Tabelas Hash em Java

Por: Ben Tindale

Traduzido por: Marcio Montenegro
16 de Maio de 2001, para Gazeta do Linux

Tabelas Hash em Java

Uma tabela hash é conceitualmente uma seção de memória contígua com um número de elementos endereçáveis, normalmente chamados de bins, nos quais dados podem ser rapidamente inseridos, apagados e encontrados. Tabelas hash representam uma perda de memória em benefício da velocidade - certamente não é o método mais eficente em uso de memória para armazenar dados, porém a velocidade de busca é bastante elevada. Tabelas hash são uma maneira usual de organizar dados, assim os projetistas do Java disponibilizaram uma série de classes para criar e manipular facilmente instâncias de tabelas hash.

Hashtable é a classe que permite a criação de tabelas hash em Java. Esta classe é uma herança direta da classe Dictionary e implementa as interfaces Map, Cloneable e Serializable. A implementação da interface Map é uma novidade do Java 1.2. Você pode acessar a documentação sobre Hashtable aqui.

Uma chave é um valor que pode ser mapeado para um dos elementos endereçáveis da tabela hash. A linguagem Java disponibiliza uma interface para geração de chaves para um conjunto de classes: Como exemplo, o trecho de código abaixo imprime a representação em chave de uma string para uso posterior em uma tabela hash.

```
String abc = new String("abc");
System.out.println("Key for \"abc\" is "+ abc.hashCode());
```

Função de hashing é aquela que realiza uma operação, a partir de um dado, em que uma chave é gerada e será usada para identificar em qual elemento de memória da tabela o dado será colocado. Existem algumas propriedades que uma função de hashing deve possuir para uso eficiente de tabelas hash:

- Os dados devem estar dispersos de maneira mais aleatória possível através da tabela para minimizar as chances de colisão. Por exemplo, uma função de hashing razoável colocaria a letra ``b'' distante da letra ``a''.
- A função de hash deve ter um tempo de execução razoável.

Infelizmente, como iremos ver posteriormente, as funções de hashing disponibilizadas pelo Java não satisfazem o primeiro critério.

O fator de carga (load fator) de uma tabela hash é definido como a relação entre o número de bins ocupados e o número de bins disponíveis. Um bin está ocupado quando contém um elemento ou apontador para este. O fator de carga é um parâmetro útil usado para estimar a probabilidade de ocorrência de uma colisão. A linguagem Java aloca mais memória para uma tabela hash quando o fator de carga for maior que 75%. O usuário pode optar por escolher o fator de carga inicial da tabela visando reduzir o numero de métodos "rehashing" necessários. O código abaixo demonstra este procedimento.

```
int initialCapacity = 1000;
float loadFactor = 0.80;
Hashtable ht = new Hashtable(initialCapacity, loadFactor);
```

Caso necessite alocar mais espaço para sua tabela hash, antes que o fator de carga atinja o valor especificado, utilize o método *rehash()* como mostrado abaixo:

```
ht.rehash();
```

Uma colisão acontece quando a mesma chave é atribuida a dois elementos distintos pela função de hashing. Como o uso de tabela hash implica em maximizar a eficiência que os dados são inseridos, apagados ou encontrados, as colisões devem ser evitadas tanto quanto possível. Se você conhece uma função de hashing usada para criar uma chave, então você pode usá-la facilmente para criar colisões. Por exemplo, o código java abaixo mostra como duas strings diferentes podem possuir o mesmo hash. (arquivo texto)

```
import Java.util.*;
import Java.lang.*;

// x + 31x = x(31 + 1) = x + 31 + 31(x-1)
public class Identical
{
    public static void main(String[] args)
    {
        String s1 = new String("BB");
        String s2 = new String("Aa");
        System.out.println(s1.hashCode());
        System.out.println(s2.hashCode());
    }
}
```

Este código gera o seguinte resultado em minha instalação RedHat 6.2, usando o compilador java kaffe .

```
[bash]$ javac Identical.java
[bash]$ java Identical
2112
2112
[bash]$
```

O método "chaining" trata colisões em uma tabela hash imaginando cada bin como um ponteiro para uma lista encadeada de elementos de dados. Quando ocorre uma colisão, o novo elemento é simplesmente inserido na lista de alguma maneira. De modo similar, uma tentativa de remover um elemento de um

bin com mais de uma entrada, envolve uma pesquisa na lista até que o elemento encontrado seja igual ao dado a ser removido. Não existe necessidade de negociar a ocorrência de colisões usando listas - uma estrutura de dados como uma árvore binária também poderia ser usada. A classe HashTable do Java usa este método para inserir elementos em uma tabela hash. Um aspecto interessante é que com este método podemos ter uma tabela hash com fator de carga maior que 100%.

O endereçamento livre ocorre quando todos os elementos em uma tabela hash são armazenados na própria tabela - não há ponteiros para listas encadeadas: Cada "bin" guarda somente um único dado. Esta é a maneira mais simples de implementar uma tabela hash, visto que a tabela se reduz a um array onde os elementos podem ser inseridos ou apagados de qualquer posição do índice em um instante qualquer.

Teste Linear é uma forma de implementar endereçamento livre escolhendo o próximo bin disponível, caso ocorra uma colisão durante a tentativa de inserir um dado. Cada bin subsequente será testado para verificar a ocorrência de colisões antes de inserir o dado.

A classe String contém um método hashCode() usado para gerar uma chave que pode ser usada em uma tabela hash. O hashcode para um objeto String é calculado pela expressão
*s[0]*31^(n-1)+s[1]*31^(n-2)+...+s[n-1]*
usando aritmética para números inteiros onde *s[i]* e o *i*ésimo caracter de uma string de tamanho *n*. O hash de uma string nula é definido como zero.

Incluí um programa exemplo chamado CloseWords que procura em um dicionário, palavras próximas do argumento da linha de comandos. Para isto o programa utiliza explicitamente uma característica da função hashing da classe String: A tendência em agrupar palavras que possuem uma composição alfanumérica similar. Esta é uma característica indesejável porque sendo o dado de entrada composto por um conjunto limitado de caracteres, haverá um grande número de colisões. A função de hashing ideal deve distribuir os dados randomicamente na tabela sem que haja uma tendência de aglomeração.

Outra limitação do método hashCode é que com o uso de um tipo inteiro para representar uma chave, os projetistas do Java limitaram a magnitude possível de uma chave em *2^32 -1*. Isto significa uma probabilidade de colisões muito maior que usando uma chave de 64 bit.

A classe Hashtable e os métodos fornecidos na Java Foundation Classes são uma ferramenta poderosa para manipulação de dados - particularmente quando é necessário uma rápida recuperação, busca ou descarte dos dados. Para um conjunto de dados muito grande, entretanto, a implementação das funções hashing em Java tende a causar um agrupamento - isto torna a execução desnecessariamente lenta. Uma melhor implementação de uma tabela hash implica numa função de hash que distribua os dados de forma mais aleatória e o uso de um tipo de dado maior para a chave.

Links e referências

Para um discussão completa sobre limitações de tabelas hash em Java e uma implementação mais eficiente, veja.

A linguagem Java possui uma excelente documentação - confira no site da SUN.

Para mais informações sobre o compilador Java de código aberto Kaffe visite o site.

CloseWords

Atenção: O código fonte pode ser obtido aqui

```
import java.lang.*;
import java.util.*;
import java.io.*;

/** CloseWords: Utilizando a tendência de aglutinação do método nativo hasCode
 * na classe string para encontrar palavras "próximas" ao argumento.
 */
public class CloseWords
{
    Hashtable ht;
    String currString;

    /** No código abaixo é criada uma instância da classe Hashtable para armazenar
     * o hash de todas as palavras de um dicionário ( claro que é uma maneira
     * ineficiente em termos de uso de memória, de indexar as palavras ).
     *
     * @param args
     */
    public static void main(String[] args)
    {
        ht = new Hashtable();
        try
            {
                DataInputStream in = new DataInputStream(
                                                    new BufferedInputStream(
                                                                        new FileInputStream("/usr/dict/words")));
                while((currString = in.readLine())!=null)
                    ht.put(new Integer(currString.hashCode()), currString);

                int i = args[0].hashCode();
                int found=0;

                while(found < 5)
                    {
                        i--;
                        if(ht.get(new Integer(i))!=null)
                            {
                                System.out.println(ht.get(new Integer(i)));
                                found++;
                            }
                    }
                i = args[0].hashCode();
                found=0;
```

```
                while(found < 5)
                    {
                        i++;
                        if(ht.get(new Integer(i))!=null)
                            {
                                System.out.println(ht.get(new Integer(i)));
                                found++;
                            }
                        }
                }
            catch(IOException ioe)
                {
                        System.out.println("IO Exception");
                }
    }
}
```

Publicado ni Número 57 da *Linux Gazette*, Setembro 2000