Estruturas de Dados 2

03 – Tabela *Hash*

Antonio Angelo de Souza Tartaglia
angelot@ifsp.edu.br

Estrutura de Dados 2

Tabela *Hash*

- Uma grande variedade de método de busca funciona segundo o mesmo princípio:

> Procurar a informação desejada com base na comparação de suas chaves, isto é, com base em algum valor que a compõe.

- Um dos problemas disso é que, para obtermos algoritmos eficientes, os elementos devem ser armazenados de forma ordenada.

- Desconsiderando o custo da ordenação, que no melhor caso é $O(n \log n)$, temos que os algoritmos mais eficientes de busca possuem complexidade $O(\log n)$.

> Uma operação de busca ideal, seria aquela que permitisse o acesso direto ao elemento procurado, sem nenhuma etapa de comparação de chaves. Neste caso, teríamos um custo de $O(1)$.

Estrutura de Dados 2

Tabela *Hash*

- *Arrays*, ou vetores, são estruturas que utilizam índices para armazenar informações.
- Por meio do índice, podemos **acessar** determinada posição do *array* com o custo $O(1)$. Infelizmente, os *arrays* não possuem nenhum mecanismo que permita calcular a posição em que uma informação está armazenada, de modo que a operação de busca em um *array*, não é $O(1)$.

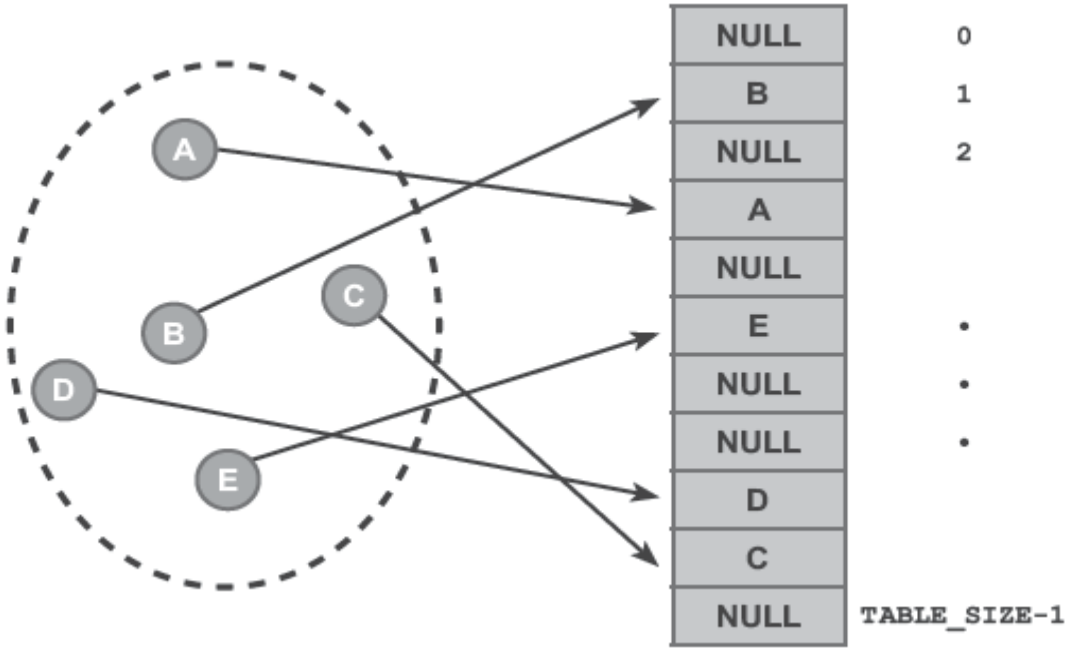
O acesso direto a um elemento de um *array* com base em parte de sua informação (chave), é possível através do uso do tabelas *Hash*.

- Também conhecidas como tabelas de indexação, ou de espalhamento, a tabela *Hash* é uma generalização da ideia de *array.*

Estrutura de Dados 2

Tabela *Hash*

- Sua ideia central é utilizar uma função, chamada **função de *hashing***, para espalhar os elementos que queremos armazenar na tabela. Esse espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do *array* que define a tabela:



Estrutura de Dados 2

Tabela *Hash*

- Mas, por que espalhar os elementos de forma não ordenada pode melhorar a busca?
- A tabela *Hash* é uma estrutura de dados especial que permite a associação de valores a chaves.
- A **chave** é uma parte da informação que compõe o elemento a ser inserido ou buscado na tabela, sendo o valor retornado pela função, a posição (índice) em que o elemento se encontra no *array* que define a tabela.
- Assim, a partir de uma chave podemos acessar de forma rápida uma determinada posição do array. Na média, essa operação tem custo $O(1)$.

Tabela *Hash*

- Várias são as vantagens em se utilizar uma tabela *Hash*:
  - Alta eficiência na operação de busca: o caso médio é $O(1)$, enquanto o da busca linear é $O(n)$, e da busca binária é $O(\log_2 n)$;
  - O tempo de busca é praticamente independente do número de chaves armazenadas na tabela;
  - Implementação simples.
- Infelizmente, esse tipo de implementação também tem suas desvantagens:
  - Alto custo para recuperar os elementos da tabela ordenados pela chave. Nesse caso é preciso ordenar a tabela;
  - O pior caso é $O(n)$, em que n é o tamanho da tabela: alto número de colisões.

Tabela *Hash*

- Uma colisão é a ocorrência de duas ou mais chaves na tabela *Hash* com o mesmo valor de posição, ou seja, uma colisão ocorre quando duas (ou mais) chaves diferentes tentam ocupar a mesma posição na tabela *Hash*.

- A colisão de chaves não é algo exatamente ruim, é apenas algo indesejável, pois diminui o desempenho do sistema.

Tabela *Hash*

- Aplicações

- Existem várias aplicações que fazem uso de tabelas *Hash.* Elas podem ser utilizadas para:

  - Busca de elementos em base de dados: estruturas de dados em memória, bancos de dados e mecanismos de busca na internet;
  - Verificação de integridade de dados e autenticação de mensagens: os dados recebidos são enviados juntamente com o resultado da função de *hashing*. Quem receber os dados recalcula a função de *hashing* usando os dados recebidos e compara o resultado obtido com o que foi recebido. Se os resultados forem diferentes, houve erro de transmissão;
  - Armazenamento de senhas com segurança: a senha não é armazenada no servidor, mas sim o resultado da função de *hashing*;
  - Implementação da tabela de símbolos dos compiladores;
  - Criptografia: MD5 e família SHA (*Secure Hash Algoritm*).

Tabela *Hash*

- Criando o Tipo Abstrato de Dado `hashTable`

- Antes de começar a implementar a nossa tabela *Hash*, é preciso definir o tipo de dado que será armazenado nela. Por utilizar a modularização, será necessário definir o **tipo opaco** que representa nossa tabela. Esse tipo será um ponteiro para a estrutura que define a tabela.

- Além disso, também será necessário definir o conjunto de funções que será visível para o programador que utilizar a biblioteca que será criada, completando assim o encapsulamento do tipo `hashTable`.

Tabela *Hash*

- No arquivo `hashTable.h`, será declarado tudo aquilo que será visível para o programador. Neste arquivo será definido:
  - O tipo de dado que será armazenado na tabela: o tipo `struct aluno`;
  - Para fins de padronização, será criado um novo nome para o tipo `struct hash`. Esse novo nome para o tipo `struct hash` será `Hash`, e deverá ser utilizado sempre que se utilizar uma tabela *Hash*.
  - As funções disponíveis ao programador, para se trabalhar com essa tabela *Hash*, e que serão implementadas no arquivo `hashTable.c`.

- Armazenaremos uma estrutura representando um aluno dentro da tabela. Esse aluno é representado pelo seu número de matrícula, nome e três notas.

Tabela *Hash*

- No arquivo `hashTable.c`, será declarado tudo aquilo que deve ficar oculto do usuário da utilizador da biblioteca, e serão implementadas as funções definidas em protótipo no arquivo `hashTable.h`, que então conterá:
  - As chamadas às bibliotecas necessárias à implementação da tabela *Hash*;
  - A definição do tipo de dado que será a tabela *Hash*, o tipo `struct hash`;
  - As implementações das funções cujos protótipos foram definidos no arquivo `hashTable.h`.
- Observe que o tipo struct hash é uma estrutura contendo três campos:
  - Um inteiro chamado `TABLE_SIZE` que indica o tamanho da tabela *Hash*;
  - Um inteiro chamado `qtd`, que indica a quantidade de elementos armazenados, e;
  - Um ponteiro para ponteiro chamado `itens`.

A ideia aqui é alocar um *array* de ponteiros de tamanho TABLE_SIZE neste campo, para armazenar os elementos inseridos na tabela

Tabela *Hash*

Por questões de desempenho, esta implementação da tabela *Hash*, armazenará apenas o endereço para a estrutura que contém os dados do aluno, e não os dados em si.

- O objetivo deste tipo de abordagem é evitar o gasto excessivo de memória. Em uma tabela *Hash*, os elementos ficam dispersos, ou seja, várias posições do *array* podem não possuir nenhum dado.

- Se o *array* armazenasse a `struct aluno`, teríamos uma grande quantidade de memória desperdiçada. Para evitar isso, a utilização de um *array* de ponteiros (que ocupa muito menos memória do que uma `struct`) e, à medida que os elementos são inseridos na tabela, é então realizada a alocação daquele único elemento.

Tabela *Hash*

- Por estar definido dentro do arquivo `hashTable.c`, os campos da struct hash nã são visíveis pelo usuário da biblioteca no arquivo `main()`, apenas o seu outro nome definido no arquivo `hashTable.h`, que então pode apenas declarar um ponteiro para ele, da seguinte forma:

```
Hash *ha;
```

```c
//Arquivo hashTable.h
struct aluno{
    int matricula;
    char nome[31];
    float n1, n2, n3;
};

typedef struct hash Hash;

Hash *criaHash(int tamanho);
void liberaHash(Hash *ha);
int valorString(char *str);
int insereHash_semColisao(Hash *ha,
                          struct aluno al);
int buscaHash_semColisao(Hash *ha, int mat,
                         struct aluno *al);
int insereHash_enderecoAberto(Hash *ha,
                              struct aluno al);
int buscaHash_enderecoAberto(Hash *ha, int mat,
                             struct aluno *al);
```

Tabela *Hash*

- Arquivo `hashTable.c`

```c
//Arquivo hashTable.c - tipo opaco e funções encapsuladas
#include <stdio.h>
#include <stdlib.h>
#include "hashTable.h"  //inclui os protótipos

//definição odo tipo Hash - tipo opaco
struct hash{
    int qtd;
    int TABLE_SIZE;
    struct aluno **itens;
};
```

Quantidade de elementos inseridos na tabela.

Tamanho máximo da tabela.

Apontará para o vetor de ponteiros.

Tabela *Hash*

- Criando e destruindo uma tabela *Hash*
  - O primeiro passo é criar uma tabela vazia. Esta tarefa é executada pela função:

```
Hash *criaHash(int TABLE_SIZE)
```

  - Basicamente o que essa função faz, é a alocação de uma área de memória para a tabela *Hash*. Essa área corresponde a memória necessária para armazenar a estrutura que define a tabela, `struct hash`.
  - Em seguida, essa função inicializa o campo TABLE_SIZE com o tamanho de tabela informado na função `main()` pela variável tamanho. Esse valor será também utilizado para alocar o *array* de ponteiros no campo itens, o qual será inicializado com o valor NULL para cada posição, indicando que a posição está vaga na tabela.
  - O campo `qtd` também é inicializado com o valor `0`, indicando que nenhum elemento foi inserido na tabela.

Tabela *Hash*

```c
//Arquivo hashTable.c - Criando a tabela de Hash
Hash *criaHash(int TABLE_SIZE){
    Hash *ha = (Hash*) malloc(sizeof(Hash));
    if(ha != NULL){
        int i;
        ha->TABLE_SIZE = TABLE_SIZE;
        ha->itens = (struct aluno**) malloc(TABLE_SIZE * sizeof(struct aluno*));

        if(ha->itens == NULL){
            free(ha);
            return NULL;
        }

        ha->qtd = 0;
        for(i = 0; i < ha->TABLE_SIZE; i++){
            ha->itens[i] = NULL;
        }
    }
    return ha;
}
```

Alocando o array de ponteiros com o tamanho máximo definido por TABLE_SIZE

Inicializando o array de ponteiros

Estrutura de Dados 2

Tabela *Hash*

- Ao se definir o tamanho de uma tabela *Hash*, o ideal é que utilize um número primo, evitando a utilização de valores que sejam potencia de dois.
- Essa escolha reduz a probabilidade de colisões, mesmo que a função de *hashing* utilizada não seja muito eficaz.
- Utilizar um valor que seja uma potência de dois como tamanho melhora a velocidade, mas pode aumentar os problemas de colisão se for utilizada uma função de *hashing* mais simples

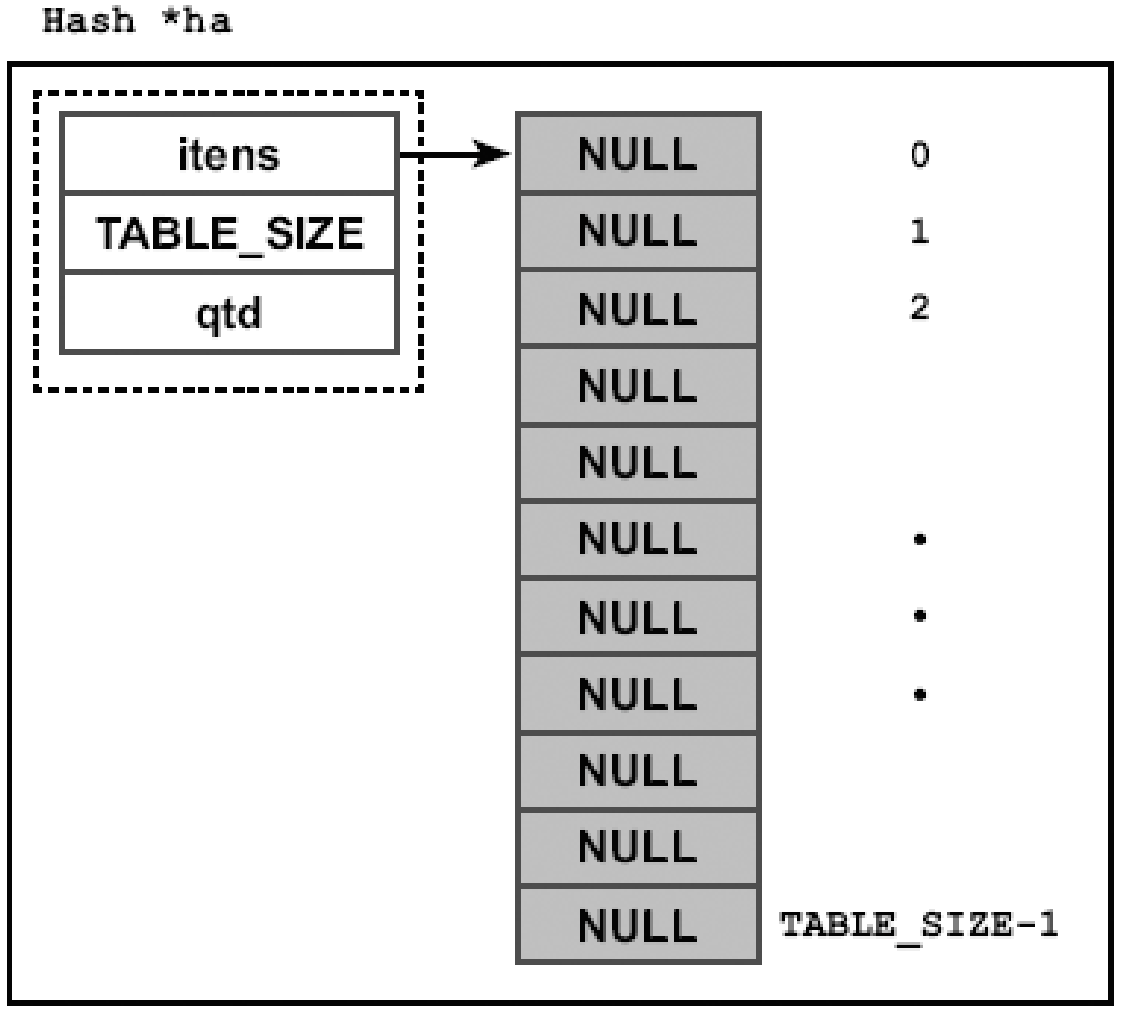


Tabela *Hash*

- A destruição, ou liberação de memória alocada de uma tabela *Hash*, é uma tarefa simples. Basicamente, percorre-se todo o *array* de ponteiros que define a tabela buscando por elementos que tenham sido armazenados, ou seja, elementos do *array* que possuam endereços de memória válidos consequentemente diferentes de NULL. Dessa forma todos os elementos do vetor que apontam para uma região de memória alocada são liberados pela função `free()`.

- Ao final do processo, libera-se a memória alocada para o próprio *array* de ponteiros e para a estrutura (`struct hash`) que representa a tabela.

```c
//Arquivo hashTable.c - Destruindo a tabela de Hash
void liberaHash(Hash *ha){
    if(ha != NULL){
        int i;
        for(i = 0; i < ha->TABLE_SIZE; i++){
            if(ha->itens[i] != NULL){
                free(ha->itens[i]);
            }
        }
        free(ha->itens);
        free(ha);
    }
}
```

Tabela *Hash*

- Calculando a posição da chave: a função de *hashing*.

- Tanto na operação de inserção de elementos, quanto na operação de busca de elementos na tabela *Hash*, é necessário calcular essa posição a partir de uma **chave** escolhida entre os dados manipulados.

- Essa função é extremamente importante para o bom desempenho da tabela. Ela que distribuirá as informações de forma equilibrada pela tabela *Hash*. Ela calcula, a partir do valor do dado, a posição dele dentro da tabela. Para isso, essa função deve satisfazer as seguintes condições:

  - Ser simples e barata de se calcular;
  - Garantir que valores diferentes produzam posições diferentes;
  - Gerar uma distribuição equilibrada dos dados na tabela, ou seja, cada posição da tabela tem a mesma chance de receber uma chave (máximo espalhamento).

Tabela *Hash*

- A implementação da função de *hashing* depende do conhecimento prévio da natureza e domínio da chave a ser utilizada.

- Por exemplo, utilizar apenas três dígitos do número do telefone de uma pessoa para armazená-lo na tabela. Neste caso seria melhor utilizar os três últimos dígitos, ao contrário dos três primeiros, pois os primeiros costumam se repetir com maior frequência e iriam gerar posições iguais na tabela, causando colisões.
- Dessa forma o ideal é utilizar um cálculo diferente de *hash* para cada tipo de chave.

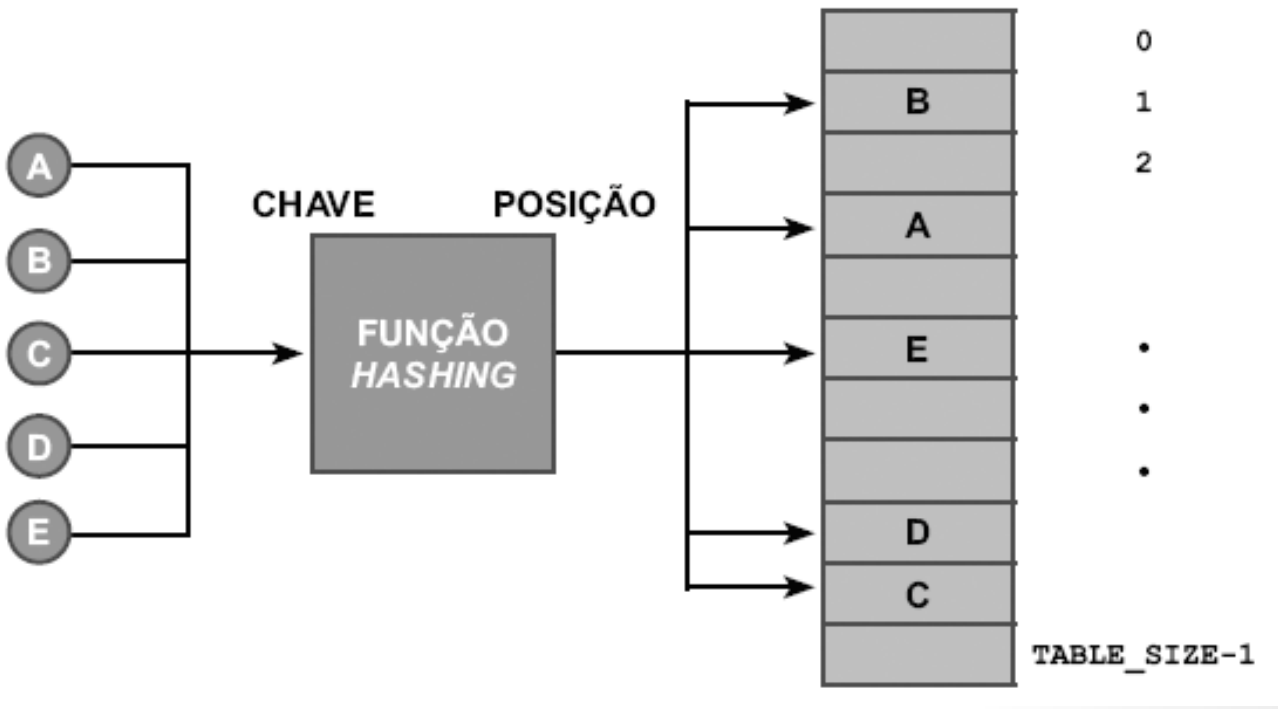
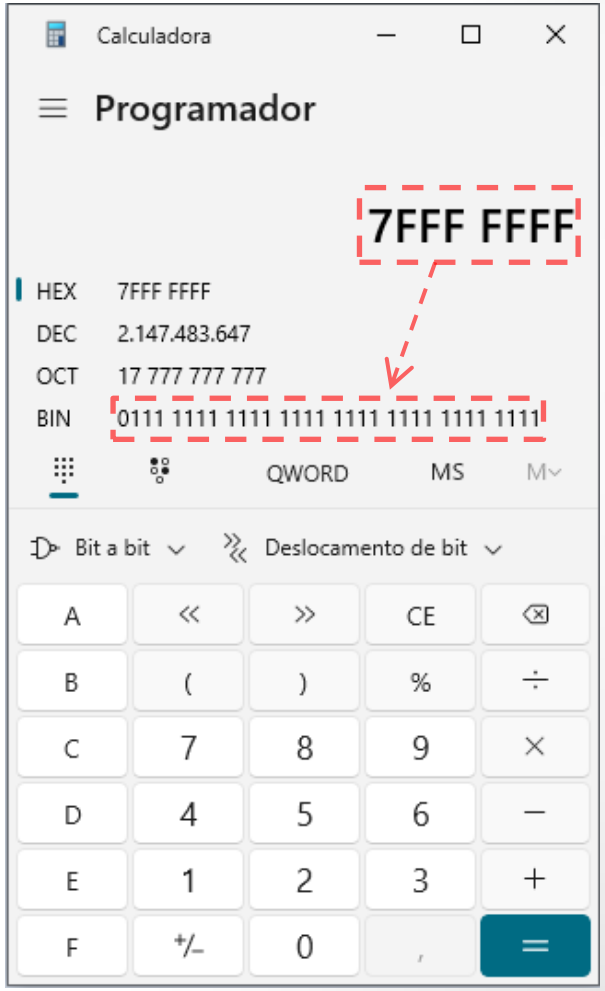


Tabela *Hash*

- Método da divisão
  - A função de *hashing* que utiliza o método da divisão (ou congruência linear) para espalhar os elementos, é bem simples e direta. Basicamente, consiste em calcular o resto da divisão do valor inteiro que representa o elemento, pelo tamanho da tabela, representado por `TABLE_SIZE`.

```c
//Arquivo hashTable.c - Chave: método da divisão
int chaveDivisao(int chave, int TABLE_SIZE){
    return (chave & 0x7FFFFFFF) % TABLE_SIZE;
}
```



- A posição é calculada utilizando uma simples operação de módulos. Antes da operação de módulo, é realizada uma operação de `E bit-a-bit (&)` com o valor `0x7FFFFFFF`. Isso é feito apenas para eliminar o bit de sinal do valor da chave, o que evita o risco de ocorrer um *overflow* e obter-se um número negativo.

Tabela *Hash*

- Apesar de simples, o método da divisão apresenta alguns problemas. Como ele trabalha com o resto da divisão, valores diferentes podem resultar na mesma posição.
- Por exemplo, o resto da divisão de 11 por 10 e de 21 por 10, são o mesmo valor de posição: 1.
- Uma maneira de reduzir esse tipo de problema é utilizar como tamanho da tabela, TABLE_SIZE, um número primo.

Tabela *Hash*

- Método da multiplicação

- A função *Hash* que utiliza o método da multiplicação (ou congruência linear multiplicativa) para espalhar os elementos é outra bastante simples e direta. Basicamente, usa uma constante fracionária denominada A. Esta constante deverá estar compreendida entre zero e um, ou seja, `0 < A < 1`, que então é utilizada para multiplicar o valor da chave que representa o elemento.
- Em seguida, a parte fracionária resultante é multiplicada pelo tamanho da tabela para calcular a posição do elemento. Para entender esse processo, considere calcular a posição da chave 123456, usando a constante fracionária `A = 0,618` e que o tamanho da tabela seja `1024`:

```
posição = ParteInteira(TABLE_SIZE * ParteFracionária(chave * A))
posição = ParteInteira(1024 * ParteFracionária(123456 * 0,618))
posição = ParteInteira(1024 * ParteFracionária(762950,808))
posição = ParteInteira(1024 * 0,808)
posição = ParteInteira(827,392)
posição = 827
```

Estrutura de Dados 2

Tabela *Hash*

- Método da multiplicação

```c
//Arquivo hashTable.c - Chave: método da multiplicação
int chaveMultiplicacao(int chave, int TABLE_SIZE){
    float A = 0.6180339887; //constante: 0 < A < 1
    float val = chave * A;
    val = val - (int) val;
    return (int)(TABLE_SIZE * val);
}
```

Conversão para inteiro

Tabela *Hash*

- Método da dobra
- Diferente dos outros métodos para espalhar os elementos na tabela, o método da dobra utiliza um esquema de dobrar e somar os dígitos do valor para calcular sua posição. Considera apenas o valor inteiro que representa o elemento como uma sequência de dígitos escritos em um pedaço de papel.
- Enquanto esse valor for maior que o tamanho da tabela, o papel é dobrado e os dígitos sobrepostos são então somados, desconsiderando as dezenas resultantes dessa soma.
- O processo deve ser repetido enquanto os dígitos formarem um número maior que o tamanho da tabela.

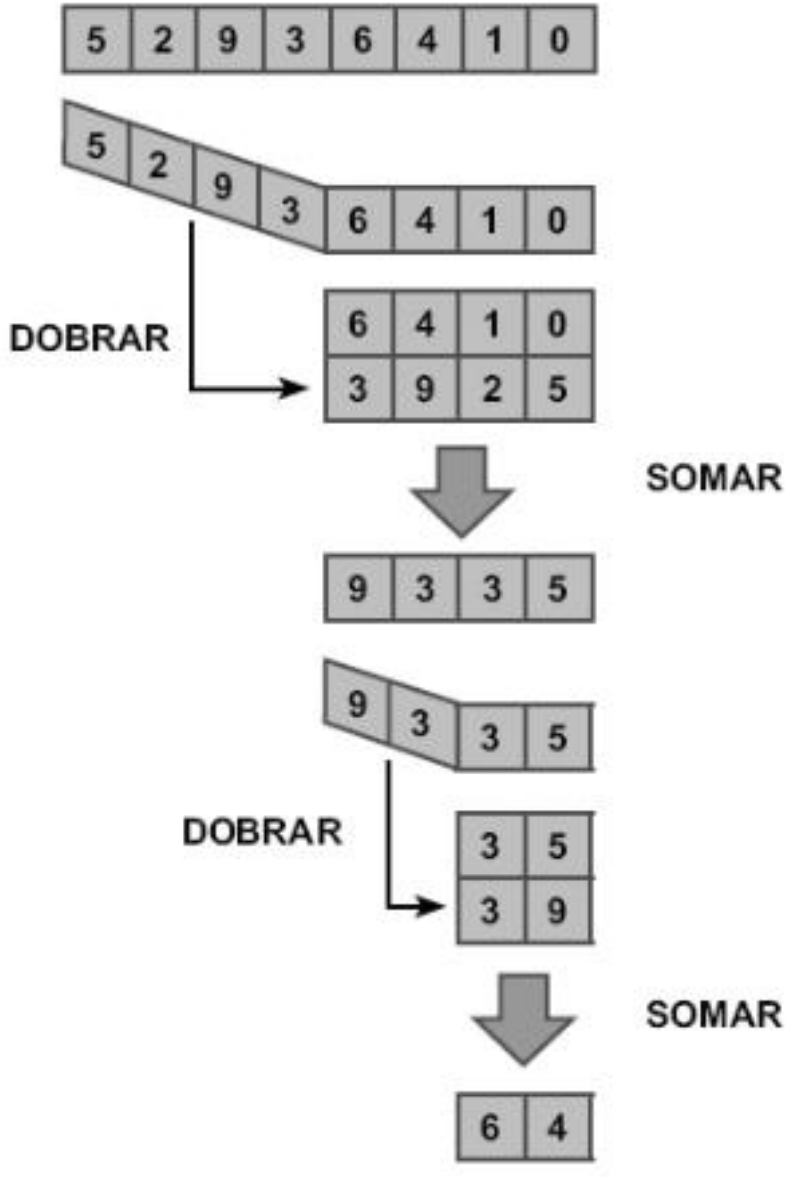


Tabela *Hash*

- Método da dobra

```c
//Arquivo hashTable.c - Chave: método da dobra
int chaveDobra(int chave, int TABLE_SIZE){
    int num_bits = 10;
    int parte1 = chave >> num_bits;
    int parte2 = chave & (TABLE_SIZE - 1);
    return (parte1 ^ parte2);
}
```

>> - desloca a direita num_bits

& - operação AND bit a bit

- Esse método também pode ser usado com valores binários. Nesse caso, ao contrário da soma, utiliza-se a operação de “ou exclusivo”. Não se usa as operações de “e” e “ou” binário, pois produzem resultados menores e maiores que os operandos.

Tabela *Hash*

- Método da dobra

- No caso de valores binários, a dobra é realizada de k em k *bits*, o que resulta em um valor de posição entre zero e $2^k + 1$. Por exemplo, queremos calcular a posição para o valor 71 (`00010 00111` em binário), usando k = 5:

```
posição = 00010 "ou exclusivo" 00111
posição = 00101
posição = 5
```

```
A -> 00010
B -> 00111
S -> 00101
```

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Tabela *Hash*

- Tratando *String* como chave

- Sempre que desejamos inserir ou buscar um elemento na tabela *Hash*, utilizamos parte desse elemento como chave para calcular a sua posição na tabela. No caso da `struct aluno` definida anteriormente, podemos utilizar o número da matricula com a chave para calcular a posição.

- Mas e se meus dados fossem apenas compostos por *Strings*, e não existissem valores numéricos disponíveis?

- No caso de elementos constituídos exclusivamente por *Strings*, é possível calcular um valor numérico a partir dessa *String*. Esse valor pode ser facilmente calculado somando os valores `ASCII` dos caracteres que compõe a *String*. O resultado dessa função pode então ser utilizado como parâmetro para uma função de *hashing*

Estrutura de Dados 2

Tabela *Hash*

- Tratando *String* como chave

```c
//Arquivo hashTable.c - Calculando o valor de uma String
int valorString(char *str){
    int i, valor = 7;
    int tam =strlen(str);
    for(i = 0; i < tam; i++){
        valor = 31 * valor + (int) str[i];
    }
    return valor;
}
```

- Porque não devemos simplesmente somar os valores ASCII dos caracteres da *String*?

Tabela *Hash*

- Tratando *String* como chave

- No caso de chaves utilizadas a partir de Strings, não é aconselhável apenas somar os valores ASCII dos caracteres porque palavras com letras trocadas irão produzir o mesmo valor, e consequentemente uma colisão:

```
cama: 99 + 97 + 109 + 97 = 402
maca: 109 + 97 + 99 + 97 = 402
```

- Assim, para *Strings* é importante considerar a posição do caractere, como definido na função `valorString()`. Nela, cada novo caractere é somado ao produto da soma dos caracteres anteriores pelo número 31. Desse modo, a posição dos caracteres terá influência no valor produzido.
- A escolha do número 31 se deve ao fato de ser um número primo, o que reduz a possibilidade de produzir valores iguais.

Tabela *Hash*

- Inserção e busca sem tratamento de colisões
  - Tanto a inserção quanto a busca são tarefas relativamente simples de se realizar em uma tabela *Hash*. Para as duas funcionalidades temos que realizar o cálculo da posição dos dados no array, a partir dos próprios dados (sua chave) a serem inseridos ou buscados.
  - Começando pela inserção, para o cálculo da posição do elemento na tabela, considere que a matrícula é a chave. Observe que o mesmo pode ser realizado com o nome do aluno (trecho comentado no código), sendo apenas necessário converter a *String* para um valor inteiro.
  - Em seguida é calculada a posição dessa chave utilizando o método da divisão. Neste ponto, a posição também poderia ser calculada com qualquer das outras funções de *hashing*.

Tabela *Hash*

```c
//Arquivo hashTable.c - inserção sem tratamento de colisão
int insereHash_semColisao(Hash *ha, struct aluno al){
    if(ha == NULL || ha->qtd == ha->TABLE_SIZE){
        return 0;
    }
    int chave = al.matricula;
    //int chave = valorString(al.nome);

    int pos = chaveDivisao(chave, ha->TABLE_SIZE);
    //int pos = chaveMultiplicacao(chave, ha->TABLE_SIZE);
    //int pos = chaveDobra(chave, ha->TABLE_SIZE);
    struct aluno *novo;
    novo = (struct aluno*) malloc(sizeof(struct aluno));
    if(novo == NULL){
        return 0;
    }
    *novo = al;
    ha->itens[pos] = novo; //armazena o endereço do novo elemento
    ha->qtd++; //atualiza quantidade de elementos
    return 1;
}
```

Calcula a posição do novo elemento no vetor de ponteiros, observe que aqui pode-se utilizar qualquer das 3 funções de *hashing*.

Aloca memória para novo elemento, e se alocação ok, copia novo elemento para memória alocada.

Tabela *Hash*

- Já a operação de busca, é uma tarefa quase imediata. Assim como na operação de inserção, é utilizado o número de matricula como chave para calcular a posição em que se encontra o elemento a se buscar.

> Observe que a posição do elemento procurado pode ser calculada, assim como na inserção, com qualquer função de *hashing*. Apenas é necessário que seja utilizada na operação de busca, a mesma função de *hashing* utilizada pela inserção.

- Caso o valor armazenado na posição calculada pela função de *hashing* seja igual a NULL, trata-se de uma posição em que nenhum dado de aluno está armazenado nesta posição. Porém, se a posição contiver um endereço válido, seu conteúdo é copiado para a estrutura apontada pelo ponteiro passado como referência, na chamada da função `buscaHash_semColisao()`.

Tabela *Hash*

```c
//Arquivo hashTable.c - Busca sem tratamento de colisão
int buscaHash_semColisao(Hash *ha, int mat, struct aluno *al){
    if(ha == NULL){
        return 0;
    }
    int pos = chaveDivisao(mat, ha->TABLE_SIZE);
    //int pos = chaveMultiplicacao(chave, ha->TABLE_SIZE);
    //int pos = chaveDobra(chave, ha->TABLE_SIZE);
    if(ha->itens[pos] == NULL){
        return 0;
    }
    *al = *(ha->itens[pos]);
    return 1;
}
```

A busca deve utilizar a mesma função de *hashing* utilizada para inserir o elemento, caso contrário, o cálculo efetuado resultará em uma posição diferente da ocupada pelo elemento.

Uma vez encontrado o elemento, basta copiá-lo para o endereço da estrutura passado para a função

Tabela *Hash*

- *Hashing* universal

- É importante lembrar que uma função de *hashing* está sujeita ao problema de gerar posições iguais para chaves diferentes. Por se tratar de uma função determinística, ela pode ser manipulada de forma indesejada.

- Conhecendo-se a função de *hashing*, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca, para `O(n)`.

O *hashing* universal é uma estratégia que busca minimizar esse problema de colisões. A proposta é escolher aleatoriamente (em tempo de execução) a função de *hashing* que será utilizada a partir de um conjunto de funções de *hashing* previamente definidos.

Tabela *Hash*

- *Hashing* universal
- Existem várias maneiras diferentes de se construir um conjunto (ou família) de funções de *hashing*. Uma família de funções pode ser facilmente obtida da seguinte forma:
  - Escolha um número primo p de tal modo que o valor de qualquer chave k a ser inserida na tabela, seja menor que p e maior ou igual a zero, assim, $0 \leq k < p$. Observe que o valor de p será obrigatoriamente maior que o tamanho da tabela, TABLE_SIZE.
  - Escolha aleatoriamente, dois números inteiros, a e b, de modo que a seja maior que zero e menor ou igual a p, $0 < a \leq p$, e b seja maior ou igual a zero e menor ou igual a p, $0 \leq b \leq p$. Dados os valores de p, a e b, definimos a função de *hashing* universal como:

$$H(k)_{(a,\ b)} = ((ak + b)\ \%\ p)\ \%\ \text{TABLE\_SIZE}$$

- Este tipo de função de *hashing* universal, permite que o tamanho da tabela, TABLE_SIZE, não seja necessariamente primo. Além disso, existem $p - 1$ valores diferentes para a, e p valores possíveis para b, dessa forma é possível gerar $p(p - 1)$ funções de *hashing* diferentes.

Tabela *Hash*

- *Hashing* perfeito e imperfeito

- A depender do tamanho da tabela, `TABLE_SIZE`, e dos valores inseridos nela, podemos classificar uma função de *hashing* como imperfeita ou perfeita.

> Uma função de *hashing* e dita imperfeita, se para duas chaves diferentes a saída gerada pela função é a mesma posição na tabela.

- Dessa forma, no *hashing* imperfeito podem ocorrer colisões das chaves armazenadas. A colisão das chaves na tabela não é algo exatamente ruim, é apenas indesejável, pois diminui o desempenho do sistema. De modo geral, muitas tabelas *Hash* fazem uso de alguma outra estruturas de dados para lidar com o problema da colisão

Tabela *Hash*

- *Hashing* perfeito e imperfeito

> Uma função de *hashing* e dita perfeita, se nunca ocorre uma colisão.

- Em outras palavras, o *hashing* perfeito garante que não haverá colisão das chaves dentro da tabela, ou seja, chaves diferentes irão sempre produzir posições diferentes na tabela. Desse modo, no pior caso, as operações de busca e inserção são sempre executadas em tempo constante, $O(1)$.

- Esse tipo de *hashing* é utilizado nos casos em que a colisão não é tolerável. Trata-se de um tipo de aplicação muito específica, por exemplo, o conjunto de palavras reservadas de uma linguagem de programação. Nesse caso, conhecemos previamente o conteúdo a ser armazenando na tabela.

Tabela *Hash*

- Tratamento de colisões

> Uma colisão é uma ocorrência de duas ou mais chaves na tabela *Hash* com o mesmo valor de posição

- Em um mundo ideal, uma função de *hashing* irá sempre fornecer posições diferentes para cada um das chaves inseridas, obtendo assim o ***hashing* perfeito**.

- Infelizmente, independente da função de hashing utilizada, existe a possibilidade da função retornar a mesma posição para duas chaves **diferentes**. A esse fenômeno dá-se o nome de colisão.

- Desse modo, a criação de uma tabela *Hash* consiste em duas coisas: uma função de hashing, e uma abordagem para o tratamento de colisões.

Tabela *Hash*

- Tratamento de colisões

> Uma escolha adequada da função de *hashing* e do tamanho da tabela, podem minimizar as colisões.

- Um dos motivos das colisões ocorrerem é porque temos mais chaves para armazenar do que o tamanho da tabela suporta. Como não há espaço suficiente para todas as chaves na tabela, colisões irão ocorrer.

- Com relação à função de *hashing*, a escolha de uma função que produza um espalhamento uniforme das chaves pode reduzir o número de colisões. Infelizmente, não se pode garantir que as funções de *hashing* possuam um bom potencial de distribuição (espalhamento) porque as colisões também são uniformemente distribuídas. Assim, independente da função que for escolhida, algumas colisões irão ocorrer.

Tabela *Hash*

- Tratamento de colisões

> Colisões são teoricamente inevitáveis, portanto, deverá sempre existir uma abordagem para trata-las.

- Independentemente da qualidade da função de *hashing* utilizada, deverá existir um método para resolver o problemas das eventuais colisões, quando ocorrerem.

- Existem diversas formas de se tratar colisões em tabelas *Hash*. Veremos as mais comuns:

  - **Endereçamento aberto**, e;
  - **Encadeamento separado**.

Tabela *Hash*

- Tratamento de colisões – Endereçamento aberto

A ideia da estratégia de endereçamento aberto é, percorrer (sondar) a tabela *Hash* buscando uma posição ainda não ocupada.

- A ideia do endereçamento aberto, que também é conhecido como *open addressing* ou *rehash*, é que todos os elementos sejam armazenados na própria tabela *Hash*, evitando assim o uso de listas encadeadas.

- Quando uma colisão ocorre, essa estratégia irá procurar posições vagas (as que possuem o valor NULL), dentro do *array* que define a tabela *Hash*, até encontrar um lugar em que aquele elemento poderá ser inserido.

Tabela *Hash*

- Tratamento de colisões – Endereçamento aberto

- Essa abordagem para o tratamento de colisões tem uma série de vantagens:

  - Maior número de posições na tabela para a mesma quantidade de memória usada no encadeamento separado: a memória utilizada para armazenar os ponteiro da lista encadeada no encadeamento separado pode ser aqui utilizada para aumentar o tamanho da tabela, diminuindo o número de colisões;
  - A busca é realizada dentro da própria tabela, o que permite a recuperação mais rápida dos elementos;
  - É voltada para aplicações com restrições de memória;
  - Ao contrário de se acessar ponteiros, calcula-se a sequencia de posições a serem armazenadas.

Tabela *Hash*

- Tratamento de colisões – Endereçamento aberto

> Uma desvantagem importante no tratamento de colisões por endereçamento aberto, é o maior esforço de processamento no cálculo das posições.

- Esse maior esforço de processamento se deve ao fato de que, quando uma colisão ocorre, um novo cálculo deve ser efetuado para uma nova posição na tabela.

- Porém, se essa nova posição também estiver ocupada, outro novo cálculo deve ser efetuado para uma nova posição, e assim por diante. Dessa forma, o cálculo da posição é refeito até que uma posição vaga seja encontrada.

- Em seu pior caso, o custo da inserção se torna O(n), quando todos os elementos inseridos colidem.

Tabela *Hash*

- Tratamento de colisões – Sondagem Linear
- Na estratégia de sondagem linear, também conhecida como **tentativa linear** ou ***rehash linear***, o algoritmo tenta espalhar os elementos de forma sequencial a partir da posição calculada utilizando a função de *hashing*:

```
//Arquivo hashTable.c - Sondagem linear
int sondagemLinear(int pos, int i, int TABLE_SIZE){
    return ((pos + 1) & 0x7FFFFFFF) % TABLE_SIZE;
}
```

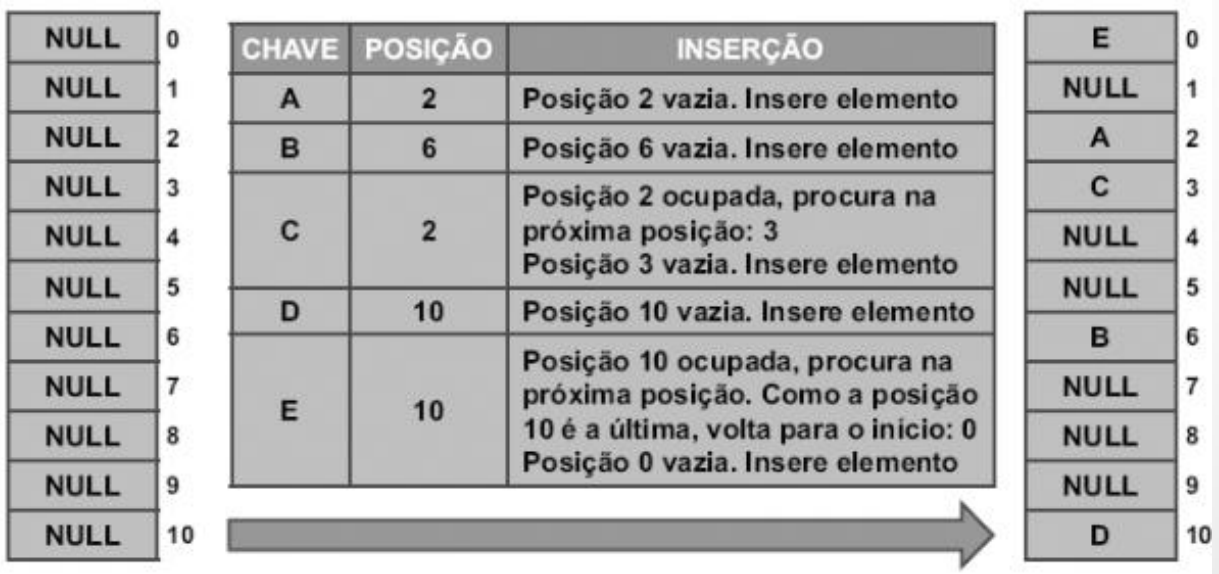
- Nesta função, o primeiro elemento (quando `i = 0`, na função de inserção) é colocado na posição obtida pela função de *hashing* (`pos`), o segundo (em caso de colisão) é colocado na posição `pos + 1` (se for possível), e assim por diante.
- A posição é calculada utilizando uma simples operação de soma e módulo, e que antes da operação de módulo, é realizada uma operação de **`E bit-a-bit (&)`**, com o valor **`0x7FFFFFFF`** para eliminar o bit de sinal do valor da chave. Isso evita o risco de ocorrer overflow e obtermos um número negativo.

Tabela *Hash*

- Tratamento de colisões – Sondagem Linear

Apesar de simples, essa abordagem de espalhamento apresenta um problema conhecido como agrupamento primário: à medida que a tabela *hash* fica cheia, o tempo para incluir ou buscar aumenta.

- À medida que os elementos são inseridos na tabela *Hash* usando a técnica de sondagem linear, começam a surgir longas sequências de posições ocupadas. A ocorrência desses agrupamentos aumenta o tempo de pesquisa na tabela, diminuindo seu desempenho. Além disso, quanto maior for o agrupamento primário, maior a probabilidade de aumenta-lo ainda mais com a inserção de um novo elemento.



NULL	0
NULL	1
NULL	2
NULL	3
NULL	4
NULL	5
NULL	6
NULL	7
NULL	8
NULL	9
NULL	10

CHAVE	POSIÇÃO	INSERÇÃO
A	2	Posição 2 vazia. Insere elemento
B	6	Posição 6 vazia. Insere elemento
C	2	Posição 2 ocupada, procura na próxima posição: 3 Posição 3 vazia. Insere elemento
D	10	Posição 10 vazia. Insere elemento
E	10	Posição 10 ocupada, procura na próxima posição. Como a posição 10 é a última, volta para o início: 0 Posição 0 vazia. Insere elemento

E	0
NULL	1
A	2
C	3
NULL	4
NULL	5
B	6
NULL	7
NULL	8
NULL	9
D	10

Tabela *Hash*

- Tratamento de colisões – Sondagem Quadrática

- Na estratégia da sondagem quadrática, também conhecida como tentativa quadrática, espalhamento quadrático ou rehash quadrático, o algoritmo tenta espalhar os elementos utilizando uma equação do segundo grau de forma $pos + (C_1 * i) + (C_2 * i^2)$, em que *pos* é a posição obtida pela função de hashing, i é a tentativa atual e *C1* e *C2* são coeficientes da equação:

```
//Arquivo hashTable.c - Sondagem quadrática
int sondagemQuadratica(int pos, int i, int TABLE_SIZE){
    pos = pos + 2 * i + 5 * i * i;
    return (pos & 0x7FFFFFFF) & TABLE_SIZE;
}
```

- Nessa estratégia, o primeiro elemento (**`i = 0`**) é colocado na posição obtida pela função de *hashing* (**pos**), o segundo (caso haja colisão) é colocado na posição $pos + (C_1 * 1) + (C_2 * 1^2)$ (se possível), o terceiro (caso nova colisão) é colocado na posição $pos + (C_1 * 2) + (C_2 * 2^2)$ (se possível), e assim por diante.

Tabela *Hash*

- Tratamento de colisões – Sondagem Quadrática

A sondagem quadrática resolve o problema de **agrupamento primário**. Porém, ela gera outro problema conhecido como **agrupamento secundário**.

- O problema do agrupamento secundário ocorre porque todas as chaves que produzem a mesma posição inicial na tabela *Hash*, também produzem as mesmas posições na sondagem quadrática. Felizmente , a degradação produzida na tabela *Hash* pelos agrupamentos secundários ainda é menor que a produzida pelos agrupamentos secundários:

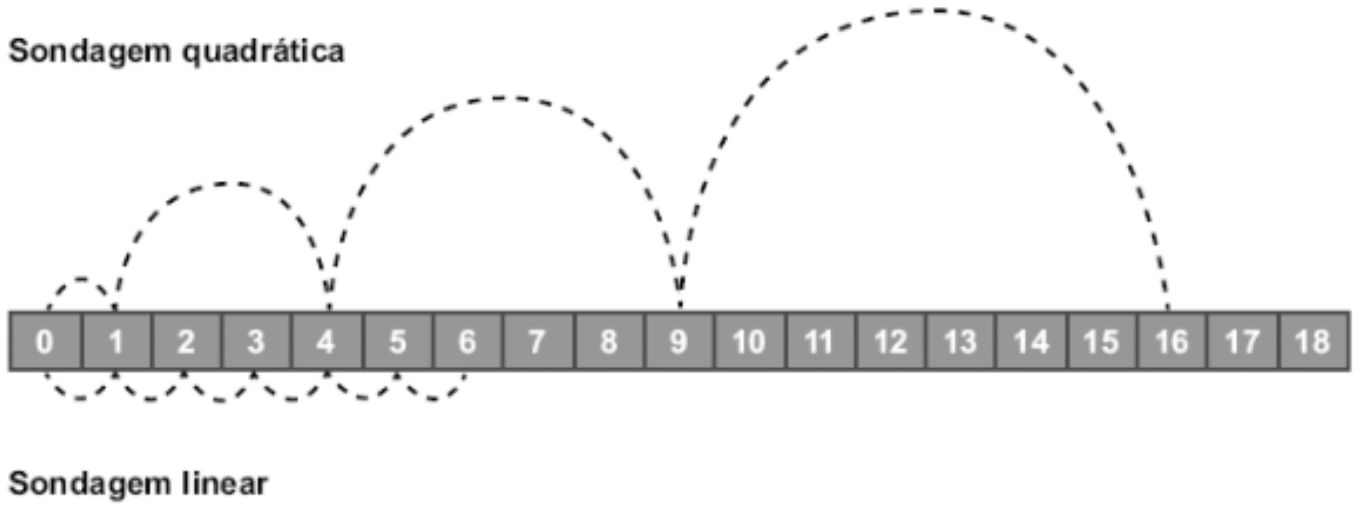


Tabela *Hash*

- Tratamento de colisões – Duplo *Hash*

- Na estratégia de duplo *hash*, também conhecida como espalhamento duplo, o algoritmo tenta espalhar os elementos utilizando duas funções de *hashing*:

  - A primeira função de *hashing*, H1, é utilizada para calcular a posição inicial do elemento;
  - A segunda função de *hashing*, H2, é utilizada para calcular os deslocamentos em relação a posição inicial, caso haja uma colisão.

- Desse modo, a posição de um novo elemento na tabela Hash é obtida a partir de $H1 + i * H2$, em que i é a tentativa atual de inserção do elemento.

Tabela *Hash*

- Tratamento de colisões – Duplo *Hash*

```
//Arquivo hashTable.c - Duplo Hash
int duploHash(int H1, int chave, int i, int TABLE_SIZE){
    int H2 = chaveDivisao(chave, TABLE_SIZE - 1) + 1;
    return ((H1 + i * H2) & 0x7FFFFFFF) % TABLE_SIZE;
}
```

Diminui-se o tamanho da tabela, e soma-se 1 à posição calculada para garantir que não seja retornado 0.

- Nessa estratégia, o primeiro elemento (`i = 0`) é colocado na posição obtida pela primeira função de *hashing* (H1), o segundo (em caso de colisão) é colocado na posição *H1 + i * H2* (se possível), o terceiro (caso nova colisão) é colocado na posição *H1 + i * H2* (se possível), e assim por diante.

- Este tipo de estratégia diminui a ocorrência de agrupamentos, o que faz dele um dos melhores métodos para tratamento de colisões em endereçamento aberto.

Tabela *Hash*

- Tratamento de colisões – Duplo *Hash*

Para o duplo *hash* funcionar corretamente, é necessário que as duas funções de *hashing* sejam diferentes. Além disso, a segunda função de *hashing* não pode resultar em um valor igual a **ZERO**, pois nesse caso, não haveria deslocamento.

- Na implementação desta função, a segunda função de *hashing* é calculada utilizando o método da divisão com um tamanho de tabela um pouco menor. É muito comum utilizar `TABLE_SIZE - 1` ou `TABLE_SIZE - 2`, além disso, é somado 1 ao valor da posição para garantir que a posição retornada não seja ZERO (`0`).

Tabela *Hash*

- Inserção e Busca com tratamento de colisão

Tanto par a inserção quanto para a busca, o que deve ser feito é calcular a posição dos dados no *array* a partir de parte dos dados (chave) a serem inseridos ou buscados. Caso ocorra uma colisão, uma nova posição deve ser calculada de acordo com alguma estratégia de tratamento de colisão

- Na implementação desta função, a segunda função de *hashing* é calculada utilizando o método da divisão com um tamanho de tabela um pouco menor. É muito comum utilizar `TABLE_SIZE - 1` ou `TABLE_SIZE - 2`, além disso, é somado 1 ao valor da posição para garantir que a posição retornada não seja ZERO (`0`).

Estrutura de Dados 2

Prevendo possíveis colisões, considera-se um certo número de tentativas antes da desistência de inserção do elemento. A quantidade de tentativas será igual ao tamanho da tabela

Calcula-se a nova posição do elemento com base em uma estratégia de tratamento de colisão, neste caso a sondagem linear, mas qualquer das outras pode ser utilizada. Na primeira tentativa (i = 0), as funções de tratamento retornam o valor obtido pela função de hashling.

```c
//Arquivo hashTable.c - Inserção com tratamento de colisão
int insereHash_enderecoAberto(Hash *ha, struct aluno al){
    if(ha == NULL || ha->qtd == ha->TABLE_SIZE){
        return 0;
    }
    int chave = al.matricula;
    //int chave = valorString(al.nome);
    int i, pos, newPos;
    pos = chaveDivisao(chave, ha->TABLE_SIZE);
    //pos = chaveMultiplicacao(chave, ha->TABLE_SIZE);
    //pos = chaveDobra(chave, ha->TABLE_SIZE);
    for(i = 0; i < ha->TABLE_SIZE; i++){
        newPos = sondagemLinear(pos, i, ha->TABLE_SIZE);
        //newPos = sondagemQuadratica(pos, i, ha->TABLE_SIZE);
        //newPos = duploHash(pos, chave, i, ha->TABLE_SIZE);
        if(ha->itens[newPos] == NULL){
            struct aluno *novo;
            novo = (struct aluno*) malloc(sizeof(struct aluno));
            if(novo == NULL){
                return 0;
            }
            *novo = al;
            ha->itens[newPos] = novo;
            ha->qtd++;
            return 1;
        }
    }
    return 0;
}
```

Verifica-se a posição obtida está vazia (NULL). Se essa afirmação for falsa, realiza-se nova tentativa para achar uma posição disponível.

Tabela *Hash*

- Inserção e Busca com tratamento de colisão

- Assim como na operação de inserção, a operação de busca exige que as colisões sejam tratadas. No caso da ocorrência de colisão, não se procura uma posição vazia, mas por uma que contenha a porção de dados que foi utilizada no cálculo da chave.

Observe que a posição do elemento procurado poderia ser calculada com qualquer função de *hashing*. É absolutamente necessário apenas que a função de *hashing* usada na busca, seja a **mesma que foi utilizada na operação de inserção**.

Tabela *Hash*

```c
//Arquivo hashTable.c - Busca com tratamento de colisão
int buscaHash_enderecoAberto(Hash * ha, int mat, struct aluno *al){
    if(ha == NULL){
        return 0;
    }
    int i, pos, newPos;
    pos = chaveDivisao(mat, ha->TABLE_SIZE);
    //pos = chaveMultiplicacao(mat, ha->TABLE_SIZE);
    //pos = duploHash(mat, ha->TABLE_SIZE);
    for(i = 0; i < ha->TABLE_SIZE; i++){
        newPos = sondagemLinear(pos, i, ha->TABLE_SIZE);
        //newPos = sondagemQuadratica(pos, i, ha->TABLE_SIZE);
        //newPos = duploHash(pos, i, ha->TABLE_SIZE);
        if(ha->itens[newPos] == NULL){
            return 0;
        }
        if(ha->itens[newPos]->matricula == mat){
            *al = *(ha->itens[newPos]);
            return 1;
        }
    }
    return 0;
}
```

No caso de colisões, será considerado um certo número de tentativas de busca (tamanho da tabela). Se o laço for chegar ao fim (elemento não existe) 0 será retornado (elemento não existe)

Se existir algum elemento armazenado na posição calculada, compara-se a matricula. Se diferentes, executa-se nova será realizada em outra posição.

Calcula-se a nova posição do elemento com base na estratégia de tratamento de colisão. Podem ser usadas outras. Na primeira tentativa (`i = 0`), as funções retornam o valor calculado pela função de *hashing*

Tabela *Hash*

- Encadeamento separado

- O encadeamento separado, ou *separate chaining*, é uma maneira um pouco diferente de se tratar a colisão em uma tabela *Hash*. Em vez de procurar por posições vagas (que contenham NULL) dentro do *array* que define a tabela, esse tipo de tratamento de colisões armazena dentro de cada posição do *array* o início de uma **lista dinâmica encadeada**. E é dentro dessa lista que serão armazenadas as colisões, ou seja, os elementos que possuem chaves iguais, para aquela posição do *array*.

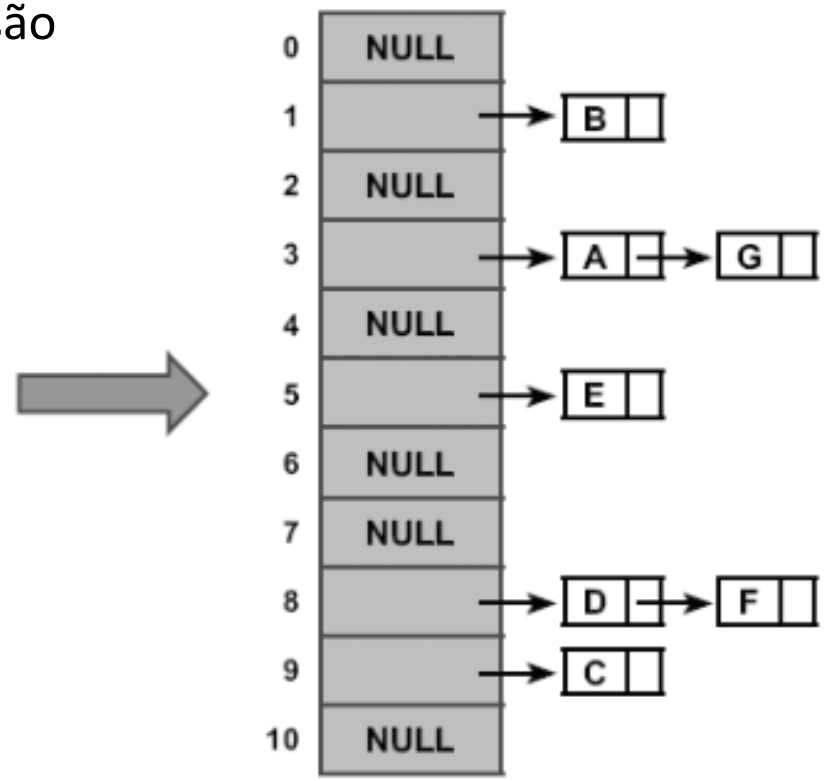
- A **lista dinâmica encadeada** mantida em cada posição da tabela pode ser ordenada ou não. Ao se utilizar uma lista não ordenada, é fácil perceber que essa estratégia de tratamento de colisão tem complexidade $O(1)$, no pior caso: basta acessar a posição da tabela correspondente à chave daquele elemento e inseri-lo no início da lista.

- Já a busca leva um tempo proporcional ao número de elementos dentro da lista armazenada naquela posição da tabela, ou seja, é preciso percorrer a lista procurando por aquele elemento específico.

Tabela *Hash*

- Inserção e Busca com tratamento de colisão

CHAVE	POSIÇÃO
A	3
B	1
C	9
D	8
E	5
F	8
G	3



Uma desvantagem importante desse tipo de tratamento de colisão diz respeito à quantidade de memória consumida: gasta-se mais memória para manter os ponteiros que ligam os diferentes elementos dentro de cada lista.

Atividade

- Implementar o programa *hashTable* no formato de TAD, com os três arquivos: `hashTable.h`, `hashTable.c` e `main.c`. Você poderá incluir mais bibliotecas caso queira (eu acho recomendável), para incluir funções auxiliares.

- Em seu projeto, você desenvolverá todo o arquivo `main.c`, que então controlará a execução de todo o programa da seguinte forma:

  - Um menu apresentará as opções de execução da tabela *Hash* com **inserções sem tratamento de colisões**, ou **inserções com tratamento de colisões**. O programa deverá funcionar somente com uma das opções, e nunca as duas juntas por motivos óbvios, uma vez que as funções de *hashing* são diferentes. Deverá apresentar ainda a opção de encerramento do programa.
  - Uma vez determinado o passo anterior, um segundo menu deverá apresentar as opções de inserção de elementos e busca de elementos, com o questionamento após cada tarefa, se o usuário deseja continuar com as operações. Se não desejar, o programa será encerrado.

- Entregue no Moodle todos os arquivos do projeto zipados.