

Inserção:

Matricula:10043284 Nome: Amila Dias Araujo

Objeto de classe Aluno é criado com as especificações acima.

```
void Insert(const Aluno& element) {  
    table[TrueHash(element)].push_front(element);  
}
```

Para saber em qual bucket na tabela o aluno será armazenado, precisamos saber o que TrueHash(element) retorna:

```
int TrueHash(const Aluno& a) const {  
    return a.Hash() % col_size;  
}
```

A função invoca a função membro Hash() de Aluno e retorna o resto da divisão do número retornado pelo Hash() pelo número de buckets da tabela. Vamos assumir que o número de buckets seja 235 (para tornar o fator de carga igual a 10):

```
int Hash() const {  
    return Aluno::Hash(matricula);  
}
```

Ela delega o trabalho para a função estática...

```
static __int32 Hash(__int32 mat) {  
    auto x = mat;  
    x = ((x >> 16) ^ x) * 0x45d9f3b;  
    x = ((x >> 16) ^ x) * 0x45d9f3b;  
    x = (x >> 16) * 0x45d9f3b;  
    return abs(x);  
}
```

(TODO: apagar isso, tornar menos dependente de recursos externos, tirar as imagens etc.)

Sabendo que a matricula da aluna é 10043284, Podemos utilizar dos seguintes recursos para verificar os cálculos:

<https://bit-calculator.com/bit-shift-calculator>

<https://miniwebtool.com/bitwise-calculator/>

<https://www.mathsisfun.com/calculator-precision.html>

<https://www.rapidtables.com/convert/number/decimal-to-binary.html>

Primeiro, X, que tem como valor atual o número de matricula 10043284, que em binário é "00000000 10011001 00111111 10010100", será bit-shifted 16 vezes para direita. Obtemos assim:

"00000000 10011001"

Note que os 0s a esquerda não são redundantes, estamos simulando como o número é armazenado em X, uma variável que NÃO é unsigned e que usa complemento de 2 para armazenar números com sinais.

Os bits acima são o que resta depois dessa operação, que é 153 em decimal.

Depois, um XOR é feito entre o número original e o resultado anterior, ou seja:
XOR:

```

00000000 10011001 00111111 10010100
                00000000 10011001
-----
00000000 10011001 00111111 00001101

```

que dá 10043149 em decimal.

Multiplicando este número por 0x45d9f3b, que dá 73244475 em decimal, obtemos 735605175851775 como resultado. Ou, em binário:

"00000000 00000010 10011101 00000111 01101101 00110011 10011010 11111111"

Resultado este que agora se torna o valor atual de X, e considerando o overflow, pegamos apenas os 32 bits mais baixos do resultado:

X="01101101 00110011 10011010 11111111"

Obtendo assim o valor 1832098559, que X agora possui.

Foi completado a execução da segunda linha do corpo da função estática Hash.

Agora será necessário repetir o mesmo processo 2 vezes pois faz parte do algoritmo, e irei ser mais breve:

01101101 00110011 10011010 11111111 >> 16 = 01101101 00110011

XOR:

```

01101101 00110011 10011010 11111111
                01101101 00110011
-----
01101101 00110011 11110111 11001100

```

01101101 00110011 11110111 11001100 x 100 01011101 10011111 00111011 (0x45d9f3b em hexadecimal, ou 73244475 em decimal) = 134192837171204100 em decimal

Em binário: "00000001 11011100 10111111 10101100 10100110 11111101 11010000 00000100"

Portanto X = "10100110 11111101 11010000 00000100" (OVERFLOW)

Mais uma vez:

X >> 16 = "10100110 11111101" = -22787

-22787*73244475 = -1669021851825

-1669021851825 = "11111111 11111111 11111110 01111011 01100110 10001011 10011111 01001111"

portanto X= "01100110 10001011 10011111 01001111" = 1720426319

Pegamos o módulo deste valor e retornamos para a versão não estática de Hash, que consequentemente retorna o mesmo valor para TrueHash:

```

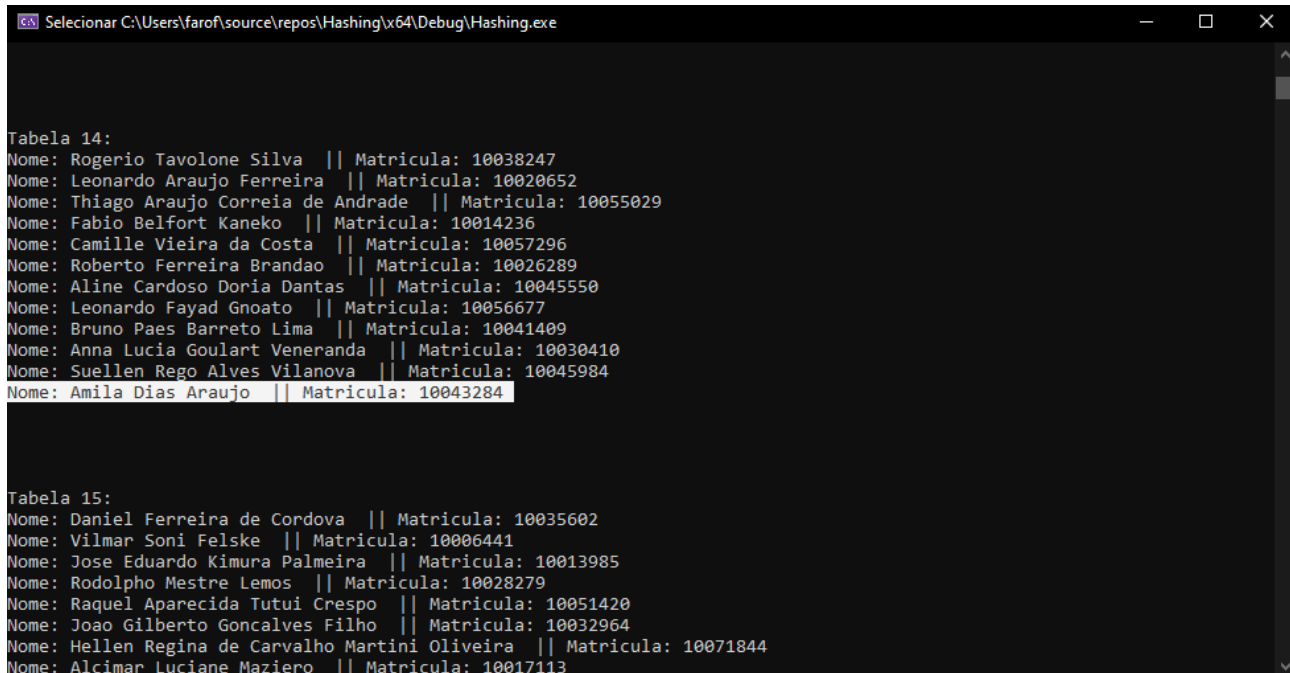
int TrueHash(const Aluno& a) const {
    return a.Hash() % col_size;
}

```

como foi decidido que `col_size` é 235, basta obter o resto de 1720426319 dividido por 235, que dá 14, portanto este é o número que `TrueHash` retorna:

```
void Insert(const Aluno& element) {
    table[TrueHash(element)].push_front(element);
}
```

Finalmente, podemos concluir que a ficha da primeira aluna da lista, Amila Dias Araujo, de Matricula 10043284, será armazenada no bucket de indice 14, e Voilà:



```
Tabela 14:
Nome: Rogerio Tapolone Silva || Matricula: 10038247
Nome: Leonardo Araujo Ferreira || Matricula: 10020652
Nome: Thiago Araujo Correia de Andrade || Matricula: 10055029
Nome: Fabio Belfort Kaneko || Matricula: 10014236
Nome: Camille Vieira da Costa || Matricula: 10057296
Nome: Roberto Ferreira Brandao || Matricula: 10026289
Nome: Aline Cardoso Doria Dantas || Matricula: 10045550
Nome: Leonardo Fayad Gnoato || Matricula: 10056677
Nome: Bruno Paes Barreto Lima || Matricula: 10041409
Nome: Anna Lucia Goulart Veneranda || Matricula: 10030410
Nome: Suellen Rego Alves Vilanova || Matricula: 10045984
Nome: Amila Dias Araujo || Matricula: 10043284

Tabela 15:
Nome: Daniel Ferreira de Cordova || Matricula: 10035602
Nome: Vilmar Soni Felske || Matricula: 10006441
Nome: Jose Eduardo Kimura Palmeira || Matricula: 10013985
Nome: Rodolpho Mestre Lemos || Matricula: 10028279
Nome: Raquel Aparecida Tutui Crespo || Matricula: 10051420
Nome: Joao Gilberto Goncalves Filho || Matricula: 10032964
Nome: Hellen Regina de Carvalho Martini Oliveira || Matricula: 10071844
Nome: Alcimar Luciane Maziero || Matricula: 10017113
```

Aviso: Houve erro na digitação do código, deveria estar escrito "Lista" ao inves de "Tabela" na imagem acima.

Remoção:

```
void Remove(int matricula) {
    table[TrueHash(matricula)].remove_if(
        [=](Aluno& a) {
            return (a.matricula == matricula) ? true : false;
        }
    );
}
```

Vamos assumir que queremos deletar a ficha desta mesma aluna, com o mesmo número de buckets. Já sabemos que, se a ficha dela existe na tabela, deve estar no bucket de indice 14.

Desta forma, o que o código acima está fazendo é invocar a função `remove_if` da tabela 14, que remove todos os elementos desta tabela que satisfazem o seguinte predicado:

```
[=](Aluno& a) {return (a.matricula == matricula) ? true : false;}
```

Isso é uma função lambda que retorna verdadeiro se a matricula do aluno sendo observado na lista é igual a matricula do aluno que queremos ter a ficha deletada, e falso caso não seja.

Vamos simular a função `Remove` sendo chamada desta forma: `Remove(10043284)`.

```
matricula= 10043284
TrueHash(matricula)=14
table[14].remove_if(=[](Aluno& a) {return (a.matricula == 10043284) ? true : false;});
```

Primeiro elemento da lista: Rogerio Tapolone Silva || Matricula: 10038247
Matricula igual a 10043284? Não, seguir em frente na lista.

Segundo elemento da lista: Leonardo Araujo Ferreira || Matricula: 10020652
Matricula igual a 10043284? Não, seguir em frente na lista.

... (Sabemos que cada matricula é única, portanto nenhum outro aluno a não ser a própria Amila terá matricula igual a 10043284, então sabemos que todos os demais darão falso e não serão removidos)

Décimo segundo elemento da lista: Amila Dias Araujo || Matricula: 10043284
Matricula igual a 10043284? Sim, deletar e seguir em frente na lista.

Não há mais nenhum elemento na lista, retornar da função remove_if. Não há mais linhas na função Remove, retornar dela também. Fim.

Impressão:

```
void Print() const {
    for (int i = 0; i < col_size; ++i) {
        std::cout << "Lista " << i << ": \n";
        for (auto a : table[i]) {
            a.Print();
            std::cout << '\n';
        }
        std::cout << "\n\n\n\n";
    }
}
```

Primeira iteração do loop:

```
i=0
std::cout << "Lista " << 0 << ": \n";
for(auto a: table[0]) {
    a.Print();
    std::cout << '\n';
}
std::cout << "\n\n\n\n";
```

Esse código irá escrever "Tabela 0:" seguido de uma nova linha. Depois, para cada elemento da lista de índice 0, irá usar a função membro "Print" deste elemento, seguido de uma nova linha. Finalmente, depois de todos os elementos da lista serem listados, teremos 5 novas linhas.

O código de Print da classe aluno:

```
void Print() const {
    std::wcout << "Nome: " << nome << " || Matricula: " << matricula;
}
```

Portanto, podemos esperar o seguinte resultado no terminal:

Tabela 0:
Nome: Fulano || Matricula: xxxxxxxx
Nome: Beltrano || Matricula: yyyyyyyy

...

Nome: Sicrano || Matricula: zzzzzzzz

Tabela 1:

...

Onde xxxxxxxx é a matricula de Fulano, yyyyyyyy de Beltrano e zzzzzzzz a de Sicrano, e todas as matriculas deles fazem TrueHash retornar 0 para o devido número de listas que foi desejado que a tabela possuia:

```
Selecionar C:\Users\farof\source\repos\Hashing\x64\Debug\Hashing.exe
Lista 0:
Nome: Vitor Carlos Villa Real Lopes || Matricula: 10044983
Nome: Luisa Domingues Ferreira Alves || Matricula: 10049739
Nome: Ricardo Vicelli Cidral da Costa || Matricula: 10072819
Nome: Gilmar Rodrigues de Oliveira || Matricula: 10007256
Nome: Alexandre Avalo Santana || Matricula: 10057890
Nome: Virginia Rosa Queiroz || Matricula: 10084440

Lista 1:
Nome: Giuliana Arruda Pessoa Fonseca || Matricula: 10034569
Nome: Marcio Takano || Matricula: 10038241
Nome: Francisco Aguiar de Farias Junior || Matricula: 10057421
Nome: Lucia Elena Arantes Ferreira Bastos || Matricula: 10040892
Nome: Ligia de Camargo Molina || Matricula: 10039897
Nome: Kleber de Souza Pinto || Matricula: 10072857
Nome: Giovanni Fregonazzi || Matricula: 10043887
Nome: Rachel Medeiros Rizel Santana || Matricula: 10019539

Lista 2:
Nome: Leandro Piva || Matricula: 10062414
```

Análise dos números de elementos dos buckets para cada fator de carga:

Considerando que temos no total 2357 alunos, foi criado um construtor que gera o numero de Listas necessarias para corresponder ao fator de carga desejado.

Com isso, iremos analisar como o número de elementos dos buckets (nesse caso, as Listas) varia a medida em que alteramos o fator de carga de 10, para 2, para 1, para 0.5 e finalmente 0.25.

com fator de carga (F) igual a 10:

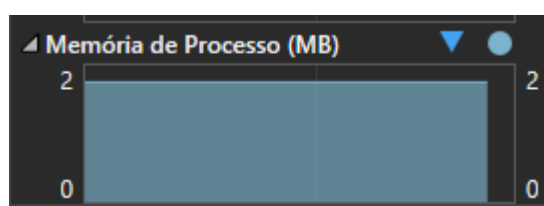
```

C:\Users\farof\source\repos\Hashing\x64\Debug\Hashing.exe
Nome: Aderson Sena dos Santos || Matricula: 10018988
Nome: Livia Laubmeyer Alves de Souza || Matricula: 10059630
Nome: Paulo de Siqueira Campos Junior || Matricula: 10011588
Nome: Andrea da Silveira Monte || Matricula: 10013641
Nome: Paulo Sergio Menendes Siqueira || Matricula: 10000703
Nome: Patricia Freire de Alencar Carvalho || Matricula: 10033667
Nome: Laura Diodatti de Castilho || Matricula: 10023713
Nome: Fabio Inokuti || Matricula: 10023490
Nome: Rafael Nogueira Bezerra Cavalcanti || Matricula: 10019188
Nome: Isabela Santana da Silva || Matricula: 10054840
Nome: Elisangela Belote Mareto || Matricula: 10052157
Nome: Tatiana Karla Almeida Martins || Matricula: 10014336

Buckets com 0 elementos: 0
Buckets com 1 elementos: 0
Buckets com 2 elementos: 0
Buckets com 3 elementos: 4
Buckets com 4 elementos: 7
Buckets com 5 elementos: 7
Buckets com 6 elementos: 16
Buckets com 7 elementos: 17
Buckets com 8 elementos: 26
Buckets com 9 elementos: 29
Buckets com 10 elementos: 28
Buckets com mais de 10 elementos: 101

```

São geradas 235 listas com este fator de carga. Podemos ver que existe uma clara tendência contra Listas vazias ou pouco ocupadas. Isso pode comprometer o tempo de busca/remoção de elementos. Felizmente isso não compromete o tempo de inserção pois ele sempre insere no início da lista. Temos também a vantagem que foram criadas apenas 235 Listas, então consome menos espaço:



Nesse caso, ocupa em torno de 2MBs (A versão Debug do programa, pelo menos).

F=2:

```
C:\Users\farof\source\repos\Hashing\x64\Debug\Hashing.exe

Lista 1176:
Nome: Leny Baumgartner || Matricula: 10004133
Nome: Roberta Wobeto Baraldi || Matricula: 10051544

Lista 1177:
Nome: Roniery Alves da Silva || Matricula: 10008377
Nome: Jeferson Domingues Diniz || Matricula: 10029278
Nome: Ana Tereza de Paula Mesquita || Matricula: 10001725

Buckets com 0 elementos: 253
Buckets com 1 elementos: 261
Buckets com 2 elementos: 253
Buckets com 3 elementos: 196
Buckets com 4 elementos: 120
Buckets com 5 elementos: 63
Buckets com 6 elementos: 20
Buckets com 7 elementos: 10
Buckets com 8 elementos: 1
Buckets com 9 elementos: 1
Buckets com 10 elementos: 0
Buckets com mais de 10 elementos:0
```

1178 listas. Agora começamos a ver uma tendência contrária: Poucas Listas cheias de elementos, com a tendência geral de 0 a 4 elementos em cada lista. Não houve alteração significativa no espaço em memória do programa.

F=1:

```
C:\Users\farof\source\repos\Hashing\x64\Debug\Hashing.exe

Lista 2355:
Nome: Juliana Alves Prates Caminha de Castro || Matricula: 10033556

Lista 2356:
Nome: Ana Paula Picanço Goes || Matricula: 10074978

Buckets com 0 elementos: 874
Buckets com 1 elementos: 865
Buckets com 2 elementos: 424
Buckets com 3 elementos: 149
Buckets com 4 elementos: 31
Buckets com 5 elementos: 11
Buckets com 6 elementos: 3
Buckets com 7 elementos: 0
Buckets com 8 elementos: 0
Buckets com 9 elementos: 0
Buckets com 10 elementos: 0
Buckets com mais de 10 elementos:0
```

Uma lista para cada aluno. Podemos ver a mesma tendência acentuada: nenhuma lista com mais de 6 elementos, com tendência geral ter 0 a 3 elementos, sendo que a grande maioria têm entre 0 a 2.

F=0.5:

```
C:\Users\farof\source\repos\Hashing\x64\Deb

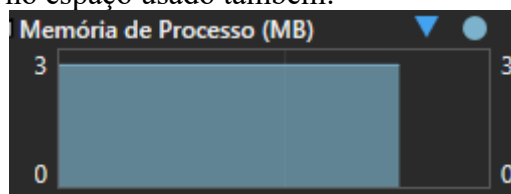
Lista 4711:

Lista 4712:

Lista 4713:

Buckets com 0 elementos: 2885
Buckets com 1 elementos: 1399
Buckets com 2 elementos: 349
Buckets com 3 elementos: 66
Buckets com 4 elementos: 13
Buckets com 5 elementos: 2
Buckets com 6 elementos: 0
Buckets com 7 elementos: 0
Buckets com 8 elementos: 0
Buckets com 9 elementos: 0
Buckets com 10 elementos: 0
Buckets com mais de 10 elementos:0
```

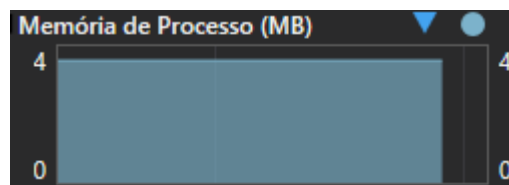
Assim como o universo em que vivemos, a tabela está ficando cada vez mais e mais vazia...
Houve mudança significativa no espaço usado também:



F=0.25:

```
C:\Users\farof\source\repos\Hashing\x64\Deb  
Lista 9425:  
  
Lista 9426:  
  
Lista 9427:  
  
Buckets com 0 elementos: 7471  
Buckets com 1 elementos: 1606  
Buckets com 2 elementos: 305  
Buckets com 3 elementos: 43  
Buckets com 4 elementos: 3  
Buckets com 5 elementos: 0  
Buckets com 6 elementos: 0  
Buckets com 7 elementos: 0  
Buckets com 8 elementos: 0  
Buckets com 9 elementos: 0  
Buckets com 10 elementos: 0  
Buckets com mais de 10 elementos: 0
```

Com 0.25 elementos por lista, podemos ver que a tabela está, realmente, vazia. Muito espaço não



sendo utilizado, mas ao mesmo tempo, elementos muito bem distribuídos, com 1606 tabelas com 1 aluno, 305 com 2 e 43 com 3, ou seja, poucas colisões, o que é ótimo. Contudo, teria sido ainda melhor com uma função hash mais competente, sendo mais específico, como sabemos de antemão todos os alunos que serão inseridos e suas matrículas, poderíamos ter feito perfect hashing, ou seja, alcançado uma tabela hash sem colisão alguma. Não é tão relevante para demonstrar o funcionamento dessa estrutura de dados mas acho importante ressaltar esta possibilidade.