar em erros durante futuras buscas na tabela hash, uma vez que com a remoção se quebrou a “lista encadeada” presente na tabela hash. Um processo simples para deleção seria colocar no lugar de quem queremos remover, o elemento para qual ele aponta, este processo então tem que ser feito recursivamente até que encontremos o elemento que tem seu ponteiro apontado para NULL. (Consegue pensar em alguma outra maneira de fazer a remoção?).

Caso não tenha ocorrido nenhuma colisão durante a inserção de dados, a remoção fica trivial, bastando apenas ir até tabela hash e remover o elemento que deseja. Lembrando que após cada remoção deve-se atualiza a posição do ponteiro R caso necessário.

A principal desvantagem desta técnica é ter que armazenar espaço para os campos de ponteiro em cada célula da tabela hash, veremos a frente uma técnica que soluciona este problema.

3. Encadeamento aberto com busca seqüencial (*Linear probing)*

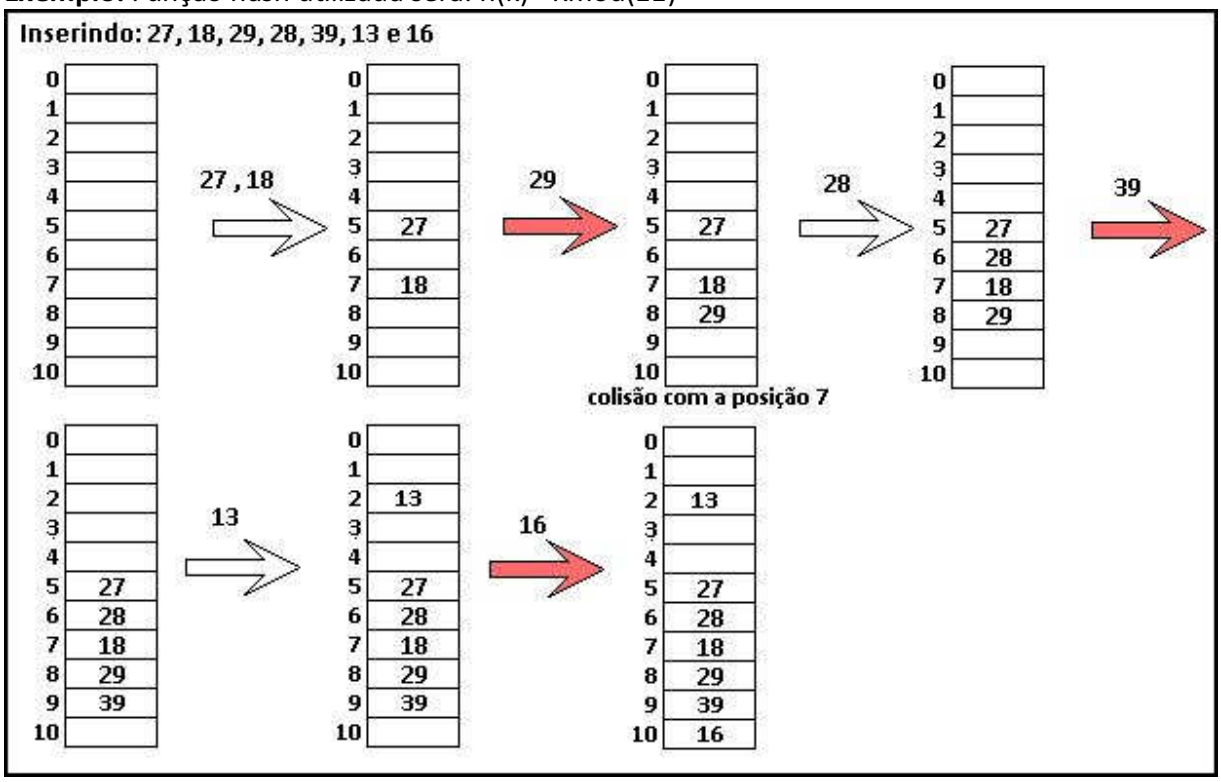
Como o próprio nome já diz, se no momento de se adicionar um elemento na tabela hash, a mesma estiver ocupada, então tentamos inserir na próxima posição, caso ela esteja cheia tenta então por na posição seguinte e assim por diante. Aí vem a seguinte pergunta: O que acontece quando eu atingir o maior endereço da tabela hash? Imagina que a mesma seja circular, você então voltará para a posição de menor endereço.

Para efetuar uma busca nesta tabela hash, é só seguir o mesmo processo da inserção, calculamos o hash do elemento que queremos encontrar na tabela hash, e vamos procurando a partir da posição inicial retornada pela função hash posição por posição até que se encontre o elemento ou até que se encontre uma célula vazia ou atingir novamente o endereço inicial de busca, o quais denotam que nossa busca foi infrutífera.

Obviamente, a performance deste algoritmo para uma busca sem sucesso é fraca, mas à frente veremos algoritmos mais eficientes para esta técnica.

Vamos mostrar um exemplo para facilitar o entendimento do processo.

Exemplo: Função hash utilizada será: h(k)= Kmod(11)



A grande desvantagem desta técnica de resolução de colisões é o grande número de buscas à tabela hash para se encontrar um registro. Devido a este problema esta técnica não é um método prático para resolução de colisões.

A deleção de registros com esta técnica requer bastante cuidado, pois a idéia é que depois que seja removido o registro desejado, seja capaz ainda de se encontrar todos os outros registros através do método de busca. Nós não podemos simplesmente remover o registro e marcar ele como NULL, desta maneira poderemos abrir um buraco durante uma seqüência de registros, o que comprometeria a busca.

Uma maneira simples de efetuar a deleção é pondo um marcador no local do registro removido, assim não comprometeria as buscas. Nas buscas caso encontremos este marcador, passaríamos por ele e continuaríamos a efetuar as buscas nas posições seguintes.

4. Duplo Hashing (*Linear quotient)*

A principal diferença entre a resolução de colisões pelo duplo hashing e busca linear, é que na técnica de duplo hashing, nós utilizamos uma variável de incremento ao invés de uma constante de incremento como na busca linear.

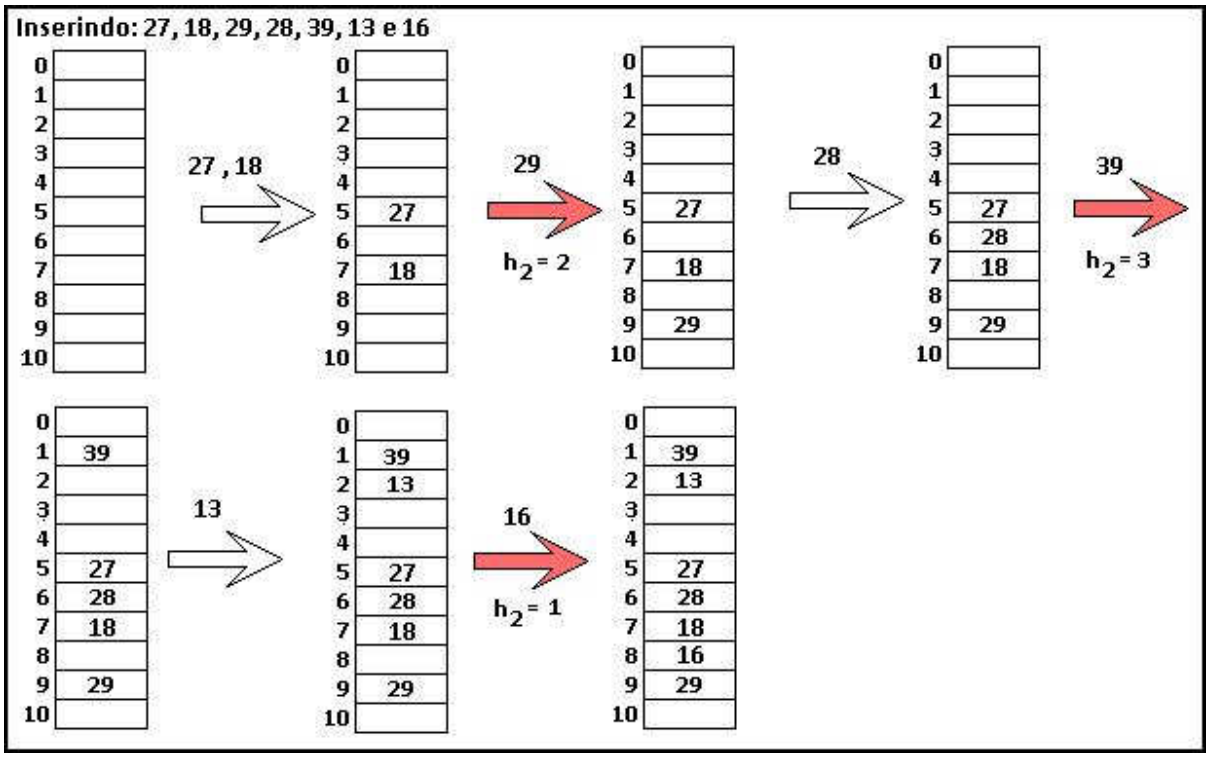
O objetivo do uso da variável é reduzir a quantidade de buscas para se alocar um novo elemento na tabela hash, caso o seu local primário de estádia esteja já ocupado. Reduzindo esta quantidade de buscas, conseqüentemente diminui a quantidade de buscas na hora de se encontrar este elemento na tabela hash.

No método duplo hashing, o incremento é uma função da chave a ser inserida, em outras palavras esta outra nova função pode ser vista como uma nova função hash, por isso o nome duplo hashing.

Uma das condições para que este método funcione sem problemas é que o tamanho da tabela hash seja de tamanho P, e que este P seja um número primo, caso contrário poderemos encontrar um grande problema. Consegue imaginar que tipo de problema seria este?

Para melhor entendimento da técnica, iremos demonstrar um exemplo do processo. Para o exemplo, iremos utilizar as seguintes funções hash.

$h_1(K) = K \bmod (11)$ e $h_2(K) = (K/11) \bmod (11)$



Observe na figura que no momento em que vamos inserir o 29, diferente da técnica de busca seqüencial o que levaria a tentar colocar o 29 na posição seguinte, aplicamos ao 29 a nossa segunda função hash, que retorna um valor de incremento, que neste caso foi o 2. Este processo ocorre com todos os outros elementos que sofreram colisões. Se mesmo assim após o uso do incremento o elemento tornar cair em uma nova célula já ocupada, o mesmo incremento é novamente utilizado, até que se chegue a uma célula vazia.

O mecanismo para determinar se uma busca foi sem sucesso é a mesma utilizada na técnica de busca linear, quando encontrarmos uma célula vazia, significa que nossa busca foi sem sucesso, ou quando retornamos ao endereço inicial de busca.

Pode se melhorar o método de duplo hashing, observando que o número de buscas para um determinado elemento na célula, depende do local onde estes registros foram armazenados.

Suponha que se deseja inserir a chave 67, ela terá como endereço baseado nas funções hash acima e na maneira de como ficou a nossa tabela hash, a posição 1 a qual já está ocupada com a chave 39. Então, aplicando a nossa segunda função hash, tentamos inserir nosso elemento 67 nas posições 7,2 e 8 até descobrirmos uma célula vazia na posição 3. Isto significa que 5 buscas serão necessárias para se encontrar o elemento 67. Se o 39 não estivesse na posição 1 da tabela hash, teríamos então ter inserido o 67 na nossa primeira tentativa.

Veremos este conceito melhor mais adiante, onde ele pode mover elementos já inseridos na tabela, com o intuito de diminuir a quantidade de buscas para se achar este elemento.

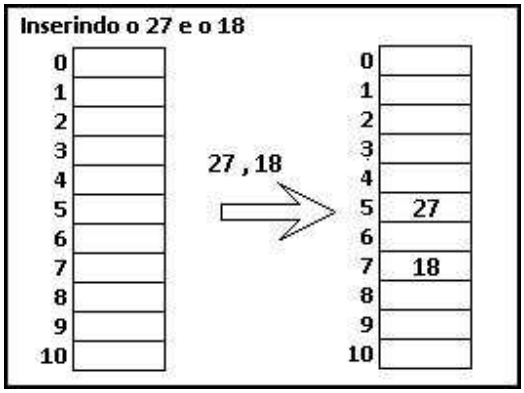
A remoção de um registro da tabela requer a utilização de um marcador assim como foi utilizado na técnica de busca seqüencial.

4. Método de Brent (*Brent method)*

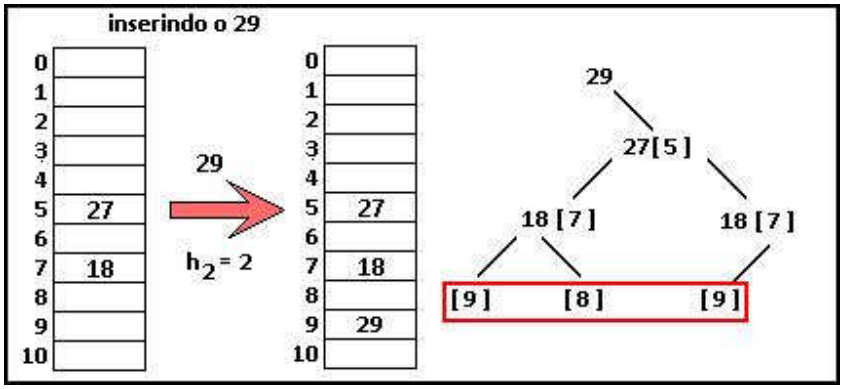
O método de Brent é um método de resolução de colisões que se preocupa com a ordem dos elementos inseridos e verifica se invertendo os elementos, melhora o número de buscas para se achar as chaves.

Alguns conceitos que serão necessários para o entendimento de Brent, chamaremos de 'i' a quantidade de buscas que tem que ser feita na tabela hash para se atingir o elemento A e chamaremos de 'j' a quantidade de buscas que tem que ser feita na tabela hash para se atingir o elemento B. A idéia é diminuir o valor de i + j, invertendo a posição das chaves.

Vamos a um exemplo passo a passo para que fique clara a idéia.
$h_1(K) = K \bmod (11)$ e $h_2(K) = (K/11) \bmod (11)$



Sem transtorno algum, os números 27 e 18 foram inseridos de forma trivial na tabela hash.



No momento da inserção do 29, ocorreu uma colisão, então por Brent temos o seguinte: podemos apenas efetuar **uma troca de posição somente** uma vez e procurar sobre todas as maneiras, aquela que nos dá a menor soma de i + j.

Para encontrarmos o 27 é somente uma busca na tabela hash, então i = 1.
Para encontrarmos o 29 segundo o método de duplo hashing sem o uso da técnica de Brent é 2, pois iremos conflitar com a posição 5 e 7 até chegarmos na posição do 29. Logo i + j = 3

Fizemos então uma árvore para explorar todas as combinações que poderiam ser feitas seguindo a regra de Brent. A raiz da árvore informa o número que desejamos inserir na tabela hash e nos outros nós estão os elementos que encontramos em cada posição da tabela hash.

Caso eu desça por um ramo a direita da árvore, ficará então denotado que foi realizado uma troca de posições e caso eu desça pelo ramos esquerdo eu aplico o segundo hash no elemento que eu desejo inserir procurando então um novo lugar para armazená-lo.

O nó interno mais alto da árvore indica a situação inicial do problema quando eu quero inserir o 29, no nó temos: 27 [5], significa que tentamos por o 29 na posição 5 mas lá tinha o número 27. Podemos então efetuar a troca entre o 29 e o 27 ou continuar a busca por uma nova posição para o 27 dentro da tabela hash.

Primeiramente vamos simular a situação de uma troca inicial, então será feita a troca entre o 29 e 27, com isto descemos para o ramo direito da árvore e aplicamos a segunda função de hash não mais no 29 mas, sim no 27. Aplicando esta segunda função de hash no 27 ela nos retorna 2, vamos então cair na célula 7 onde temos já armazenado o elemento 18. Como já efetuamos uma troca, não podemos descer mais uma vez pelo ramo direito da árvore agora só o esquerdo.

Então descendo pelo ramo esquerdo, aplicamos novamente a segunda função de hashing ao elemento 27 e atingimos assim a célula 9 da tabela hash a qual está vazia.

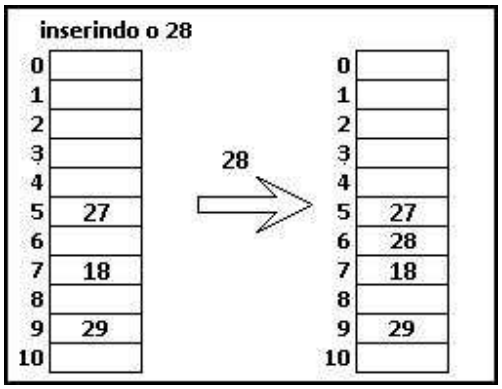
Subindo na árvore vamos simular outra situação, vamos simular uma situação onde a troca não ocorra entre o 29 e 27, mas sim entre o 29 e o 18. Então estamos no ramo 27 [5] na árvore, como não desejamos realizar a troca, descemos para o ramo esquerdo e aplicamos a segunda função de hash no 29, ela então retornará 2 e cairemos então na célula 7 onde temos o elemento 18, como ficou dito acima, iremos então efetuar a troca entre o 29 e o 18, assim então descemos pelo nó a direita da árvore e neste momento aplicamos a segunda função de hashing ao 18 e não mais ao 29.

Aplicando a função ela nos retornará 1, e então chegaremos a posição 8 da célula que está vazia.

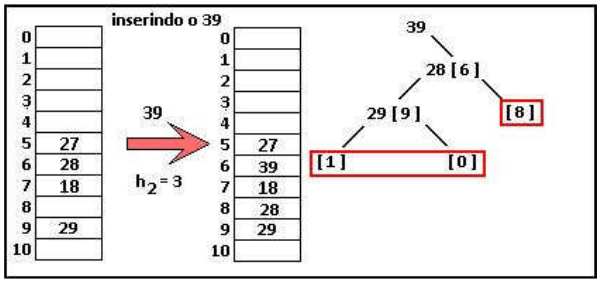
Por último vamos simular a situação onde não iremos fazer troca alguma, iremos realizar o procedimento que a técnica de duplo hashing simplesmente faria. Estamos então no nó 27 [5], descemos então para a esquerda, aplicando a função de hash ao 29, onde atingiríamos a posição 18 [7], onde novamente aplicaríamos a segunda função de hash e chegaríamos a uma posição vazia na tabela hash a posição 9.

Temos agora todas as 3 possibilidades pelo método de Brent, mas como vocês podem ver, todas as 3 possibilidades estão no mesmo nível da árvore, então não existe uma solução melhor do que a provida pelo método de duplo hashing, você pode escolher qualquer uma das 3 e continuar o processo de inserção.

Caso tivéssemos encontrado uma solução que se encontrasse em um nível mais acima que as outras, refletiríamos então na tabela hash a simulação feita até chegar nesta solução de melhor custo.



A inserção do 28 ocorrerá sem problemas, será feita de forma trivial.

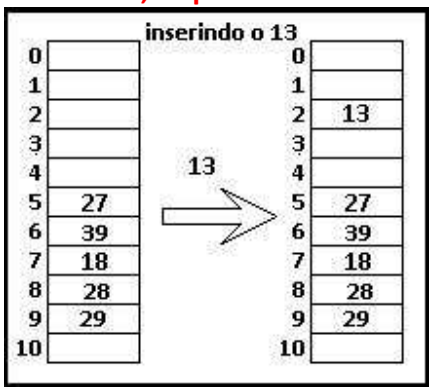


Diferente da inserção do 29, conseguimos uma solução com melhor custo quando trocamos o 39 pelo 28 e aplicamos a segunda função hash no 28.

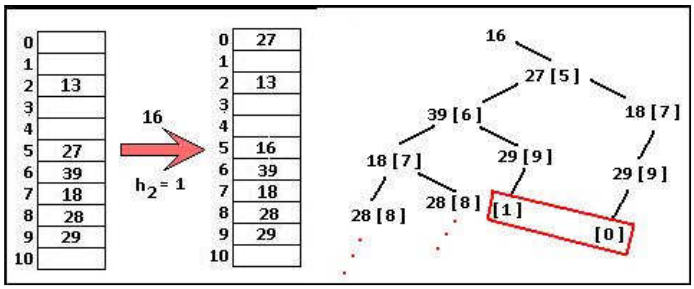
O custo que era inicialmente de 3, passou a ser 2.

É interessante pegar a explanação da árvore anterior e seguir nesta árvore para fixar o funcionamento da mesma.

OBS.: Em Brent não podemos descer mais que uma vez para o ramo direito, uma vez que descemos para o ramo direito da árvore, ou em outras palavras, efetuamos uma troca de elementos, só podemos então descer pelo lado esquerdo da árvore.



O elemento 13 é inserido também sem muitos problemas.



Pelo que se observa, encontramos duas soluções as quais são as de menor nível, uma delas é:

1. Trocar o 27 pelo 16, e aplicar a segunda função hash no 27.

2. Aplicar a segunda função hash uma vez ao 16, depois trocar o 16 com o 39, aplicar a segunda função hashing ao 39 duas vezes.

Veja que o restante da árvore não foi necessário ser desenhado pelo fato de que, qualquer outra

solução encontrada por aqueles caminhos, com certeza serão maior que as duas já encontradas.

Para seguir a bibliografia escolheremos a primeira opção descrita acima, mas nada impediria de escolhermos a segunda opção.

Apesar do método de Brent diminuir o número de buscas para os elementos pertencentes de uma tabela hash, ele nem sempre apresentará soluções ótimas. Um outro método que vamos estudar apresenta **SEMPRE** a melhor solução para aquela dada tabela hash e aquele dado conjunto de valores, é o método da árvore binária.

6. Método com árvore binária (*Binary Tree)*

O método funciona de forma idêntica ao método de Brent, a única diferença é que agora pode ser realizada mais do que um troca, podemos trocar agora quantas vezes for necessário para termos uma boa solução na diminuição do número de buscas para os elementos presentes na tabela hash.

Isto significa que agora na hora de criarmos a árvore de possibilidades, esta árvore sempre será completa, tendo assim agora uma gama maior de possibilidades a serem analisadas.

Fica como exercício fazer a inserção do exemplo dado no método de Brent só que agora usando o método da árvore binária. As funções de hash continuam as mesmas também do exemplo do método de Brent.

Uma das possíveis respostas é a seguinte:

Possível resposta da questão

0	
1	39
2	13
3	
4	
5	27
6	16
7	18
8	28
9	29
10	

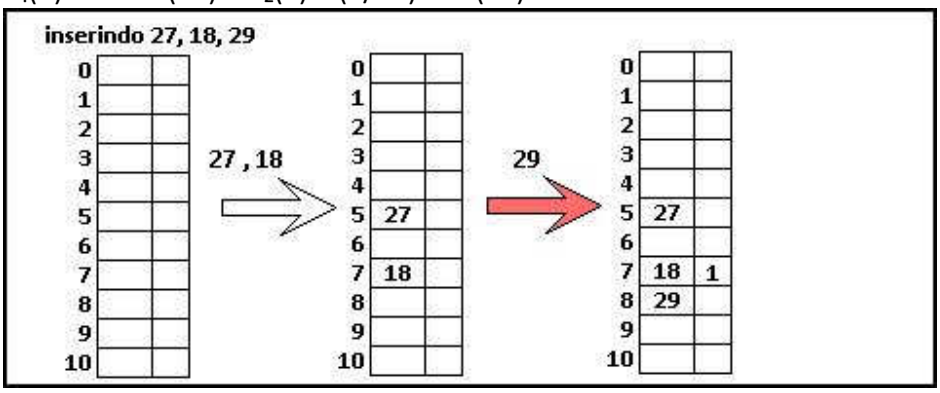
7. Pseudoponteiro - Cadeia Computada (*Computed Chaining)*

Nós consideramos dois tipos de procedimentos para resolver colisões até o momento, um deles usa um campo para ponteiros (por exemplo: coalesced hashing) e aqueles que não usam este campo para ponteiros (por exemplo: método de Brent e o método da árvore binária).

As técnicas que usam o campo de ponteiro têm uma melhor performance, porém requer mais espaço para armazenamento para o campo dos ponteiros. Os métodos que não usam um campo de ponteiro requerem menos espaço, porém possuem performance pior. Chegamos então a uma conclusão, não existe um melhor método para todas as circunstancias.

Veremos agora a uma terceira classe de métodos que está no meio termo entre as classes demonstradas acima. A peculiaridade desta classe está no sentido de que ao invés de armazenar um endereço de memória no campo de ponteiro do nó, armazenamos um campo de pseudoponteiro para localizar o próximo endereço de busca.

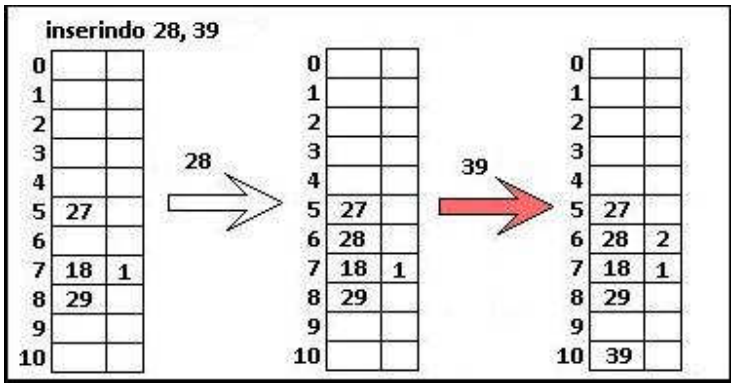
Vamos apresentar um exemplo para facilitar o entendimento.
$h_1(K) = K \bmod (11)$ e $h_2(K) = (K/11) \bmod (11)$



Pelo que se ver na figura as chaves 27 e 18 foram inseridas sem nenhum problema, porém quando se tentou inserir o 29, ela colidiu com o 18 que já estava na célula 7. Nós primeiro então devemos checar se o registro que já está armazenado, está de fato na sua célula, para saber desta informação basta calcular o hash de 18 novamente e averiguar se ele está na sua célula.

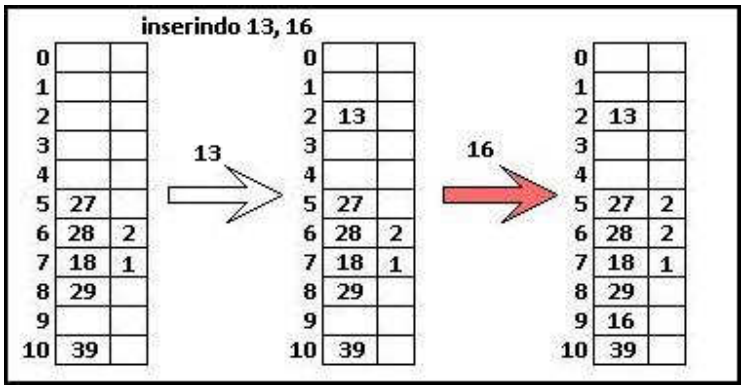
Efetuando este teste, averiguamos que o 18 está na sua casa, então agora devemos determinar se ele possui um sucessor imediato. Um procedimento para verificar se o 18 possui um sucessor imediato, é olhar o seu pseudoponteiro, se ele estiver NULL ou vazio significa que o 18 não possui nenhum sucessor imediato. Então aplicamos a segunda função hash ao elemento que está na célula e encontramos então um lugar para armazenar o 29.

Aplicando a segunda função de hash ao 18, ele nos retorna um incremento de 1, então aplicamos este incremento uma vez e verificamos se atingimos um local vazio, se sim armazenamos o 29 nesta posição e colocamos 1 no pseudoponteiro do 18, caso contrário continuamos a aplicar o incremento do 18, até encontrarmos uma célula da tabela hash e colocamos no campo do pseudoponteiro quantas vezes o incremente foi utilizado para se encontrar uma célula vazia.



O 39 colidiu com o 28 na posição 6, como o 28 está na sua célula corretamente e não possui um sucessor, aplicaremos a segunda função hash ao 28 e utilizaremos o incremento para encontrar uma nova casa para o 39.

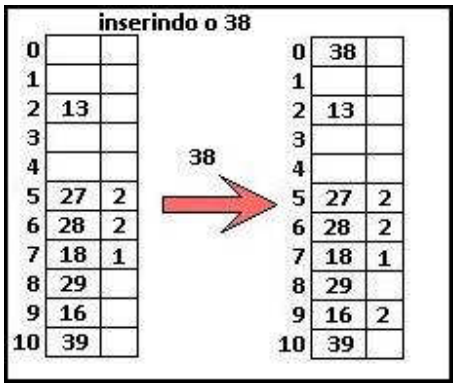
A segunda função hash no 28 retornou 2, como tivemos que executar este incremento duas vezes até encontrar um lugar para o 39, pusemos o 39 no lugar vago que encontramos e pusemos 2 no pseudoponteiro do 28 refletindo assim que utilizamos duas vezes seu incremento.



O 13 é inserido sem problema algum, porém no momento de inserção do 16, ocorre uma colisão.

O 16 colide com o 27 que se encontra na posição 5, verificamos que o 27 está na sua casa, pois aplicamos ao 27 a primeira função hash nele, e ela retornou 5 como resposta.

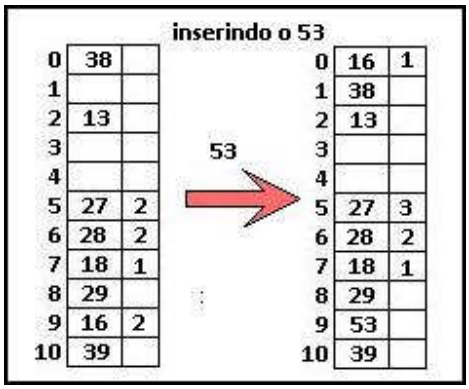
Uma vez que o 27 está na sua posição e que ele não possui nenhum outro sucessor, aplicamos então a segunda função hash nele, encontramos então um incremento de 2, tem-se então que utilizar este incremento pelo menos duas vezes para se encontrar uma célula vaga para o 16 na tabela hash.



Inserindo o 38, nos deparamos com uma situação um pouco diferente da qual tínhamos visto anteriormente. O 38 colide com o 27 que como já sabemos, está na casa que é dele por direito, porém agora o seu pseudoponteiro não está mais vazio, ele nos informa que o próximo elemento com o mesmo valor de hash estará aplicando a segunda função hash no 27 e utilizando seu incremento duas vezes na tabela hash.

A segunda função de hash retorna o valor 2 com relação ao 27, utilizando ela duas vezes, chegamos ao 16, percebe-se que o pseudoponteiro do 16 está vazio, então vamos aplicar a segunda função de hash ao 16 e verificar quantas vezes teremos que usar este incremento para encontrar um local para o 38.

A segunda função retorna 1 para o 16, aplicando então duas vezes, chegamos a célula 0 na tabela hash a qual é preenchida com o elemento 38 e atualizamos o pseudoponteiro do 16 informando quantas vezes foi necessário o uso do seu incremento para encontrar uma célula vaga.



Aplicando o hash ao valor 53, ele nos retorna 9, porém a célula 9 já está ocupada pelo 16, que diferente das situações anteriores não está na sua casa, mais uma vez para descobrirmos isto, basta aplicar a primeira função hash ao 16 e notar que ela retorna o valor 5, porém o 16 foi inserido na casa 9 pelo fato da casa 5 já está ocupada. Esta situação agora nos trás uma complicação para se inserir o 53.

Devemos então fazer o seguinte, armazenar fora da tabela hash o valor de onde o 53 deveria na realidade ficar e todos os seus sucessores. Pelo exemplo temos que armazenar o 16, depois armazenar o elemento 38.

Para saber todos os sucessores de um número basta aplicar a segunda função hash e ver quantas vezes precisamos aplicar o incremento informado no campo do pseudoponteiro.

16, segunda função hash retorna 1 e devemos aplicar ela duas vezes, então chegamos a célula 0 onde está armazenado o 38. Pelo exemplo nosso processo acaba aqui pois o pseudoponteiro da posição 0 é vazio.

Caso o pseudoponteiro da célula não fosse vazia, aplicaríamos então a segunda função hash ao 38 e veríamos pelo pseudoponteiro quantas vezes deveríamos aplicar na tabela para se pegar o seu sucessor.

Uma vez que já temos esta lista em mãos, removemos estes elementos da tabela hash e marcamos NULL ou vazio para o pseudoponteiro que nos levaria pela tabela hash até a posição 9. Em outras palavras marcamos ponteiro NULL ao predecessor imediato ao 16, que no caso da nossa tabela hash do exemplo é o pseudo ponteiro da célula 5. Para confirmar isto, basta calcular a segunda função hash do 27 e aplicar o seu resultado duas vezes na tabela, que chegaremos então até a célula onde estava o 16, no caso a célula 9.

Neste momento temos então uma tabela hash igual a apresentada acima porém sem os números 16 e 38, agora vamos fazer o processo de inserção do 53, pelo fato do 16 ter sido removido a inserção do 53 se torna trivial, basta apenas colocação na célula 9 da tabela hash.

Feito este procedimento, devemos por novamente na tabela hash os elementos que armazenamos fora da tabela que no caso é o 16 e o 38, este processo de inserção segue os mesmos passos para inserção de qualquer outro elemento.

Ao tentar colocar o 16 de volta na tabela, ocorre um choque então com o 27 que está na posição 5, seu pseudoponteiro está vazio, lembra que apagamos. Então calculamos a segunda função hash do 27 e verificamos quantas vezes devemos aplicar seu resultado até encontrar uma célula vazia para se por o 16, o mesmo procedimento acontece com o 38.

A vantagem da cadeia computada é que obtemos melhor performance do que nos métodos de colisão sem ponteiro. Uma busca sem sucesso por este método acaba quando chegamos a uma célula onde seu pseudoponteiro é NULL ou vazio.

Assim como em outras técnicas de resolução de colisão a busca sem sucesso em uma cadeia computada demora mais que uma busca com sucesso.

A deleção de um registro da tabela hash, implica em ter que inserir novamente todos os sucessores daquele registro na tabela hash.

A cadeia computada, assim como vimos em outras técnicas, requer mais tempo para inserir e deletar registros do que somente efetuar uma simples busca.