1 - Introdução

Considere a necessidade de armazenar um conjunto de números inteiros positivos numa estrutura para fazer buscas. Deseja-se que essa estrutura tenha tempo constante para inserir e buscar um valor. Porém considere que os números a serem apresentados são desconhecidos, aleatórios, com repetições e serão apresentados em um fluxo sequencial - um a um.
Considere as duas opções a seguir para manter os números a serem apresentados:

(i) Manter os números em um array, onde o valor a ser armazenado será o índice do array. O valor de cada elemento do array indicará a quantidade de vezes que o valor foi apresentado.
- Vantagem: a recuperação é imediata, uma vez que o índice determina a posição na memória;
- Desvantagens: é necessário saber qual é o maior valor a ser apresentado e várias posições do array ficarão vazias, gerando desperdício de memória, principalmente quando os valores assumirem a casa dos bilhões.

(ii) Manter os números em uma lista encadeada.
- Vantagem: otimização de memória;
- Desvantagens: o tempo de inserir um novo elemento na lista aumenta à medida que novos valores são adicionados, pois valores repetidos precisam ser contabilizados. Outro problema é que o tempo de busca não é constante.

2 - Tabela de Dispersão

Uma tabela de dispersão - também conhecida por tabela de espalhamento ou tabela hash, do inglês hash - é uma estrutura de dados especial, que associa chaves de pesquisa a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado (https://pt.wikipedia.org/wiki/Tabela_de_dispersão).
Termos importantes sobre hashing (espalhamento):

- Chaves: são os valores a serem armazenados;
- R: é o universo de chaves, que é o conjunto de todas as possíveis chaves disponíveis na aplicação. Podemos dizer que qualquer chave possui um valor no intervalo [0, R-1];
- M: é o conjunto das chaves efetivamente usadas pela aplicação, é, em geral, apenas uma pequena parte do universo de chaves. Nessas circunstâncias, faz sentido usar uma tabela de tamanho bem menor que o tamanho do universo de chaves;
- Tabela hash (tabela de dispersão): é uma estrutura de dados usada para manter as chaves, ela associa chaves a valores, onde cada chave será um índice da tabela;
- Função hash (função de espalhamento): é um algoritmo que mapeia (transforma) uma chave, do universo de chaves R, para um índice da tabela hash, que é um valor no intervalo [0, M-1]. A função hash *espalha* as chaves pela tabela hash;
- Colisões: acontece quando a função hash mapeia duas chaves diferentes para o mesmo valor hash e, portanto, são levadas para a mesma posição da tabela hash.

O conjunto das chaves efetivamente usadas em uma determinada aplicação é, em geral, apenas uma pequena parte do universo de chaves R. Desta forma, faz sentido usar uma tabela de tamanho bem menor que o tamanho do universo de chaves.

Considere que a nossa tabela tenha M índices, no formato tabela[0, M-1]. O papel da função de espalhamento é mapear qualquer chave no conjunto de índices tabela[0, M-1]. A função de espalhamento

recebe uma chave ch e devolve um número inteiro hc no intervalo [0, M-1].

O número hc é o código de espalhamento (hash code) da chave ch.

Características de uma função de espalhamento:

- É fundamental que a função de espalhamento devolva sempre o mesmo hc para a mesma ch, pois a função de espalhamento será usada no momento de inserir e buscar;
- Uma boa função de espalhamento espalha as chaves uniformemente pelo conjunto de índices evitando colisões.

Uma função de espalhamento popular para números inteiros positivos é obter o resto da divisão, assim como mostrado na Figura 1. Neste exemplo, a tabela de espalhamento terá 10 índices (Figura 2), mas veja que existem colisões nos índices 4, 7 e 8, enquanto que os índices 0, 3, 6 e 9 não possuem chaves, ou seja, conclui-se que a função de espalhamento ch%10 não produz um bom espalhamento. Não existe um regra para a escolha da função de espalhamento, mas recomenda-se usar um número primo se a função envolver módulo.

ch	hc = ch % 10
21	1
12	2
14	4
74	4
25	5
17	7
87	7
98	8
8	8

Figura 1 – Resultado da função de espalhamento.

índice	chaves
0	
1	21
2	12
3	
4	14, 74
5	25
6	
7	17, 87
8	98, 8
9	

Figura 2 – Tabela de dispersão do exemplo da Figura 1.

Existem vários algoritmos para tratar as colisões na tabela de dispersão, a seguir serão mostrados dois algoritmos simples.

3 - Espalhamento com Lista Encadeada

Uma solução popular para resolver colisões é conhecida como *separate chaining*: para cada índice hc da tabela há uma lista encadeada que armazena todas as chaves da função de espalhamento que resulta em hc. Essa solução é boa se as listas encadeadas forem curtas.

A Figura 3 e Figura 4 mostra uma implementação da tabela de dispersão usando lista encadeada para resolver os problemas de colisão. O exemplo da Figura 5 mostra o uso da classe HashEncadeada para manter 50 números aleatórios no intervalo [0,99] e a Figura 6 mostra um exemplo de resultado. Neste exemplo, a chave 11 ocorreu 2 vezes.

```
public class No {
   public int chave, ocorrencia;
   public No proximo;
```

```
    @Override
    public String toString(){
       return "("+chave +":"+ ocorrencia+")";
    }
}
```

Figura 3 – Código da classe No.

```
public class HashEncadeado {
   private int m; /* número de elementos da tabela */
   private No tabela[]; /* tabela de dispersão */

   public HashEncadeado(int m){
      this.m = m;
      this.tabela = new No[m];
   }

   /* Converte a chave em um valor [0,m-1].
    * O valor retornado será a posição da chave na tabela de dispersão.
    * Essa função é chamada de função de espalhamento, pois ela mapeia um valor (chave) do
    * intervalo [0,R-1] para o intervalo [0,m-1], onde m é menor que R */
   public int hash(int ch){
      return ch % m;
   }

   /* As colisões na tabela de dispersão podem ser resolvidas usando listas encadeadas:
    * todas as chaves que têm um mesmo código hash são armazenadas na lista */
   public void inserir(int ch){
      /* cria um nó */
      No no = new No();
      no.chave = ch;
      no.ocorrencia = 1;
      /* obtém a posição na tabela de dispersão */
      int hc = hash(ch);
      if( tabela[hc] == null ){ /* lista vazia */
         tabela[hc] = no;
      }
      else{
         No aux = tabela[hc];
         /* busca o final da lista ou um valor repetido */
         while( aux.proximo != null && aux.chave != ch ){
            aux = aux.proximo;
         }
         if( aux.chave == ch ){
            aux.ocorrencia++;
         }
         else{
            aux.proximo = no;
         }
      }
   }

   public void imprimir(){
      No aux;
      /* percorre o array */
      for(int i = 0; i < tabela.length; i++){
         System.out.print( i + ": ");
         /* percorre a lista encadeada que está na posição i do array */
         aux = tabela[i];
         while( aux != null ){
            System.out.print( aux + " ");
            aux = aux.proximo;
         }
         System.out.println();
      }
   }
}
```

Figura 4 – Código da classe HashEncadeado.

```java
public class Principal {
   public static void main(String[] args) {
      int nro;
      HashEncadeado hash = new HashEncadeado(11);
      for( int i = 0; i < 50; i++ ){
         nro = (int) (Math.random()*100);
         hash.inserir(nro);
      }
      hash.imprimir();
   }

}
```

Figura 5 – Código da classe Principal.

```
0: (11:2) (88:2) (66:1) (44:1) (99:1)
1: (56:2) (67:1) (23:1)
2: (24:1) (90:1) (57:2)
3: (3:1) (25:2) (14:2) (47:1)
4: (37:1) (48:1) (26:1) (70:1) (81:1)
5: (16:3) (49:1) (60:2) (38:1)
6: (28:1)
7: (40:1) (51:2)
8: (74:2) (63:1) (85:1)
9: (86:1) (97:1) (75:1)
10: (10:3) (54:1) (87:1) (76:1)
```

Figura 6 – Resultado da classe Principal da Figura 5.

4 - Espalhamento com Sondagem Linear

O conceito do algoritmo de sondagem linear (linear probing) é “se uma posição da tabela estiver ocupada, então tente a próxima”.

O algoritmo consiste em obter o hc (hash code) de uma chave e fazer M tentativas – conhecidas como sondagens – para encontrar uma posição livre na tabela. Existem as seguintes situações na busca de uma posição na tabela:

- A posição de hc está livre na tabela de dispersão – neste caso o algoritmo coloca a chave na posição hc da tabela;
- A posição de hc já possui a mesma chave – neste caso o algoritmo incrementa o número de ocorrências da chave;
- A posição de hc possui uma chave diferente – neste caso o algoritmo busca a próxima posição vazia e coloca a chave nesta posição. A busca fracassa somente se a tabela estiver cheia.

A Figura 7 e Figura 8 mostra uma implementação da tabela de dispersão usando sondagem linear para resolver os problemas de colisão. O exemplo da Figura 9 mostra o uso da classe HashSondagem para manter 8 números e a Figura 10 mostra o resultado. A função irá falhar quando a tabela estiver cheia.

```java
public class Registro {
   public int chave, ocorrencia;

   public Registro(int chave){
      this.chave = chave;
   }

   @Override
   public String toString(){
      return chave +" "+ ocorrencia;
   }
}
```

Figura 7 – Código da classe Registro.

```java
public class HashSondagem {
   private int m; /* número de elementos da tabela */
   private Registro tabela[]; /* tabela de dispersão */

   public HashSondagem(int m){
      this.m = m;
      this.tabela = new Registro[m];
   }

   public int hash(int ch){
      return ch % m;
   }

   public void inserir(int ch){
```

```
        int hc = hash(ch), tentativa;
        /* quantidade de tentativas */
        for( tentativa = 0; tentativa < m; tentativa++ ){
           /* quando a posição está vazia no array */
           if( tabela[hc] == null ){
              tabela[hc] = new Registro(ch);
              tabela[hc].ocorrencia++;
              break;
           }
           /* quando a chave já existe no array */
           else if( tabela[hc].chave == ch ){
              tabela[hc].ocorrencia++;
              break;
           }
           hc = (hc + 1) % m; /* próxima posição na tabela de dispersão */
        }
        if( tentativa >= m ){
           /* falha quando a tabela de dispersão está cheia */
           System.out.println("Falhou: "+ ch);
        }
     }

     /* retorna null se a chave não existir */
     public Registro buscar(int ch){
        int hc = hash(ch);
        while (tabela[hc] != null){
           if (tabela[hc].chave == ch){
              return tabela[hc];
           }
           hc = (hc + 1) % m; /* próxima posição na tabela de dispersão */
        }
        return null;
     }

     public void imprimir(){
        for(int i = 0; i < tabela.length; i++){
           System.out.println( tabela[i] );
        }
     }
}
```

Figura 8 – Código da classe HashSondagem.

```
public class Principal {
   public static void main(String[] args) {
      HashSondagem hash = new HashSondagem(10);
      hash.inserir(53);
      hash.inserir(36);
      hash.inserir(57);
      hash.inserir(21);
      hash.inserir(17);
      hash.inserir(46);
      hash.inserir(56);
      hash.inserir(73);
      hash.inserir(17);
      hash.imprimir();
      System.out.println("Buscar:" );
      System.out.println("17: "+ hash.buscar(17) );
      System.out.println("27: " + hash.buscar(27) );
      System.out.println("36: " + hash.buscar(36) );
   }
}
```

Figura 9 – Código da classe Principal.

```
56 1
21 1
null
53 1
73 1
null
36 1
57 1
17 2
46 1
Buscar:
17: 17 2
27: null
36: 36 1
```

Figura 10 – Resultado da classe Principal da Figura 9.

5 - Exercícios

1 – Alterar o método inserir da classe HashEncadeado (Figura 4) para que os valores sejam colocados de forma ordenada na lista encadeada.

2 – Programar o método buscar(nro:int):boolean na classe HashEncadeado. Ele deverá retornar true se o nro existir na tabela de dispersão.

3 – Programar o método remover(nro:int):void na classe HashEncadeado. Ele deverá remover o elemento que possui o nro da tabela de dispersão.