Algoritmos de Busca em Tabelas - Hash

Técnicas de Hash

Outra forma de se fazer busca rápida em uma tabela, é dividi-la em várias sub-tabelas de tal forma que seja possível determinar a priori em qual sub-tabela estará cada elemento. Assim, a busca se restringe a uma quantidade menor de elementos.

Uma simplificação desta técnica é determinar a posição do elemento dentro da tabela através de seu valor. Definimos então uma função que transforma o valor do elemento numa posição da tabela (função de hash). Basta verificar se o elemento está ou não 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]	None	None	None	None	34	45	None	67	78	89

`None` – indica que não existe elemento naquela posição.

```
def hash(x):
  return x % 10


def insere(a, x):
   a[hash(x)] = x


def busca_hash(a, x):
   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 mudar a função de hash para **`(x % 17)`**, com uma tabela de 17 posições. Os elementos ocuparão posições distintas.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a[i]	None	52	None	None	None	None	23	58	42	None	None	None	12	None	None	None	33

Observe que a função de hash pode ser escolhida de forma a atender da melhor 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. 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

Voltando agora a ideia de dividir a tabela em sub-tabelas, podemos **espalhar** os elementos numa mesma tabela de forma que a mesma seja vista pelo algoritmo como sub-tabelas distintas. Haverá 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]	None	None	42	23	33	52	12	None	58	None

Vamos aos algoritmos. Chamemos de **`M`** o tamanho da tabela:

```
def hash(x, M):
   return x % M

def insere_hash(a, x):
    M = len(a)
    cont = 0
    i = hash(x, M)
    # procura a próxima posição livre
    while a[i] != None:
        if a[i] == x: return -1 # valor já existente na tabela
        cont += 1               # conta os elementos da tabela
        if cont == M: return -2 # tabela cheia
        i = (i + 1) % M         # tabela circular
    # achamos uma posição livre - coloque x nesta posição
    a[i] = x
    return i

def busca_hash(a, x):
    M = len(a)
    cont = 0
    i = hash(x, M)
    # procura x a partir da posição i
    while a[i] != x:
        if a[i] == None: return -1  # não achou x, pois há uma vazia
        cont += 1                   # conta os elementos da tabela
        if cont == M: return -2;    # a tabela está cheia
        i = (i + 1) % M             # tabela circular
    # encontrou
    return i
```

Função de Hash – exemplos

A escolha da função de hash é livre. A orientação a ser seguida é sempre o compromisso entre o tamanho máximo e a quantidade estimada de comparações necessárias para se encontrar o elemento. 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 – `%`) 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 forem números entre s e t, a função pode ser ((x - s) * (M – 1)) / (t - s) arredondando ou truncando para o inteiro mais próximo.

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 hash. Uma chave com caracteres pode ser transformada num valor numérico.
Vejamos uma função de hash que recebe um string **a** e o tamanho da tabela **M** e devolve um valor numérico, calculado a partir do string:

```
def hash(a, M):
    s = 0
    # s conterá a soma dos valores numéricos dos caacteres
    for chr in a:
        s = s + ord(chr)
    return s % M
```

Outro exemplo:

```
def hash(a, M):
    # função de hash baseada no tamanho da string
    k = strlen(a)
    return (k * 17 + k * 19) % M
```

O tratamento de colisões

Lista linear simples de hash – 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 (**M** elementos ocupados), teremos que percorrer os **M** elementos antes de encontrar de encontrar o elemento ou concluir que ele não está na tabela.
O algoritmo é O(**M**).

Entretanto, quando se tem alguma informação sobre os elementos, é possível construir uma função de hash sobre uma tabela que busque minimizar o número de comparações para se encontrar um inserir um novo elemento. É uma situação bem melhor que a simples busca sequencial. Note que melhor que isso seria fazer busca binária, mas para isso os dados teriam que chegar em ordem. Não é o caso geral. Os dados podem ser inseridos a medida que chegam, numa tabela de hash.

Lista linear – duplo hash

Quando a tabela está muito cheia a busca sequencial 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 **M**) dê sempre um número diferente até que os **M** números sejam verificados. Para isso **M** e o valor desta função devem ser **primos entre si**.

Uma maneira é escolher **M** primo e garantir que a segunda função de hash tenha um valor constante **K** menor que **M** e maior que 1. Dessa forma **M** e **K** são primos entre si. A expressão **(j * K) % M (j=0, ..., a M -1)** gera todos os números de 0 a **M** -1. O mesmo ocorre com a expressão **(c + j * K) % M**, onde **c** é uma constante.

Exemplo: Considere uma tabela com 11 elementos (11 é primo) e passo 3 (valor da segunda função de hash). Supondo que o valor da função de hash para o elemento procurado seja 4, a sequência de posições determinada pela expressão (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 (**M**=11), primeira função de hash = **x % M** e segunda função de hash = **3.**
Inserir os seguintes elementos na tabela: 25, 37, 48, 59, 32, 44, 70, 81 (nesta ordem).

i	0	1	2	3	4	5	6	7	8	9	10
a[i]	44	None	32	25	37	70	None	48	81	None	59

Vamos agora aos algoritmos de duplo hash:

```
def hash(x, M):
   return ...  # valor da função

int hash2():
   return ...  # valor da função - passo

def insere_hash(a, x):
    M = len(a)
    cont = 0
    i = hash(x, M)
    k = hash2()
    # procura a próxima posição livre
    while a[i] != None:
        if a[i] == x: return -1 # valor já existente na tabela
        cont += 1               # conta os elementos da tabela
        if cont == M: return -2 # tabela cheia
        i = (i + k) % M         # tabela circular
    # achamos uma posição livre - coloque x nesta posição
    a[i] = x
    return i

def busca_hash(a, x):
    M = len(a)
    cont = 0
    i = hash(x, M)
    k = hash2()
    # procura x a partir da posição i
    while a[i] != x:
```

```
        if a[i] == None: return -1  # não achou x, pois há uma vazia
        cont += 1                   # conta os elementos da tabela
        if cont == M: return -2;    # a tabela está cheia
        i = (i + k) % M             # 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(M).

Hash com sub-listas do Python

Usando o fato que os elementos de uma lista do Python podem ser mistos (elementos simples e outras listas), podemos construir uma tabela hash contendo sub-listas com os elementos que têm o mesmo valor da função de hash. Usando como exemplo uma lista com 23, 42, 33, 52, 12, 58 (inseridos nesta ordem) em uma tabela de 10 elementos e **`(x % 10)`** como função de hash, a lista contendo a tabela ficaria:

```
[None, None, [42, 52, 12], [23, 33], None, None, None, None, 58, None]
```

As funções de inserção e busca ficariam:

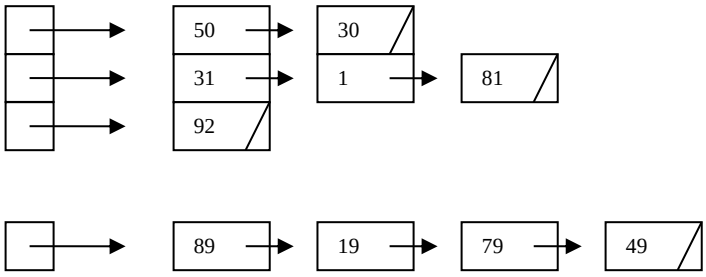
```
# Função de hash
def hash(x, M):
   return x % M

# Insere x na tabela de hash a
#   a[i] é um elemento da tabela ou uma outra lista
#   contendo os elementos com a mesmo valor de hash
# Devolve (None, None) - não inseriu porque já estava
#         (i, None) - se inseriu em a[i]
#         (i, j) - se inseriu em a[i][j]
def insere_hash(a, x):
   M = len(a)
   i = hash(x, M)
   # tentar inserir x na tabela
   if a[i] is None:
      a[i] = x
      return (i, None)
   # se a[i] é uma lista
   if type(a[i]) is list:
      # procura x em a[i]
      if x in a[i]:
          return (None, None) # x já está na tabela
      # pode inserir x na lista a[i]
      k = len(a[i])
      a[i].append(x)
      return (i, k)
   # a[i] é um elemento simples
   if a[i] == x:
      return (None, None) # já está
   # iniciar a lista em a[i] e inserir elemento
   a[i] = [a[i], x]
   return(i, 1)

# Procura x na tabela de hash a
# Devolve (None, None) - se x não está na tabela
```

```
#          (i, None) - se x == a[i]
#          (i, j) - se x == a[i][j]
def busca_hash(a, x):
   M = len(a)
   i = hash(x, M)
   # x não está na tabela
   if a[i] is None:
      return (None, None)
   # se a[i] é uma lista
   if type(a[i]) is list:
      # procura x em a[i]
      for k in range(len(a[i])):
         if x == a[i][k]:
            return (i, k)
      # não encontrou
      return (None, None)
   # a[i] é um elemento simples
   if a[i] == x:
      return (i, None) # encontrou
   # não encontrou
   return(None, None)
```

Hash com Listas ligadas

Uma lista ligada pode ser criada com os elementos que têm o mesmo valor da função de hash, em vez de deixá-los todos na mesma tabela. O efeito é o mesmo da solução anterior. Diminuir o número de comparações para se procurar um elemento. Assim, teremos uma lista com as referências iniciais de cada lista ligada.

Como exemplo, considere uma tabela de inteiros e como função de hash **`(x % 10)`**.



A classe **`TabelaHashListaLigada`** abaixo, implementa essa estrutura:

```
# classe _Node
class Node:
   def __init__ (self, info, prox):
      # inicia os campos
      self._info = info
      self._prox = prox

class TabelaHashListaLigada:

    # métodos da classe
```

```
    def __init__ (self, M):
        ''' cria M LLs vazias.'''
        self._numheads = M
        self._head = [None] * M

    def hash(self, x, M):
        return x % M

    def insere(self, e):
        ''' adiciona elemento no inicio da LL
            adiciona sempre - inclusive se já estiver.'''
        # novo nó referencia o inicio da LL
        k = self.hash(e, self._numheads)
        novo = Node(e, self._head[k])
        # novo nó será o inicio da LL
        self._head[k] = novo

    def procura(self, e):
        ''' procura elemento com info = e.
            devolve referência para esse elemento ou None se não acha.'''
        k = self.hash(e, self._numheads)
        # p percorre a lista
        p = self._head[k]
        while p is not None:
            if p._info == e: return p # achou
            p = p._prox # vai para o próximo
        # se chegou aqui é porque não achou
        return None

    def MostraTabHash(self):
        ''' mostra cada uma das listas.'''
        for k in range(self._numheads):
            print("\nLista ", k)
            p = self._head[k]
            while p is not None:
                print("***", p._info)
                p = p._prox        prox
```

O exemplo abaixo cria um objeto deste tipo, insere e procura alguns elementos:

```
# cria tabela de valores
tabela = [34, 54, 89, 98, 134, 85,99]
print("tabela de elementos:\n", tabela)

# Cria tabela hash ligada com NumLL LLs e insere valores
NumLL = 5
TabHash = TabelaHashListaLigada(NumLL)

# insere os elementos de tabela
for k in tabela:
    TabHash.insere(k)

# mostra a tabela construida
TabHash.MostraTabHash()

# teste - procura elementos
```

```
print("\nteste - procura elementos")
if TabHash.procura(33) is None:
   print("não achou")
if TabHash.procura(55) is None:
   print("não achou")
for elem in tabela:
   if TabHash.procura(elem) is None:
      print(elem, " * não achou")
   else:
      print(elem, " * achou")
```

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 para a tabela de referências, mas em média, se temos N elementos, cada uma das listas terá apenas `N/M` posições. Entretanto no pior caso, todos os `N` elementos estão na mesma lista (se 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 tende a ser sempre muito rápida.

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

Vamos também usar variáveis globais na implementação abaixo. O mesmo poderia ser feito, criando um Tipo de Dado Abstrato Tabela_Hash.

```
# Inicia as variáveis globais da tabela hash dinâmica
M = 100     # tamanho da tabela
MM = 0      # novo tamanho da tabela
N = 0       # quantidade de elementos da tabela
p = [None] * M    # esta é a tabela de hash em uso
q = []      # nova tabela de hash

def hash(x, T):
   # devolve a função de hash para o elemento x e tabela de tamanho T
   return x % T

# insere novamente x na nova tabela q
def insere_nova(x):
   global M, MM, N, p, q
   i = hash(x, MM);
   # procura a próxima posição livre
   # sempre tem lugar, pois a tabela foi duplicada
   while q[i] != None:
      i = (i + 1) % MM # tabela circular
   # achamos uma posição livre
```

```
   q[i] = x
   return i

# expande a tabela para o dobro do tamanho
def expande():
   global M, MM, N, p, q
   MM = 2 * M
   q = [None] * MM  # nova lista com o dobro de elementos
   # insere todos os elementos na tabela nova
   for i in range(M):
      if p[i] != None: insere_nova(p[i])
   # novos valores para M e p
   M = MM
   p = q  # p e q são a mesma lista

# insere elemento na tabela hash
def insere(x):
   global M, MM, N, p, q
   # verifica se o tamanho da tabela está adequado
   if M < N + N:
      # se já tem a metade cheia, é melhor expandir
      expande()
   i = hash(x, M);
   # procura a próxima posição livre - sempre vai encontrar
   while p[i] != None:
      # encontrou - insere x se já não está
      if p[i] == x: return -1    # valor já existente na tabela
      i = (i + 1) % M            # tabela circular
   # achamos uma posição livre
   p[i] = x
   N += 1
   return i;

def busca_hash(x):
   global M, MM, N, p, q
   i = hash(x, M)
   cont = 0
   # procura x a partir da posição i
   while p[i] != x:
       if p[i] == None: return -1  # não achou x, pois há uma vazia
       cont += 1                   # conta os elementos da tabela
       if cont == N: return -2;    # esse é o último – não achou
       i = (i + 1) % M             # tabela circular
   # encontrou
   return i;
```

Comentários

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

- Lista de hash simples: 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 ou lista com sub-lista: é interessante, mas precisa de um alocador rápido de memória.

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

A grande vantagem do hash é que se as chaves são razoavelmente conhecidas e se dispomos de memória suficiente, podemos encontrar uma função de hash que possibilite um tempo constante ou aproximadamente 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 simples de hash ou mesmo dupla hash. Quando se remove um elemento, a tabela perde sua estrutura de hash. Portanto, remoções não podem ser feitas.

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. É necessário usar-se lista ligada. Dentre as estruturas usando lista ligada, as melhores são as árvores de busca que serão vistas posteriormente.

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 fracionários entre 0 e 1 com no máximo 10 dígitos significativos.