Tabelas *Hash*

- O uso de listas ou árvores para organizar informações é interessante e produz resultados bastante bons.
- Entretanto, em nenhuma dessas estruturas se obtém o acesso direto a alguma informação, a partir do conhecimento de sua chave.
- Uma maneira de organizar dados, que apresenta bons resultados na prática, é conhecida como *hashing*, é baseada na ideia de distribuir os dados em posições aleatórias de uma tabela.
- Uma tabela *hash* é construída através de um vetor de tamanho *n*, no qual se armazenam as informações.
- Nele, a localização de cada informação é dada a partir do cálculo de um índice através de uma função de indexação, a função de *hash.*
- A posição de um elemento é obtida aplicando-se ao elemento a função de *hash* que devolve a sua posição na tabela. Daí basta verificar se o elemento realmente está nesta posição.
- O objetivo então é transformar a chave de busca em um índice na tabela.
- Exemplo: Construir uma tabela com os elementos 34, 45, 67, 78, 89. Supõe-se uma tabela com 10 elementos e uma função de *hash x%10* (resto da divisão por 10).
- -1 indica que não existe elemento naquela posição.

i	0	1	2	3	4	5	6	7	8	9
A[i]	-1	-1	-1	-1	34	45	-1	67	78	89

```
int hash(int x)
{
  return x % 10;
}
void insere(int a[], int x)
{
   a[hash(x)] = x;
}
int busca_hash(int a[], int x)
{
   int k;
   k = hash(x);
   if (a[k] == x) return k;
   return – 1;
}
```

Funções de *Hash*

- Há muitas maneiras de determinar uma função de *hash.*
- **Divisão**
  - Uma função de *hash* precisa garantir que o valor retornado seja um índice válido para uma das células da tabela.
  - A maneira mais simples é usar o módulo da divisão como *h(k) = k%S*, sendo *k* um número e *S* o tamanho da tabela.
  - O método da divisão é bastante adequado quando se conhece pouco sobre as chaves.
- **Enlaçamento**
  - Neste método a chave é dividida em diversas partes que são combinadas ou "enlaçadas" e transformadas para criar o endereço.
  - Existem 2 tipos de enlaçamento: enlaçamento deslocado e enlaçamento limite.
- **Enlaçamento deslocado**
  - As partes da chave são colocadas uma embaixo da outra e processadas.
  - Por exemplo, um código 123-45-6789 pode ser dividido em 3 partes: 123-456-789 que são adicionadas resultando em 1368.
  - Esse valor pode usar o método da divisão *valor%S*, ou se a tabela contiver 1000 posições pode-se usar os 3 primeiros números para compor o endereço.

- **Enlaçamento limite**
  - As partes da chave são colocadas em ordem inversa.
  - Considerando as mesmas divisões do código 123-456-789.
  - Alinha-se as partes sempre invertendo as divisões da seguinte forma 321-654-987.
  - O resultado da soma é 1566.
- **Meio-quadrado**
  - A chave é elevada ao quadrado e a parte do resultado é usada como endereço.
- **Extração**
  - Neste método somente uma parte da chave é usada para criar o endereço.
  - Para o código 123-45-6789 pode-se usar os primeiros ou os últimos 4 dígitos ou outro tipo de combinação como 1289.
  - Somente uma porção da chave é usada.
- **Transformação da raiz**
  - A chave é transformada para outra base numérica.
  - O valor obtido é aplicado no método da divisão *valor%S* para obter o endereço.
- Suponha agora os elementos 23, 42, 33, 52, 12, 58.
- Com a mesma função de *hash*, tem-se mais de um elemento para determinadas posições (42, 52 e 12; 23 e 33).
- Pode-se usar a função *x%17*, com uma tabela de 17 posições.
- A função de *hash* pode ser escolhida à vontade de forma a atender da melhor forma a distribuição.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
-1	52	-1	-1	-1	-1	23	58	42	-1	-1	-1	12	-1	-1	-1	33

- A escolha da função é a parte mais importante.
- É sempre melhor escolher uma função que use uma tabela com um número razoável de elementos.
- No exemplo se houvesse a informação adicional que todos os elementos estão entre 0 e 99, poderia se usar também uma tabela com 100 elementos onde a função de *hash* é o próprio elemento.
- Mas seria uma tabela muito grande para uma quantidade pequena de elementos.
- A escolha da função é um compromisso entre a eficiência na busca o gasto de memória.
- A idéia central das técnicas de *hash* é sempre espalhar os elementos de forma que os mesmos sejam rapidamente encontrados.

Colisões – Lista Linear

- No caso geral, não há informações sobre os elementos e seus valores.
- É comum conhecer apenas a quantidade máxima de elementos que a tabela conterá.
- Problema: Como tratar os elementos cujo valor da função de *hash* é o mesmo? Chamamos tal situação de colisões.
- Para tratar as colisões, pode-se colocar o elemento na primeira posição livre seguinte e considerar a tabela como circular (o elemento seguinte ao último $a[n\text{-}1]$ é o primeiro $a[0]$. Isso se aplica tanto na inserção de novos elementos quanto na busca.
- Esta técnica é conhecida como Lista Linear, *Linear Probing* ou Sondagem.
- Considere os elementos do exemplo anterior e a função *x%10*.

i	0	1	2	3	4	5	6	7	8	9
A[i]	-1	-1	42	23	33	52	12	-1	58	-1

```
int hash(int x) { return x % 10;}
int insere(int a[], int x, int n) {
   int i, cont = 0;
   i = hash(x);
   while (a[i] != -1) // procura a próxima posição
     livre
     {if (a[i] == x) return -1; // valor já existente
     na tabela
      if (++cont == n) return -2; // tabela cheia
       if (++i == n) i = 0;  // tabela circular
   }
    a[i] = x; // achou uma posição livre
   return i;
}
```

```
int busca_hash(int a[], int x, int n) {
   int i, cont = 0 ;
   i = hash(x); // procura x a partir da posição
     i
   while (a[i] != x)
     {if (a[i] == -1) return -1;  // não achou
     x
      if (++cont == n) return -2;  // a tabela
     está cheia
      if (++i == n) i = 0;  // tabela circular
     }// encontrou
   return i;
```

A função de *Hash* – Critérios de escolha

- A operação "resto da divisão por" (módulo – % em C) é a maneira mais direta de transformar valores em índices.
- Exemplos:
  - Se o conjunto é de inteiros e a tabela é de *M* elementos, a função de *hash* pode ser simplesmente *x%M*.
  - Se o conjunto é de valores fracionários entre 0 e 1 com 8 dígitos significativos, a função de *hash* pode ser *floor(x*108)%M.*
  - Se são números entre *s* e *t*, a função pode ser *floor ((x-s)/(t-s)*M)*
- A escolha é bastante livre, mas o objetivo é sempre espalhar ao máximo dentro da tabela os valores da função para eliminar as colisões.
- A função de *hash* deve ser escolhida de forma a atender melhor a particularidade da tabela com a qual se trabalha.
- Os elementos procurados, não precisam ser somente números para se usar *hashing*.
- Uma chave com caracteres pode ser transformada num valor numérico.

Colisões - Duplo hashing ou rehash

- Se a tabela está muito cheia a busca sequencial pode levar a um número muito grande de comparações.
- No pior caso (*N* elementos ocupados), deve-se percorrer os *N* elementos antes de encontrar o elemento ou concluir que ele não está na tabela.
- A grande desvantagem da Lista Linear é o aparecimento de agrupamentos.
- Uma forma de permitir um espalhamento maior é fazer com que o deslocamento em vez de 1 seja dado por uma segunda função de *hash*.
- Essa segunda função de *hash* tem que ser escolhida com cuidado, não deve gerar um valor nulo (*loop* infinito).
- Deve ser tal que a soma do índice atual com o deslocamento (módulo *N*) dê sempre um número diferente até que os *N* números sejam verificados.
- Para isso *N* e o valor desta função devem ser primos entre si.
- Uma maneira é escolher *N* primo e garantir que a segunda função de *hash* tenha um valor *K* menor que *N*. Dessa forma *N* e *K* são primos entre si.
- Existem duas funções de *Hash*: Uma para usar normalmente e outra para usar quando há colisões.
  - $h1(x)=(x \% N) = C_1$
  - $h2(x) = (x \% N-1)+1 = C_2 \rightarrow$ usada quando há colisões
  - Para calcular o primeiro índice usa-se C_1;
  - Para calcular o segundo índice (se existir colisões) usa-se $(C_1+C_2)\%N$;
  - Para calcular o terceiro índice (se também houver colisões) usa-se $(C_1+2C_2)\ \%N$, depois $(C_1+3C_2)\%N$, etc.
- Exemplo: Inserir os elementos 5, 10, 12 e 19 na tabela com *N=7*.

- $h_1(x) = x \% 7 = C_1$
- $h_2(x) = (x \% 6) + 1 = C_2$
  - $5 \rightarrow h_1(5) = 5 \rightarrow$ vazio;
  - $10 \rightarrow h_1(10) = 3 \rightarrow$ vazio;
  - $12 \rightarrow h_1(12) = 5 \rightarrow$ ocupado, faz-se h_2; $\rightarrow 12 \rightarrow h_2(12) = 1 \rightarrow (C_1+C_2)\%7 \rightarrow 6 \rightarrow$ vazio;
  - $19 \rightarrow h_1(19) = 5 \rightarrow 5$ ocupado, faz-se h_2; $\rightarrow 19 \rightarrow h_2(19) = 2 \rightarrow (C_1+C_2)\%7 \rightarrow 0 \rightarrow$ como o *rehash* é circular vai-se inserir no 0;

0	1	2	3	4	5	6
19			***10***		***5***	***12***

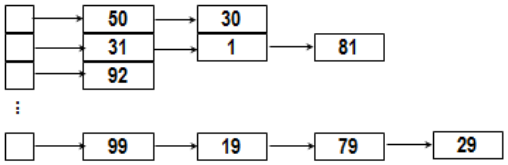
- Como a seleção da segunda função de *hash* é livre pode-se escolher um valor fixo.
- Exemplo : Inserir os elementos 25, 37, 48, 59, 32, 44, 70, 81 (nesta ordem) com N=11.
- $h_1(x) = x \% 11 = C1$
- $h_2(x) = 3 = C2$
  - $25 \rightarrow h_1(25) = 3 \rightarrow$ vazio;
  - $37 \rightarrow h_1(37) = 4 \rightarrow$ vazio;
  - $48 \rightarrow h_1(48) = 4 \rightarrow$ ocupado, $h_2 = 3\ (4+3)=7 \rightarrow$ vazio;
  - $59 \rightarrow h_1(59) = 4 \rightarrow$ ocupado, $h_2 = 3\ (4+3)=7 \rightarrow$ ocupado $(C_1+2C_2)\%N = 10 \rightarrow$ vazio;
- **Exercício: Completar a tabela com os números restantes.**

```
int hash(item x, int N) {
   return ...;  // o valor da função
}
int hash2(item x, int N) {
   return ...;  // o valor da função
}
int insere(item a[], item x, int N) {
   int i = hash(x);
   int k = hash2(x);
   int cont = 0;
   // procura a próxima posição livre
   while (a[i] != -1) {
      if (a[i] == x) return -1; // valor já existente na tabela
      if (++cont == N) return -2; // tabela cheia
      i = (i + k) % N; // tabela circular
   }
   // achou uma posição livre
   a[i] = x;
   return i;
}
int busca_hash(item a[], item x, int N) {
   int i = hash(x);
   int k = hash2(x)
   int cont = 0 ;
   // procura x a partir da posição i
   while (a[i] != x) {
      if (a[i] == -1) return -1;  // não achou x, pois há uma vazia
      if (++cont == N) return -2;  // a tabela está cheia
      i = (i + k) % N; // tabela circular
   }
   // encontrou
   return i;
}
```

Colisões - *Hash* com Lista encadeada

- Podemos criar uma lista ligada com os elementos que tem a mesma chave em vez de deixá-los todos na mesma tabela.
- Com isso pode-se até diminuir o número de chaves geradas pela função de *hash*.

- Tem-se, dessa forma, um vetor de ponteiros.
- Considere uma tabela de inteiros e como função de *hash x%10.*



Colisões

- Cada um dos métodos apresentados tem seus prós e contras:
  - A Lista linear é o mais rápido se o tamanho de memória permite que a tabela seja bem esparsa.
  - O Duplo *hash* usa melhor a memória, mas depende também de um tamanho de memória que permita que a tabela continue bem esparsa.
  - A Lista ligada é interessante, mas precisa de um alocador rápido de memória.
- A escolha de um ou outro depende da análise particular do caso.
- Em particular, a probabilidade de colisão pode ser reduzida usando uma tabela suficientemente grande em relação ao número total de posições a serem ocupadas.
- Por exemplo, uma tabela com 1000 entradas para uma empresa que deseja armazenar 500 posições haveria uma probabilidade de 50% de colisão, se fosse feita a inserção de uma nova chave.
- Considera-se que, em uma tabela *hash* bem dimensionada, devemos ter 1,5 acessos à tabela, em média, para encontrar um elemento.
- Isto corresponde a uma situação em que metade dos acessos é feita diretamente, e, para a outra metade, ocorre uma colisão.

Hash perfeita

- O ideal para a função *hash* é que sejam sempre fornecidos índices únicos para as chaves de entrada.
- A função perfeita (*hash* perfeita) seria a que, para quaisquer entradas A e B, sendo A diferente de B, fornecesse saídas diferentes.
- A tabela deve conter o mesmo número de elementos.
- Nem sempre o número de elementos é conhecido a priori.
- Na prática, funções *hash* perfeitas ou quase perfeitas são encontradas apenas onde a colisão é intolerável (por exemplo, nas funções *hash* da criptografia, ou quando se conhece previamente o conteúdo da tabela armazenada).

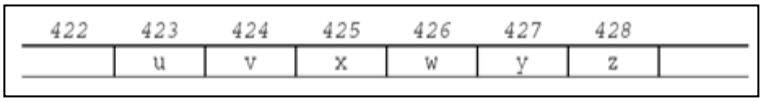
Exercícios

1. Escolha uma boa função de *hash* e o número de elementos da tabela para os números: 1.2, 1.7, 1.3, 1.8, 1.42, 1.51
2. Idem para os números:
   i. 7 números entre 0.1 e 0.9 (1 casa decimal)
   ii. 15 números entre 35 e 70 (inteiros)
   iii. 10 números entre -42 e -5 (inteiros)
3. Idem com um máximo de 1.000 números entre 0 e 1 com no máximo 5 algarismos significativos.
4. Usando a função: *h(k) = k%13* insira as chaves: 18, 41, 22, 44, 59, 32, 31, 73
5. Insira as mesmas chaves usando Double *Hash*ing com as seguintes funções:
   h1(k) =k%13
   h2(k) =8 - k%8

Remoção

- Quando se remove um elemento, a tabela perde sua estrutura de *hash.*
- Suponha que a tabela *hash* a seguir trata colisões por Lista Linear e o elemento *x* é removido.

- Com a remoção de *x* apenas *u* e *v* continuariam acessíveis: o acesso a *w*, *y* e *z* seria perdido.
- Para remover *x*, de forma correta, seria necessário mudar diversos outros elementos de posição na tabela.



- Uma técnica simples é a de marcar a posição do elemento removido como apagada (mas não livre).
- Isso evita a necessidade de movimentar elementos na tabela, mas cria muito lixo.
- Uma melhoria nessa técnica é reaproveitar as posições marcadas como removidas no caso de novas inserções.
- É eficiente se a frequência de inserções for equivalente à de remoções.
- Em casos extremos é necessário refazer o *hashing* completo dos elementos.
- Embora permita o acesso direto ao conteúdo das informações, o mecanismo das tabelas *hash* possui uma desvantagem em relação a listas e árvores.
- Numa tabela *hash* é virtualmente impossível estabelecer uma ordem para os elementos, ou seja, a função de *hash* faz indexação, mas não preserva ordem.
- Avaliar uma boa função *hash* é um trabalho difícil e relacionado à estatística.
- Dependendo da aplicação utras estruturas devem ser levadas em conta.

Exercício para apresentar em sala

Elaborar um programa contendo uma função de *hash* que recebe uma *string* e devolve um valor numérico, calculado a partir da *string*

O programa deverá ter as seguintes opções:

- **Inserção:** Entrada de uma *string*, calcula o valor a partir da função de *hash* adotada e apresenta o valor, insere na tabela *hash*.
- **Busca:** Se o valor estiver na tabela apresentar a posição senão informar que a *string* não se encontra na tabela.
- **Remoção**: Remover a *string* da tabela, se esta estiver na tabela.

Observações:

- **A tabela deverá ter tamanho 10.**
- Projete uma função de *hash* adequada, tentando evitar colisões, pode-se usar o *double hashing*.