Tabela Hash (Hashing)

- É uma extensão à busca indexada por chave
  - adequa melhor à aplicações de busca mais típicas quando não temos chaves que casam exatamente com os índices.

- Algoritmos de busca baseados em Hashing são divididos de duas partes:

  1) Computar a função hash que transforma a chave de busca em um endereço da tabela
     - Idealmente chaves diferentes resolverão para endereços diferentes
       - Mas é frequente que uma ou mais chaves resolvem para o mesmo endereço

  2) Tratar colisão: processo que processa a questão das chaves que resolvem para o mesmo endereço.
     - Um dos métodos de colisão é o uso de lista encadeada
       - útil em situações que não se prevê o número de chaves de busca
     - Os outros dois métodos de tratamento de colisão obtém melhor resultado com os itens armazenados em um vetor fixo.

- Hashing é um bom exemplo de "time-space tradeoff"
  - Se não tivesse limite de memória, então faríamos a busca com apenas um acesso à memória usando a chave como um endereço de memória, como uma busca indexada pela chave
  - Se não tivesse limite de tempo, usaríamos a menor quantidade de memória e faríamos busca sequencial.

- Hashing provê um balanceamento entre os dois extremos apenas ajustando o tamanho o tamanho da tabela, sem precisar reescrever código ou escolhendo outros algoritmos

- Hashing é um problema clássico de computação: Os algoritmos são bem estudados, em profundidade e diversidade.
  - busca e inserção ficam em tempo constante

- Nem tudo são flores:
  - O tempo de execução depende do tamanho da chave, podendo ser impraticável para chaves muito compridos
  - Não há implementação eficiente para "sort" e "select"

⇒ Funções HASH

- Se temos uma tabela que armazena M itens, precisamos de uma função que transforme chaves em inteiros $[0, M-1]$

- Uma função hash ideal é fácil de computar e se aproxima de uma função aleatória

- A função hash é dependente do tipo da chave
  - precisamos de funções diferentes para tipos diferentes

- Ideia simples para números em ponto flutuante
  - Chaves são números maiores que 0 e menores que 1 basta multiplicar por M e arredondar para o inteiro mais próximo para pegar um endereço entre 0 e $M-1$.
  - Generalizando: As chaves são números em ponto flutuante maiores que "s" e menores que "t" para quaisquer "s" e "t" que sejam fixos.
    - re-escalemos a chave subtraindo por "s" e dividindo por "t-s", colocando entre 0 e 1, então multiplicamos por M.

- Se as chaves são números inteiros de "w" bits, podemos converter para números em ponto flutuante dividindo por 2^w para obter números em ponto flutuante entre 0 e 1, depois multiplicamos por M.

- Se operações em números em ponto flutuante são caras e os números não são muito grandes (a ponto de não causar overflow!), fazemos:

  - Multiplicar a chave por M
  - fazer "right shift" em w-bits e divide por 2^w

- Problema:

  - Só funciona para hash se as chaves forem distribuídos uniformemente no intervalo, pois o valor hash é definido pelos primeiros dígitos de chave

- O algoritmo mais comum e mais simples para chaves inteiras de "w" bits

  - Escolha um tamanho M para a tabela que seja um número primo

  - Para qualquer chave "k", computar o resto dividindo k por M, ou:

$$h(k) = k \bmod M \qquad k \cdot M$$

- também podemos usar hashing modular para chaves em ponto-flutuante.

  Se as chaves estão em um intervalo pequeno, reescalamos para um número entre 0 e 1, multiplicamos por 2^w e, então, fazemos a hash modular.

- E quando a chave é maior que uma palavra? Ou se ela possui parte dos bits fixos?

exemplo: Chaves de 3 caracteres, representado em

3 chars, cada char possui 8 bits, no entanto cada letra é codificada em 7 bits, logo podemos tratar como inteiros de base 128, então a chave

"abc" é codificado como

$$97 \cdot 128^2 + 98 \cdot 128^1 + 99 \cdot 128^0$$

$$= 1589577$$

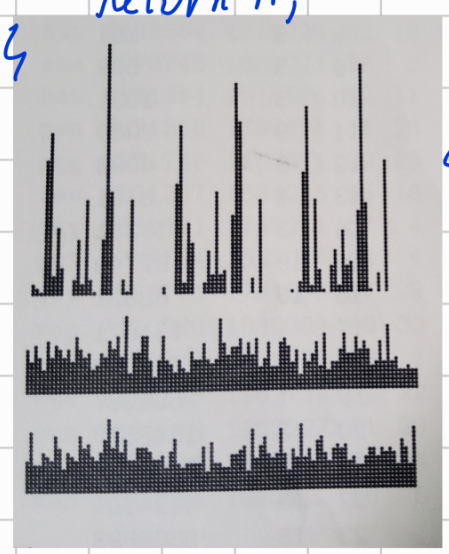
Escolher M como 64 é ruim para esta função pois $(x \bmod 64)$ não é afetado com a soma de múltiplos de 64 (ou 128) a x — a função hash de qualquer chave é o valor dos últimos 6 bits

Há um efeito similar quando a chave é múltiplo de 2

Para evitar isso, devemos usar um número primo

⇒ Função hash para strings

```
int hash (char *v, int M)
{
    int h=0, a=127;
    for(; *v != '\0'; v++)
        h = (a*h + *v) % M;
    return h;
}
```



← M=96, a=128

← M=97, a=128

← M=96, a=127

→ As primeiras 1000 palavras de Moby Dick

Função Hash Universal (para strings)

```
int hashU(char *v, int M)
{
  int h, a = 31415, b = 27183;
  for (h = 0; *v != '\0'; v++, a = a*b % (M-1))
      h = (a*h + *v) % M;
  return h;
}
```

- Valores "aleatórios" para os coeficiente, e um número aleatório diferente para cada dígito de chave

⇒ Sumário

```
#define hash(v, M) (((v-1)/(t-1))*M)
#define hash(v, M) (v % M)
```

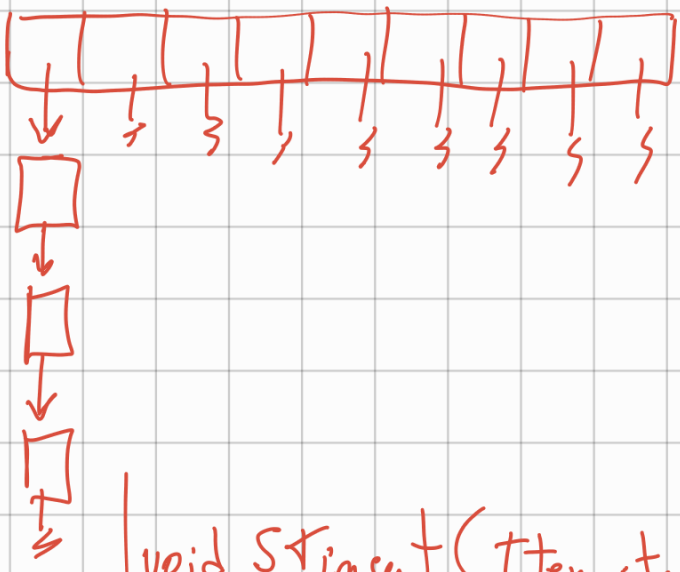
Se M não for primo:

```
#define hash(v, M) ((int)(.616161 * (float) v) % M)
#define hash(v, M) (16161 * (unsigned) v) % M)
```

→ Hash universal pode ser muito mais lento que os métodos mais simples

⇒ Estratégia de Colisão

⇒ Encadeamento Separado

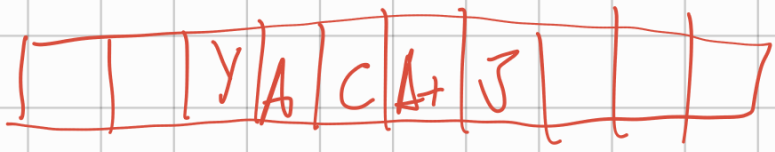


```
link *heads, z;
int M1;
void STinit(int M)
{
  heads = malloc(M * sizeof(*link));
  z = NEW(NULLitem, NULL);
  for(int i=0; i<M; i++)
    heads[i] = NEW(NULLitem, z);
  M1 = M;
}
```

```
void STinsert(Item item)
{
  int h = hash(key(item), M1);
  insertLE(heads[h], item);
}
```

```
Item STsearch(Key k)
{
  int h = hash(k, M1);
  return SearchLE(heads[h], k);
}
```

⇒ Endereçamento Aberto




```
#define hash(v, M)  (v % M)
Item *ht; int M;
void STinit(int Max)
{
    ht = malloc(Max * sizeof(Item));
    M = Max;
    for(int i=0; i< Max; i++)
        ht[i] = NULLItem;
}
```

```
#define MaxCols 10
```

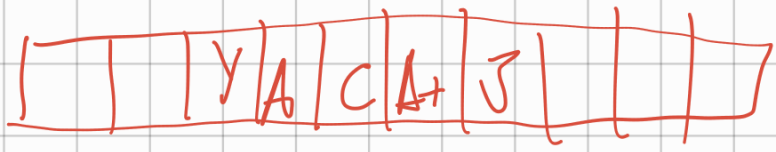

```
int HTinsert(Item ni)
{
  • int h = hash(key(ni), M);
    int c = MaxCol;
    while(c && !eq(key(ht[h]), key(NULLItem))
  •   c--, h = (h+1) % M;
    if(!c)
      return false;
    ht[h] = ni;
    return true;
}
```

```
void STinsert(Item ni)
{
    if(!HTinsert(ni))
        blueScreen();
}
```

```
Item HTsearch(Key k)
{
    int h = hash(k, M);
    int c = MaxCol;
    while(c && !eq(k, ht[h]))
        h = (h+1) % M, c--;

    if(!c)
        return NULLItem;
    return ht[h];
}
```

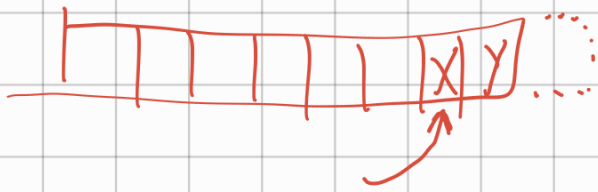
```
void HTRemove(Key k)
{
    int h = hash(k, M);
    int c = MaxCol;
    while(c && !eq(k, ht[h]))
        h = (h+1) % M, c--;

    if(!c) return;
    ht[h] = NULLItem;
}
```

⇒ Double Hesh



- `#define hash(v, M) (v%M)`
  `Item *ht; int M;`
  `void STinit(int Max)`
  `{`
  `ht = malloc(Max * sizeof(Item));`
  `M = Max;`
  `for(int i=0; i<Max; i++)`
  `ht[i] = NULLItem;`
  `}`

```
#define MaxCols 10
#define hashtwo(v) (v%97 +1)
```



`int HTinsert(Item ni)`
`{`

- `int h = hash(key(ni), M); int h2 = hashtwo(key(ni))`
  `int c; int hn = -1;`
  `for(c=MaxCol; c>0; c--, h=(h+h2)%M)`
  `{`
  `if(eq(key(ht[h]), key(NULLItem)))`
  `hn = h;`
  `else if(eq(key(ht[h]), key(ni)))`
  `{ hn = h; break; }`
  `}`
  `if(hn == -1)`
  `return false;`
  `ht[hn] = ni;`
  `return true;`

`}`

```
void STinsert(Item ni)
{
  if(!HTinsert(ni))
    bluescreen();
}
```

```
Item HTsearch(Key k)
{ int h2 = hashtwo(k);
  int h = hash(k, M);
  int c = MaxCol;
  while(c && !eq(k, ht[h]))
      h = (h+h2)%M, c--;

  if(!c)
    return NULLItem;
  return ht[h];
}
```

```
void HTRemove(Key k)
{ int h2 = hashtwo(k)
  int h = hash(k, M);
  int c = MaxCol;
  while(c && !eq(k, ht[h]))
      h = (h+h2)%M, c--;

  if(!c) return ;
  ht[h] = NULLItem;
}
```

```
void STgrow(int Max)
{
  Item *ht2 = malloc(Max * sizeof(Item))
  Item *htvelho = ht; ht = ht2;
  for(int i=0; i<Max; i++)
      ht2[i] = NULLItem;
  int Mvelho = M; M = Max;
  for(int i=0; i<Mvelho; i++)
  {
     if(eq(key(NULLItem), key(htvelho[i])))
        continue;

     HIinsert(htvelho[i]);
  }
  free(htvelho)
}
```