Algoritmos de Busca em Tabelas - Hash

Técnicas de Hashing

Outra forma de se fazer busca em uma tabela, é construir-se a tabela de forma a facilitar a busca, colocando-se cada elemento numa posição pré-determinada. Tal posição é obtida aplicando-se ao elemento uma função (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. Vamos usar uma tabela com 10 elementos e a função de hash `x%10` (resto da divisão por 10). A tabela ficaria.

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

`-1` indica que não existe elemento naquela posição.

```
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;
}
```

Suponha agora os elementos 23, 42, 33, 52, 12, 58.
Com a mesma função de hash, teremos mais de um elemento para determinadas posições (42, 52 e 12; 23 e 33). Podemos usar a função **`x%17`**, com uma tabela de 17 posições.

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

Observe que a função de hash pode ser escolhida a vontade de forma a atender da melhor forma a distribuição. É sempre melhor escolher uma função que use uma tabela com um número razoável de elementos. No exemplo acima, se tivéssemos a informação adicional que todos os elementos estão entre 0 e 99, podemos 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 a parte mais importante.
É um compromisso (trade-off) entre a eficiência na busca o gasto de memória.

Hash e Espalhamento

A idéia central das técnicas de hash é sempre **espalhar** os elementos de forma que os mesmos sejam rapidamente encontrados. Haverão elementos vazios na tabela, mas isso não é um problema. Estamos gastando mais memória mas aumentando a eficiência dos algoritmos de busca.

Colisões

No caso geral, não temos informações sobre os elementos e seus valores. É comum sabermos somente a quantidade máxima de elementos que a tabela conterá.
Assim, ainda existe um problema a ser resolvido. Como tratar os elementos cujo valor da função de hash é o mesmo? Chamamos tal situação de colisões.

Existe uma forma simples de tratar o problema das colisões. Basta colocar o elemento na primeira posição livre seguinte e considerar a tabela 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.

Considere os elementos acima e a função **`x%10`**. A tabela ficaria:

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);
   /* 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 */
      if (++i == n) i = 0;  /* tabela circular */
   }
   /* achamos uma posição livre */
   a[i] = x;
   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, pois há uma vazia */
      if (++cont == n) return -2;  /* a tabela está cheia */
      if (++i == n) i = 0;  /* tabela circular */
     }
   /* encontrou */
   return i;
}
```

Função de Hash – exemplos

tabela com máximo de 1000 elementos entre 0 e 1		tabela com máximo de 1000 elementos entre 0 e 999 (inteiros)		
valor	x*1000000%1000	valor	x%111	x%1000
0,00274393	743	451	7	451
0,31250860	508	49	49	49
0,93238920	389	33	33	33
0,36855872	558	67	67	67
0,97586401	864	20	20	20
0,26181556	815	486	42	486
0,00374296	742	340	7	340
0,36952102	521	451	7	451
0,02565065	650	483	39	483
0,68874429	744	123	12	123
0,79637276	372	78	78	78
0,76964729	647	85	85	85
0,76121084	210	411	78	411
0,63431038	310	11	11	11
0,66531025	310	465	21	465
0,79660979	609	493	49	493
0,14684603	846	453	9	453
0,45284238	842	200	89	200
0,78491416	914	362	29	362
0,58450957	509	145	34	145
0,85097057	970	396	63	396
0,16247350	473	25	25	25
0,15729077	290	38	38	38

A função de hash

A operação "resto da divisão por" (módulo – % em C) é a maneira mais direta de transformar valores em índices. Exemplos:

- Se tivermos um conjunto de inteiros e uma tabela de M elementos, a função de hash pode ser simplesmente **`x%M`**.
- Se tivermos um conjunto de valores fracionários entre 0 e 1 com 8 dígitos significativos, a função de hash pode ser **$x*10^8\%M$**.

Se são números entre s e t, a função pode ser **`((x-s)/(t-s))*(M-1)`.**

A escolha é bastante livre, mas o objetivo é sempre espalhar ao máximo dentro da tabela os valores da função. Ou seja, eliminar ao máximo as colisões.

Função de Hash – outras considerações

A função de hash deve ser escolhida de forma a atender melhor a particular 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.
Vejamos uma função de hash que recebe um string e devolve um valor numérico, calculado a partir do string (soma dos valores numéricos dos caracteres):

```
int hash(unsigned char a[], int n) {
   int k, i, s = 0;
   k = strlen(a);
   for (i = 0; i < k; i++) s = s + i * abs(a[i]);
   return s % n;
}
```

Outro exemplo:

```
int hash(unsigned char a[], int n) {
   int k;
   k = strlen(a);
   return (k * 17 + k * 19) % n;
}
```

O tratamento de colisões

Lista Linear – busca sequencial

O tratamento das colisões pode ser feito como mostrado anteriormente, isto é, todos os elementos compartilham a mesma lista e se ao inserir um elemento a posição estiver ocupada, procura-se a próxima posição livre à frente. Sempre considerando a lista de forma circular.
No pior caso, no qual a lista está completa (N elementos ocupados), teremos que percorrer os N elementos antes de encontrar de encontrar o elemento ou concluir que ele não está na tabela.
O algoritmo é O(N).

Lista Linear – duplo hashing

Quando a tabela está muito cheia a busca seqüencial pode levar a um número muito grande de comparações antes que se encontre o elemento ou se conclua que ele não está na tabela.
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 pode dar zero (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. A expressão `(j*K) % N (j=0, 1, 2,..., N-1)` gera todos os números de `0` a `N-1`.
Exemplo: Considere uma tabela com 11 elementos (11 é primo) e passo 3 (valor da segunda função de hash). Supondo que o valor da primeira função de hash seja 4, a sequência de posições por (4+3*j)%11 será:

4+3*j	(4+3*j)%11
4	4
7	7
10	10
13	2
16	5
19	8
22	0
25	3
28	6
31	9
34	1

Exemplo: Considere agora uma tabela com 11 elementos (N=11), primeira função de hash = **x%N** e segunda função de hash = **3.**
Inserir os seguintes elementos na tabela: 25, 37, 48, 59, 32, 44, 70, 81 (nesta ordem)

0	1	2	3	4	5	6	7	8	9	10
81	-1	32	25	37	44	-1	48	70	-1	59

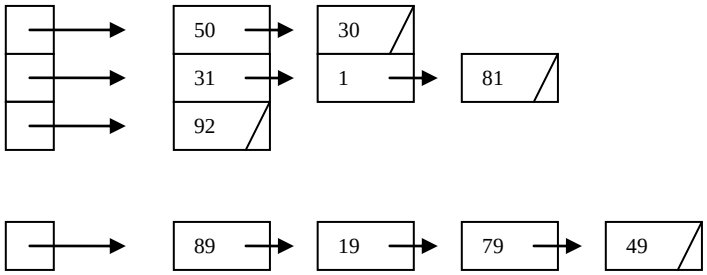
Vamos agora aos algoritmos de duplo hash:

```
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 */
   }
   /* achamos 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;
}
```

O duplo hash espalha mais os elementos, mas no pior caso ainda será necessário percorrer a tabela inteira e o algoritmo continua sendo O(N).

Hash com Listas ligadas

Podemos criar uma lista ligada com os elementos que tem a mesma chave em vez de deixá-los todos na mesma tabela. Com isso podemos até diminuir o número de chaves possíveis geradas pela função de hash. Com isso teremos um vetor de ponteiros.
Considere uma tabela de inteiros e como função de hash x%10.




```
typedef struct item * link;
struct item {
   int info;
   link  prox;
};

static link head;

void inicia(int M) {
   int i;
   head = malloc(M*sizeof(link);
   for (i=0;i<M;i++) head[i] = NULL;
}

int hash(int x) {
   return x%10;
}

void insere(int x) {
   int k = hash(x);
   link s = malloc(sizeof item));
   /* insere no início da lista */
   s->prox = head[k];
   head[k] = s;
}

item procura(int x) {
   int k = hash(x);
   link s = head[k];
   /* procura item com info = x */
   while (s != NULL)
        if (x == s->info) return s;
   return NULL;
}
```

A função **`insere`** poderia ser melhorada:

- Só inserir se já não estiver na tabela. Para isso seria necessário percorrer a lista até o final;
- Inserir elemento de modo que a lista fique em ordem crescente;

Com essa solução usa-se um espaço extra de `M` posições, mas em média, cada uma das listas terá apenas `N/M` posições, supondo que os elementos se distribuem uniformemente na lista. Sem dúvida, representa um ganho

em relação a solução anterior, onde para se achar um elemento teremos que percorrer uma lista de N posições no pior caso.
Entretanto no pior caso, todos os `N` elementos estão na mesma lista (a função de hash não foi bem escolhida) e o tal algoritmo continua sendo `O(N)`.

Tabelas dinâmicas

A busca numa tabela de hash de qualquer tipo fica mais demorada à medida que a tabela fica mais cheia. Enquanto a tabela está com poucos elementos, a busca é sempre muito rápida, qualquer que seja o tipo da tabela de hash.

Uma solução para o problema de tabela muito cheia é aumentar o seu tamanho quando começar a ficar cheia. O aumento da tabela não é uma operação eficiente, pois todos os elementos devem ser reinseridos. Entretanto, pode valer a pena se esta operação é realizada poucas vezes.

Vamos exemplificar com uma Lista Linear, mas o mesmo vale para Hashing Duplo e Lista Ligada.
Adotando a seguinte convenção: quando a tabela passa de `N/2` elementos, dobramos o seu tamanho. Assim sempre a tabela terá menos da metade de seus elementos ocupados.

```
int N;    /* tamanho da tabela */
int NN;   /* novo tamanho da tabela */
int M;    /* quantidade de elementos da tabela */
item * p; /* ponteiro para o início da tabela */
item * q; /* ponteiro para o início da nova tabela */

int hash(item x, int T) {
   /* o valor de hash é também função do tamanho da tabela T. */
   /* ao expandirmos a tabela, a posição dos elementos será mudada */
   return ...;
}

#define VAZIO -1
/* preenche todas as posições da tabela com o valor VAZIO */
void limpa(item * r, int m) {
  int k;
  for (k = 0; k < m; k++) r[k] = VAZIO;
}

/* inicia as variáveis */
void inicia() {
   N = 1000;  /* valor inicial */
   M = 0;
   p = malloc(N*sizeof(item));
   limpa(p, N);
}

int insere_nova(item x) {
   int i = hash(x, NN);
   /* procura a próxima posição livre */
   /* sempre tem lugar pois a tabela foi duplicada */
   while (q[i] != VAZIO) {
     i = (i+1) % NN; /* tabela circular */
   }
   /* achamos uma posição livre */
   q[i] = x;
   return i;
```

```
}

void expande() {
    int i, NN = 2*N;
    q = malloc(NN*sizeof(item));
    limpa(q, NN);
    /* insere todos os elementos na tabela nova */
    for (i = 0; i < N; i++)
      if (p[i] != VAZIO) insere_nova(p[i]);
    N = NN;
    free(p);
    p = q;
}

void insere(item x) {
   int i;
   if (N < M+M) expande();
   i = hash(x, N);
   /* procura a próxima posição livre */
   while (p[i] != VAZIO) {
      if (p[i] == x) return -1; /* valor já existente na tabela */
      i = (i+1) % N; /* tabela circular */
   }
   /* achamos uma posição livre */
   p[i] = x;
   M++;
   return i;
}

int busca_hash(item x) {
   int i = hash(x, N);
   int cont = 0;
   /* procura x a partir da posição i */
   while (p[i] != x) {
      if (p[i] == VAZIO) return -1;  /* não achou x, pois há uma vazia */
      if (++cont == M) return -2;  /* já verificou todos os elementos */
      i = (i+1)%N; /* tabela circular */
   }
   /* encontrou */
   return i;
}
```

Comentários

Cada uma dos métodos acima de resolver o problema das colisões tem seus prós e contras:

- Lista linear é o mais rápido se o tamanho de memória permite que a tabela seja bem esparsa.
- Duplo hash usa melhor a memória, mas depende também de um tamanho de memória que permita que a tabela continue bem esparsa.
- Lista ligada é interessante, mas precisa de um alocador rápido de memória.

A escolha de um ou outro depende da análise particular do caso.

A grande vantagem do hash é que se as chaves são razoavelmente conhecidas e se dispormos de memória suficiente, podemos encontrar uma função de hash que possibilite um tempo constante para o algoritmo. Portanto O(1), o que é melhor que os métodos que vimos até agora que são O(N) ou O(lg N).

Sobre a remoção de elementos

Considere a estrutura de lista linear ou mesmo duplo hash. Quando se remove um elemento, a tabela perde sua estrutura de hash. Portanto, remoções não podem ser feitas.
Isso não ocorre com a estrutura de lista ligada.

Para que uma tabela de busca (hash ou não) seja dinâmica, isto é, permita inserções e remoções juntamente com a busca, a estrutura de dados deve ser outra. Para esses casos deve-se usar uma estrutura de árvores de busca.

Exercícios:

1. Deseja-se construir uma tabela do tipo hash com os números:

1.2 1.7 1.3 1.8 1.42 1.51

Diga qual seria uma boa função de hash e o número de elementos da tabela.

2. Idem para os números:

1.2 1.3 1.8 5.3 5.21 5.7 5.43 8.3 8.4 8.47 8.8

3. Idem para os seguintes números:

7 números entre 0.1 e 0.9 (1 casa decimal)
15 números entre 35 e 70 (inteiros)
10 números entre -42 e -5 (inteiros)

4. idem com um máximo de 1.000 números entre 0 e 1 com no máximo 5 algarismos significativos.

5. Idem com um máximo de 1.000.000 de números entre 0 e 1