Hashing

Referências

- Hashing (PF),
- Hash Tables (S&W),
- slides (S&W)

Vídeo

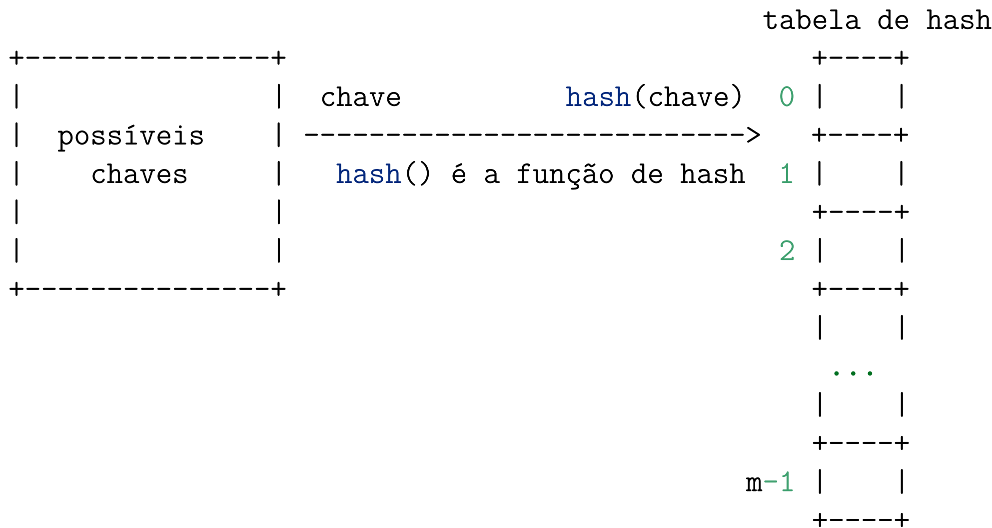
Hashing Functions (S&W)

Tabelas de hash

Uma **tabela de hash** é de certa forma uma generalização de um vetor.

Temos um universo U de possíveis chaves e desejamos construir um esquema que associa cada chave a idealmente uma posição de um vetor ou tabela onde armazenamos o valor associado a chave.

O esquema ou função que associa as chaves a índices desta tabela é chamada de **função de hash**.



Alguns parâmetros importantes

Parâmetros importantes:

- `m` = número de posições na tabela de hash
- `n` = número de chaves da tabela de símbolos
- α = `n/m` = fator de carga (=*load factor*)

Funções de hashing

Função de espalhamento ou **função de hashing** (=*hash function*): transforma cada chave em um índice da tabela de hash. A função de hashing responde a pergunta "Em que posição da tabela de hash devo colocar

esta chave?". A função de hashing espalha as chaves pela tabela de hash. A função de hashing associa um valor hash (hash value), entre 0 e `m-1`, a cada chave.

Infelizmente, nem sempre é possível associar cada elemento a um índice diferente. Tipicamente o conjunto de possíveis chaves é muito grande. Para isso precisaríamos de muito espaço.

Funções injetoras são raras. O paradoxo do aniversário diz que em um grupo de 23 pessoas a probabilidade que não haja duas que façam aniversário no mesmo dia é 0.4927 (menor que meio). Em outras palavras, se selecionar uniformemente ao acaso uma função de 23 elementos em um conjunto com 365 a probabilidade dessa função não ser injetora é maior que meio.

A função de hashing produz uma **colisão** quando duas chaves diferentes têm o mesmo valor hash e portanto são levadas na mesma posição da tabela de hash:

Para usar uma tabela de hash, programadores devem tomar duas decisões:

- escolher uma função de hash `hash()` que dada uma chave retorna um inteiro
- selecionar um método para tratar colisões

Chaves, usualmente, tem grande redundância. Devemos ser cuidadosos para encontrar uma função de hash que quebre *clusters* de chaves quase idênticas a fim de reduzir o número de colisões.

Testes sobre conjuntos típicos de chaves têm mostrado que dois tipos de função de hash funcionam muito bem. Um baseado em divisão e o outro em multiplicação.

Função de hashing modular

A ideia mais usual para a função de hashing é escolher o tamanho `M` da tabela como dendo um primo e função é dado por

```
private int hash(int key) {
    return key % M;
}
```

A função é fácil de calcular e usando um primo, uma boa dispersão das chaves é obtida. Isso se deve ao fato de que

> se `m` não é primo nem todos os bits do número são usados no cálculo da função

Para `m = 1000` só os últimos 3 dígitos importam.

Exemplos com m = 100 e com m = 97:... "'

hashCode() de Java

Em Java, todo tipo-de-dados tem uma método padrão `hashCode()` que produz um inteiro entre -2^{31} e $2^{31}-1$, Exemplo:

```
String s = StdIn.readString();
int h = s.hashCode();
```

Exemplo:

```
public class Teste {
     private int valor;
     public Teste(int valor) {
```

```
        this.valor = valor;
    }
}
```

```
Welcome to DrJava.  Working directory is /home/coelho/mac0323
> Teste t = new Teste(5)
> t.hashCode()
22767675
> Teste t = new Teste(6)
> t.hashCode()
27103358
> Teste r = new Teste(27)
> r.hashCode()
5836093
>
```

`hashCode()` deve ser consistente com `equals()`:

- se `a.equals(b) == true`, então `a.hashCode() == b.hashCode()`

Para converter o `hashCode()` em um número entre 0 e `m-1`, tome o resto da divisão por `m`. Antes, é melhor desprezar o bit mais sigificativo para evitar que `%` lide com números negativos e produza um resultado negativo:

```
    private int hash(Key x) {
        return (x.hashCode() & 0x7fffffff) % M;
    }
```

O uso da máscara `0x7fffffff` seria desnecessário se Java tivesse um tipo-de-dados `unsigned int`.

Chaves `Integer`

Em Java o valor de hash de um int é ele mesmo;

```
public class Integer {
    private final int value;

    public int hashCode() {
        return value;
    }
}
```

Chaves `Double`

Para `doubles` o Java usa hashing modular na representação binária do `double`.

```
public class Double {
    private final double value;

    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >> 32));
```

```
    }
}
```

Chaves `Boolean`

```
public class Boolean {
    private final boolean value;

    public int hashCode() {
        if (value) return 1231;
        return 1237;
    }
}
```

Chaves `String`

Consideramos o string como um número muito grande na base `R` (= 31).

Se `s` é um `String` de comprimento `k`, `s.hashCode()` é o valor

$$\mathtt{s}[0] \times 31^{\mathtt{k}-1} + \mathtt{s}[1] \times 31^{\mathtt{k}-2} + \ldots + \mathtt{s}[\mathtt{k}-2] \times 31 + \mathtt{s}[\mathtt{k}-1].$$

O método `hashCode()` utiliza o método de Horn para calcular esse valor.

```
int h = 0;
for (int i = 0; i < s.length(); i++)
    h = (31 * h + s.charAt(i)) % M;
```

No lugar do multiplicador 31, poderia usar qualquer outro inteiro `R`, de preferência primo, mas suficientemente pequeno para que os cálculos não produzam overflow.

```
public class String {
    private final int hash; // caching
    private final char[] s;

    public int hashCode() {
        int h = hash;
        if (h != 0) return h;
        for (int i = 0; i < length(); i++) {
            h = s[i] + (31 * h);
        }
        hash = h
        return h;
    }
}
```

Mais exemplos

```
Welcome to DrJava.  Working directory is /home/coelho/mac0323/
> "a".hashCode()
```

```
97
> "b".hashCode()
98
> "b67612876".hashCode()
1746433347
> "Como é bom estudar MAC0323!".hashCode()
-638314223
> ("Como é bom estudar MAC0323!".hashCode())&0x7fffffff
1509169425
> (1.23&0.7fffffff)
Invalid top level statement
> ("Como é bom estudar MAC0323!".hashCode())&0x7fffffff
1509169425
> "Como é bom estudar MAC0323!".hashCode()
-638314223
> "Este é um string".hashCode()
90150209
> Integer i = 0
> i.hashCode()
0
> Float x = 3.1415926
Static Error: Bad types in assignment: from double to Float
> Double x = 3.1415926;
> x.hashCode()
219937201
> Float x = 3.14
Static Error: Bad types in assignment: from double to Float
> Float x = (float)3.14
> x.hashCode()
1078523331
> Float x = (float)3.1415926
> x.hashCode()
1078530010
>
```

O que se espera de uma função de hashing

Queremos uma função de hashing que:

- possa ser calculada eficientemente e
- espalhe bem as chaves pelo intervalo `0,1,...,m-1`.

Hipótese do Hashing Uniforme: Vamos supor que nossas funções de hashing distribuem as chaves pelo intervalo de inteiros `0..M-1` de maneira uniforme (todos os valores hash igualmente prováveis) e independente.

Isso significa que se as cada chave `k` é escolhida de um universo `U` de acordo com uma distribuição de probabilidade `P`; ou seja, `P(k)` é a probabilidade de k ser escolhida. Então a hipótese do hashing uniforme nos diz que

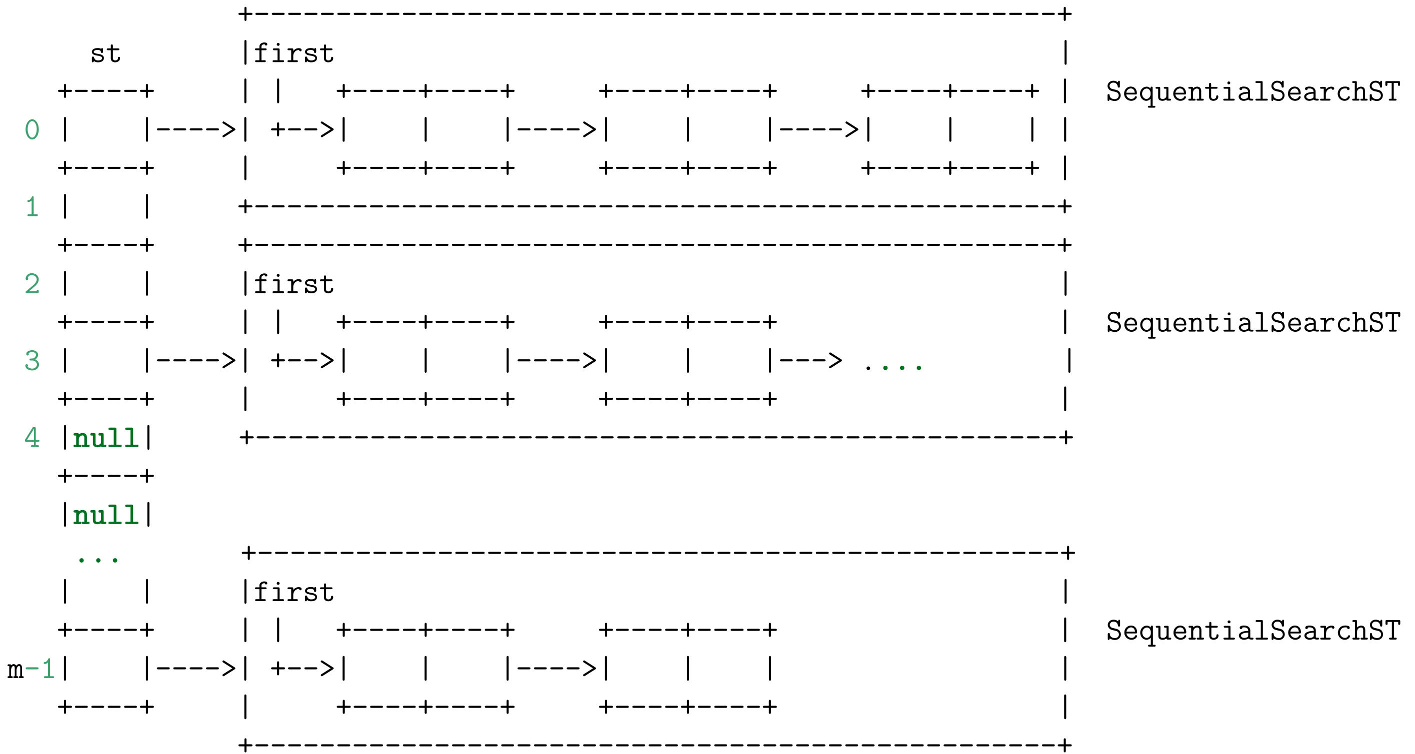
$$\sum_{k:h(k)=j} P(k) = \frac{1}{m}$$

para `j = 0,1,2,...,m-1`.

Colisões: encadeamento

Resolução de colisões por encadeamento (*separate chaining*): `m` listas ligadas, cada uma implementa uma tabela de símbolos.

Classe `SequentialSearchST`: implementação de tabela de símbolos em uma lista ligada não ordenada:




```
public class SeparateChainingHashST<Key, Value> {

    private int n;  // número de chaves
    private int m;  // tamanho da tabela de hash

    private SequentialSearchST<Key,Value>[] st;  // vetor de TSs

    public SeparateChainingHashST(int m) {
        this.m = m;
        st = (SequentialSearchST<Key,Value>[]) new SequentialSearchST[M];
        for (int i = 0; i < m; i++)
            st[i] = new SequentialSearchST<Key,Value>();
    }

    public SeparateChainingHashST() {
        this(997); // tamanho de tabela default
    }

    private int hash(Key key) {
```

```
        return (key.hashCode() & 0x7fffffff) % m;
    }

    public Value get(Key key) {
        int h = hash(key);
        return st[h].get(key);
    }

    public void put(Key key, Value val) {
        int h = hash(key);
        st[h].put(key, val);
    }
}
```

Redimensionamento da tabela com encadeamento

```
    // insert key-value pair into the table
    public void put(Key key, Value val) {
        if (val == null) { delete(key); return; }

        // double table size if average length of list >= 10
        if (n >= 10*m) resize(2*m);

        int i = hash(key);
        if (!st[i].contains(key)) n++;
        st[i].put(key, val);
    }

    // resize the hash table to have the given number of chains
    // rehashing all of the keys
    //
    // redimensiona a tabela de espalhamento de modo que ela tenha
    // chains listas ligadas e reinsere todas s chaves
    private void resize(int chains) {
        SeparateChainingHashST<Key, Value> temp;
        temp = new SeparateChainingHashST<Key, Value>(chains);
        for (int i = 0; i < m; i++) {
            for (Key key : st[i].keys()) {
                temp.put(key, st[i].get(key));
            }
        }
        this.m  = temp.m;
        this.n  = temp.n;
        this.st = temp.st;
    }

    // delete key (and associated value) if key is in the table
    public void delete(Key key) {
        int i = hash(key);
```

```
        if (st[i].contains(key)) n--;
        st[i].delete(key);

        // halve table size if average length of list <= 2
        if (m > INIT_CAPACITY && n <= 2*m) resize(m/2);

    }

    // return keys in symbol table as an Iterable
    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        for (int i = 0; i < m; i++) {
            for (Key key : st[i].keys())
                queue.enqueue(key);
        }
        return queue;
    }
```

Análise

O comprimento médio das listas é `n/m`. Mas poderíamos ter uma lista muito longa e todas as demais muito curtas, não? Para eliminar essa possibilidade, precisamos saber ou supor algo sobre os dados.

Proposição K: Em uma tabela de hash encadeada com `m` listas e `n` chaves, se vale a hipótese do hashing uniforme, a probabilidade de que o número de chaves em cada lista não passa de `n/m` multiplicado por uma pequena constante, essa probabilidade é muito próxima de 1.

Exemplo: Se `n/m = 10`, a probabilidade de que uma lista tem comprimento maior que 20 é inferior a `0.009`.