Tabelas *Hash*

Tabelas Hash

- O uso de listas ou árvores para organizar informações é interessante e produz bons resultados.
  - Porem, em nenhuma dessas estruturas se obtém o acesso direto a alguma informação, a partir do conhecimento de sua chave.
- Hashing é uma maneira de organizar dados que:
  - apresenta bons resultados na prática,
  - distribui 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**.

Tabelas Hash

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

- 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.

Tabelas *Hash*

```
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.

Funções de Hash

- **Enlaçamento deslocado**
  - As partes da chave são colocadas uma embaixo da outra e processadas.
  - Por exemplo, um código 123-456-789 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.
  - 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.

Funções de Hash

- **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 123-456-789 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.

Tabelas Hash

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

- 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) para a função x%10.
- 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**.

Tabelas Hash

- 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-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.

Colisões – Lista Linear

- 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

- Esta forma de tratamento de colisões tem uma desvantagem que é:
  - a tendência de formação de grupos de posições ocupadas consecutivas,
  - fazendo com que a primeira posição vazia, na prática, possa ficar muito longe da posição original, dada pela função de hash.

- Para inserir um determinado valor **x** na tabela, ou para concluir que o valor não se encontra na tabela, é necessário encontrar a primeira **posição vazia** após a posição h(x).

Colisões

```
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 bemlivre, mas o objetivo é sempre espalhar ao máximo dentro da tabela os valores da função para eliminar as colisões.

A função de *Hash*

- 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.

Colisões - Duplo *hashing* ou *rehash*

- 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.
  - $h1(x)=(x \% N) = C1$
  - $h2(x) = (x \% N-1)+1 = C2$ → usada quando há colisões
  - Para calcular o primeiro índice usa-se *C1*;
  - Para calcular o segundo índice (se existir colisões) usa-se $(C1+C2) \%N$;
  - Para calcular o terceiro índice (se também houver colisões) usa-se $(C1+2C2) \%N$, depois $(C1+3C2) \%N$, etc.

Colisões - Duplo *hashing* ou *rehash*

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

Exemplo: Inserir os elementos 5, 10, 12 e 19 na tabela com *N=7*.

- *h1(x) = x % 7 = C1*
- *h2(x) = (x % 6) +1 = C2*
  - *5* → *h1 (5) = 5* → *5* está vazio*;*
  - *10* → *h1 (10) = 3* → *3* está vazio*;*
  - *12* → *h1 (12) = 5* → *5* está ocupado, faz-se *h2;*
    12 → *h2 (12) = 1* → *(C1+C2)%7* → *6* está vazio;
  - *19* → *h1 (19) = 5* → *5* está ocupado, faz-se *h2;*
    19 → *h2 (19) = 2* → *(C1+C2)%7* → *0* → como o *rehash* é circular vai-se inserir no 0;

Colisões - Duplo *hashing* ou *rehash*

- 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*.
- *h1(x) = x % 11 = C1*
- *h2(x) = 3 = C2*
  - *25 → h1 (25) = 3 →* vazio*;*
  - *37 → h1 (37) = 4 →* vazio*;*
  - *48 → h1 (48) = 4 →* ocupado, *h2 = 3 (4+3)=7 →* vazio*;*
  - *59 → h1 (59) = 4 →* ocupado, *h2 = 3 (4+3)=7 →* ocupado *(C1+2C2)%N = 10 →* vazio*;*
- **Exercício**: Completar a tabela com os números restantes.

Colisões - Duplo *hashing* ou *rehash*

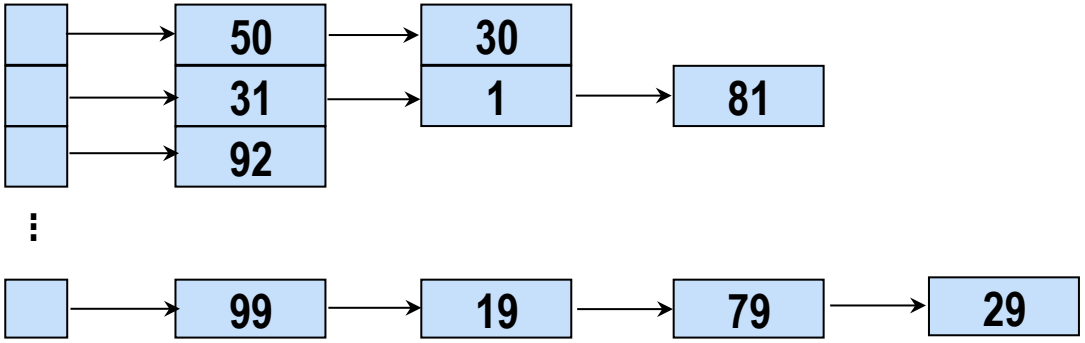
```
int hash(item x, int N) {
   return ...;  // o valor da função
}
int hash2(item x, int N) {
   return ...;  // o valor da função
}
int insere(item a[], item x, int N) {
   int i = hash(x);
   int k = hash2(x);
   int cont = 0;
   // procura a próxima posição livre
   while (a[i] != -1) {
      if (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;
}
```

Colisões - Duplo *hashing* ou *rehash*

```
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*.



- A função pode ser melhorada:
  - Só inserir se já não estiver na tabela. Para isso deve-se percorrer a lista até o final;
  - Inserir elemento de modo que a lista fique em ordem crescente.

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.

Colisões

- 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 uma tabela *hash* bem dimensionada,
  - Média 1,5 acessos à tabela para encontrar um elemento.

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:
  - 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.

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.

422	423	424	425	426	427	428	
	u	v	x	w	y	z	

- 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.

Remoção

- 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.

Tabelas *Hash*

- 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.
  - 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 outras estruturas devem ser levadas em conta.

Exercício

- Faça um programa que crie uma lista com Duplo *hashing, com as funções:*
- *Inserção de um elemento*
  - *Deve mostrar a posição e quando ocorre colisão e o novo hash.*
- *Procura de um elemento*
  - *Deve mostrar a necessidade de novo hash.*