Universidade Federal do Piauí – UFPI

Departamento de Computação/CCN

Disciplina: Estrutura de Dados

Docente: Raimundo Santos Moura

Teresina - Piauí

Trabalho Final

Integrantes:

Kauã Marques do Nascimento

Davi do Nascimento Santos

Lucas Herlon dos Santos Moreira Cunha

Introdução

A tarefa final, tarefa essa exigida pela disciplina de Estrutura de Dados, consiste em testar um conjunto de classes presentes no Java Collections, essas classes são usadas para armazenar objetos usando diferentes estruturas de dados. A tarefa consiste em testar os diferentes tipos de classes visando a solução de algumas questões propostas, adiante encontram-se as soluções.

1 - Breve explicação das Collections do Java

Vector

Vector (Java Platform SE 8) (oracle.com)

Trata-se de uma classe que cria um array de tamanho dinâmico, ou seja, sua capacidade máxima pode ser alterada (aumentada ou reduzida) em tempo de execução. Por ser um array seus elementos localizam-se em endereços contíguos da memória e podem ser acessados por meio de índices. Sua principal diferença em relação à classe ArrayList, que também implementa a interface List, é que esta não é thread-safe enquanto aquela é.

LinkedList

LinkedList (Java SE 18 & JDK 18) (oracle.com)

Trata-se de uma classe Java que implementa as interfaces List e Deque, criando uma lista duplamente encadeada com dois descritores. A operação de busca nesta classe se dá percorrendo a lista a partir do descritor de início ou do descritor do fim, a depender de qual descritor está mais perto do índice pesquisado. Ainda assim, essa operação tem complexidade O(n), mas a operação de inserção é pouco custosa e tem complexidade O(1).

ArrayList (Java SE 18 & JDK 18) (oracle.com)

Classe que implementa a interface List, criando um array de tamanho dinâmico, o que significa que sua capacidade máxima de armazenamento pode ser alterada em tempo de execução, diferentemente dos arrays tradicionais. Um objeto ArrayList possui, dentre outros, os métodos add, get, isEmpty e size. Por ser um array as operações de busca podem ser feitas a tempo constantes por meio do índice,a tempo linear, sem uso do índice, caso o vetor não esteja ordenado ou a tempo logarítmico, caso contrário. As operações de adição e

remoção podem ser mais custosas por eventualmente envolver operações de redimensionamento.

Implementação:

As tabelas de símbolos implementadas a partir das Java Collections seguem um padrão, primeiro, criamos uma classe que é o elemento que será adicionado na tabela de símbolos, atribuímos o nome de “Entry” a essa classe, a classe Entry tem dois atributos: key e value, definimos que a chave “key” seria a palavra que seria adicionada e o value um valor inteiro que seria atribuído a essa chave. Segue abaixo a implementação:

```java
public class Entry<K, V> {
    private final K key;
    private V value;
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
}
```

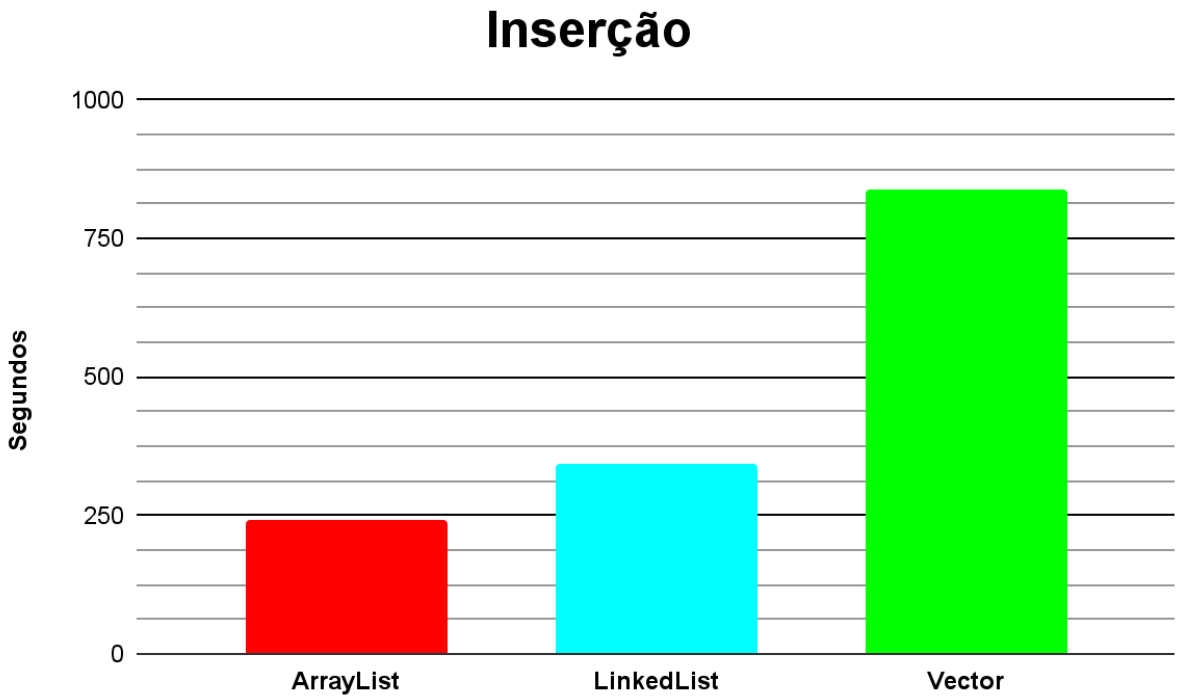
As tabelas de símbolos implementadas com as Java Collections tem o mesmo padrão, no início de cada classe é definido qual será a forma de armazenar os elementos, assim, o que diferencia uma tabela de símbolos de outra é apenas como esses dados estão sendo armazenados, fazendo com que o restante da classe sejam iguais às das outras classes. Segue abaixo uma implementação de exemplo:

```java
import java.util.ArrayList;
public class TabelaArrayList<K, V> {

    private final ArrayList<Entry<K, V>> table;

    public TabelaArrayList() {
        table = new ArrayList<Entry<K, V>>();
    }
```

O método que adiciona um elemento a tabela de símbolos recebe dois parâmetros, o primeiro é a palavra a ser adicionada, o segundo é número que está relacionado diretamente a essa palavra, como convenção definimos que não poderiam ter chaves duplicadas, e que se tentássemos adicionar uma chave já existente, o método só mudaria o valor associado a essa chave. Segue abaixo o método implementado:

```java
public void put(K key, V value) {
    // Search for the entry with the same key
    for (Entry<K, V> entry : table) {
        if (entry.getKey().equals(key)) {
            entry.setValue(value);
            return;
        }
    }
    // If the key is not found, add a new entry
    table.add(new Entry<K, V>(key, value));
}
```

Resultados obtidos nos testes:



ArrayList: 240 segundos (4 minutos)

LinkedList: 344 segundos (5 minutos e 44 segundos)

Vector: 837 segundos (13 minutos e 57 segundos)

2 - Breve explicação das Collections do Java

HashSet

[HashSet (Java SE 18 & JDK 18) (oracle.com)]
Classe que implementa a interface Set, por baixo dos panos é instanciado um HashMap, criando uma tabela hash que não aceita elementos duplicados. Se tentarmos adicionar a esse conjunto um elemento duplicado ele simplesmente será ignorado. Essa classe oferece as operações add, remove, contains e size a tempo constante, partindo do pressuposto de que a função hash espalhe os elementos de forma eficiente, ela também não garante a ordenação dos elementos nela inseridos.

LinkedHashSet

[LinkedHashSet (Java SE 18 & JDK 18) (oracle.com)]
Classe que une a implementação de uma lista duplamente encadeada com uma tabela hash. Essa collection tem como diferencial em relação a outras classes que implementam a interface Set a capacidade de manter a ordem dos elementos, tendo como critério de ordenação a ordem de entrada dos elementos. Essa ordenação é o

motivo de se utilizar uma lista ligada na implementação da tabela hash. Nessa classe as operações de add, contains e remove também possuem tempo constante.

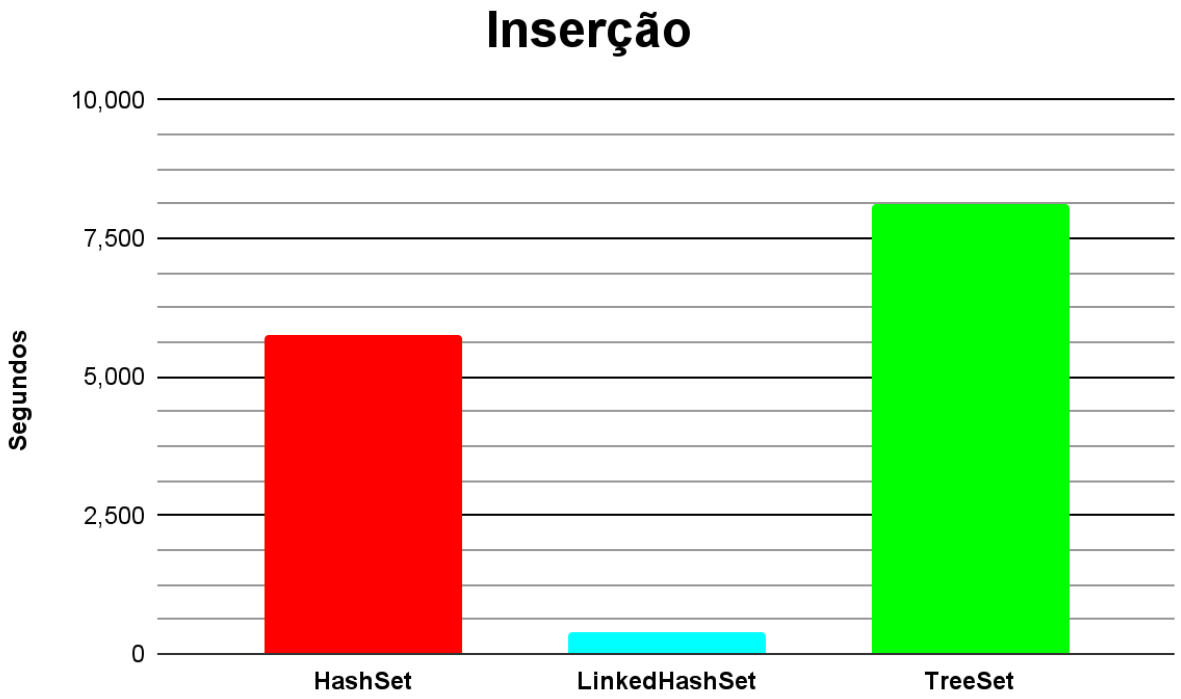
TreeSet

[TreeSet (Java SE 18 & JDK 18) (oracle.com)]

Classe que implementa a interface NavigableSet sobre a classe TreeMap. Assim como a classe anterior, os elementos são ordenados e as principais operações são feitas a tempo O(logn) (add*, remove, contains).

Implementação:

A implementação das tabelas de símbolos seguem o mesmo padrão das já citadas acima, assim sendo, abordaremos apenas os resultados obtidos nos testes.



HashSet: 5,246 segundos (1 hora e 45 minutos)

LinkedHashSet: 400 segundos (6 minutos e 40 segundos)

TreeSet: 8,137 segundos (2 horas e 26 minutos)

3 - Breve explicação das Collections do Java

HashMap

[HashMap (Java SE 18 & JDK 18) (oracle.com)]

Tabela Hash implementada a partir da interface Map. Esta classe não garante a ordenação dos elementos nela inseridos. Por se tratar de uma tabela hash, as operações get e put são realizadas a tempo constante. A performance dessa classe depende de dois de seus atributos: initial capacity e load factor. Caso a capacidade inicial seja muito alta, a performance da iteração sobre esta classe pode ser comprometida. Essas duas variáveis controlam o tamanho da tabela hash a depender da quantidade de elementos que nela são inseridos.

LinkedHashMap

LinkedHashMap (Java SE 18 & JDK 18) (oracle.com)

Implementação da interface Map, implementando uma tabela hash em que os seus elementos estão ligados entre si em forma de uma lista duplamente encadeada. O uso da lista ligada nessa estrutura permite que a tabela hash conserve uma ordem, que neste caso é a ordem de inserção. Esse ordenamento não seria garantido em uma implementação simples de tabela hash. As operações add, contains e remove são realizadas a tempo constante nessa classe, mas com performance ligeiramente inferior à da classe HashMap por conta da obrigação de manipular os links entre os elementos.

TreeMap

TreeMap (Java SE 18 & JDK 18) (oracle.com)

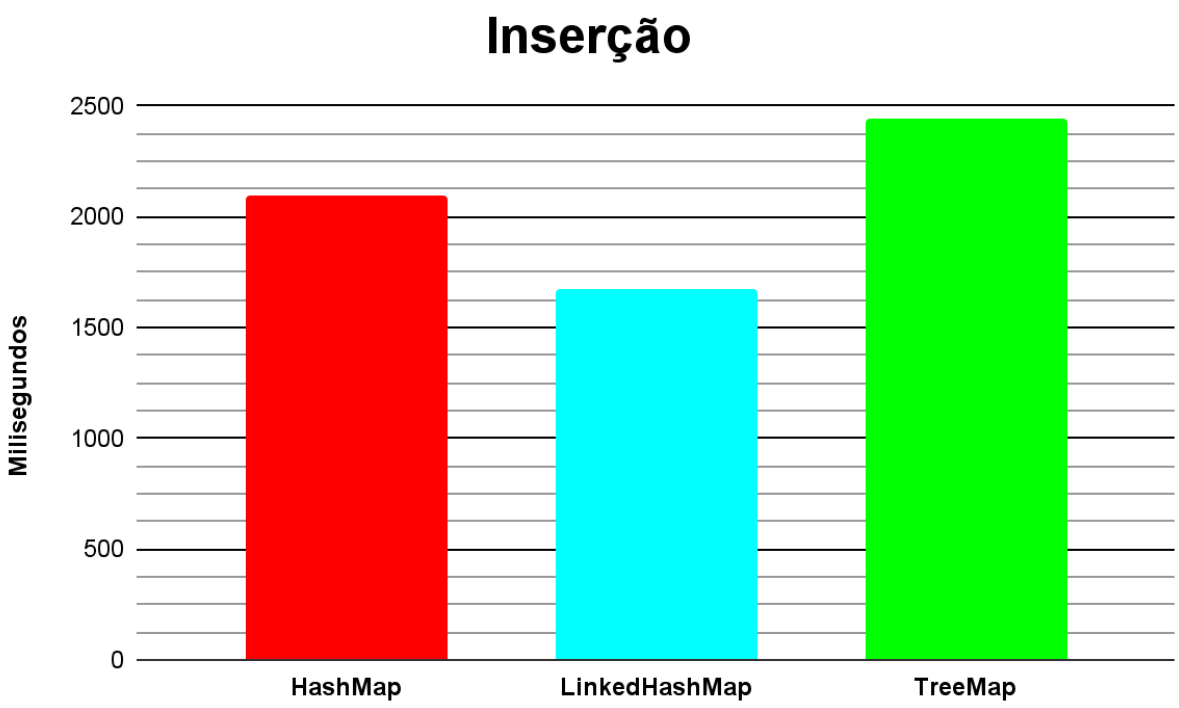
Classe que cria uma árvore rubro-negra em Java, implementando a interface NavigableMap. Portanto a TreeMap une características de uma tabela hash com uma árvore balanceada. Os elementos nessa estrutura de dados são ordenados de acordo com a ordenação natural de suas chaves, ou a ordenação fornecida pelo método compareTo() dessas mesmas chaves. Caso sejam adicionados elementos com chave repetida, o valor associado à chave será trocado. As principais operações nessa estrutura de dados são feitas a tempo O(logn), exceto pela inserção que envolve a operação de balanceamento da árvore..

Implementação:

As últimas classes implementam uma tabela de símbolos em sua estrutura, assim, passamos apenas a chave e o valor para serem adicionados na tabela de símbolos. Como a classe já implementa uma tabela de símbolos, para adicionar um novo elemento só usamos o método interno da classe. Segue o exemplo abaixo:

```java
import java.util.LinkedHashMap;
public class TabelaLinkedHashMap<K, V> {
    private final LinkedHashMap<K, V> table;

    public TabelaLinkedHashMap() {
        table = new LinkedHashMap<K, V>();
    }
    public void put(K key, V value) {
        table.put(key, value);
    }
```

Seguem abaixo os resultados obtidos no testes:



HashMap: 2100 milisegundos (2.10 segundos)

LinkedHashMap: 1671 milisegundos (1.67 segundos)

TreeMap: 2439 milisegundos (2.43 segundos)

4 - Gráficos do tempo de inserção

Os gráficos do tempo de inserção do tempo das questões anteriores já foram respondidos acima.

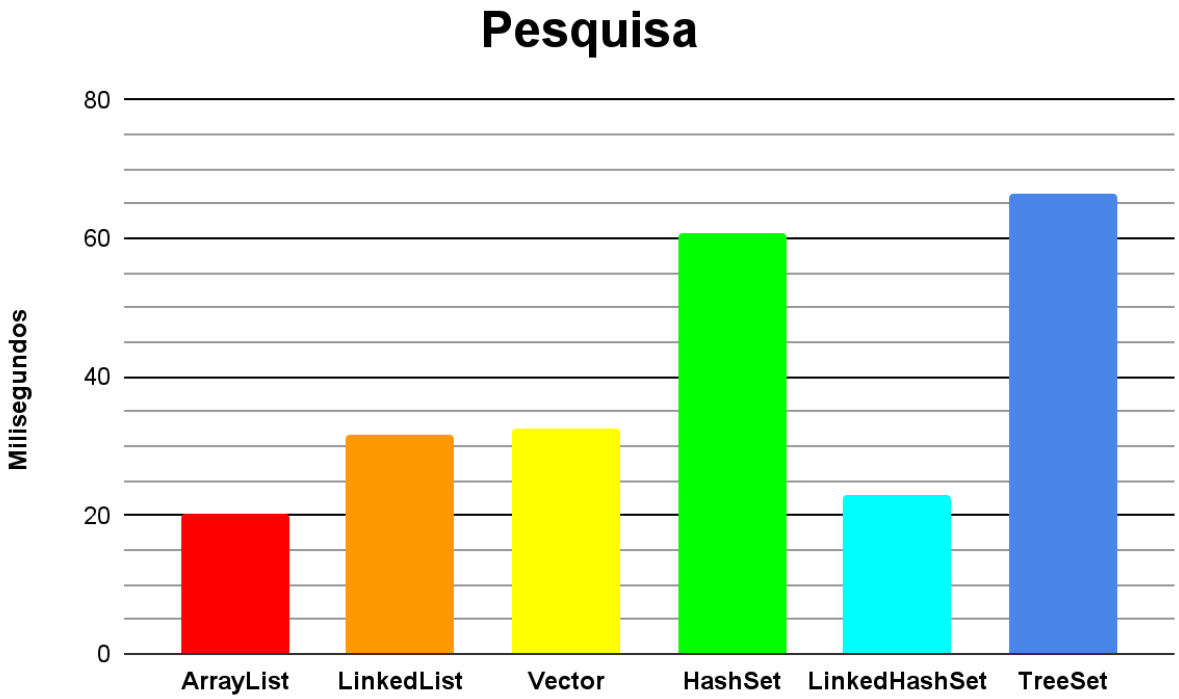
5 - Método de pesquisa

A implementação do método de pesquisa é semelhante nas seis primeiras classes, o método funciona recebendo uma chave como elemento a ser pesquisado, é implementado um laço de repetição para percorrer os elementos da tabela de símbolos, se a chave é encontrada o método retorna o valor a ela associado, se a chave não é encontrada o método retorna null. Nas últimas classes é usado o método interno da classe para pesquisar um elemento. Seguem os exemplos abaixo:

```java
public V get(K key) {
    // Search for the entry with the same key
    for (Entry<K, V> entry : table) {
        if (entry.getKey().equals(key)) {
            return entry.getValue();
        }
    }
    // If the key is not found, return null
    return null;
}
```

```java
public V get(K key) {
    // Search for the entry with the same key
    return table.get(key);
}
```

6 - Resultado das pesquisas



ArrayList: 20.2 milisegundos

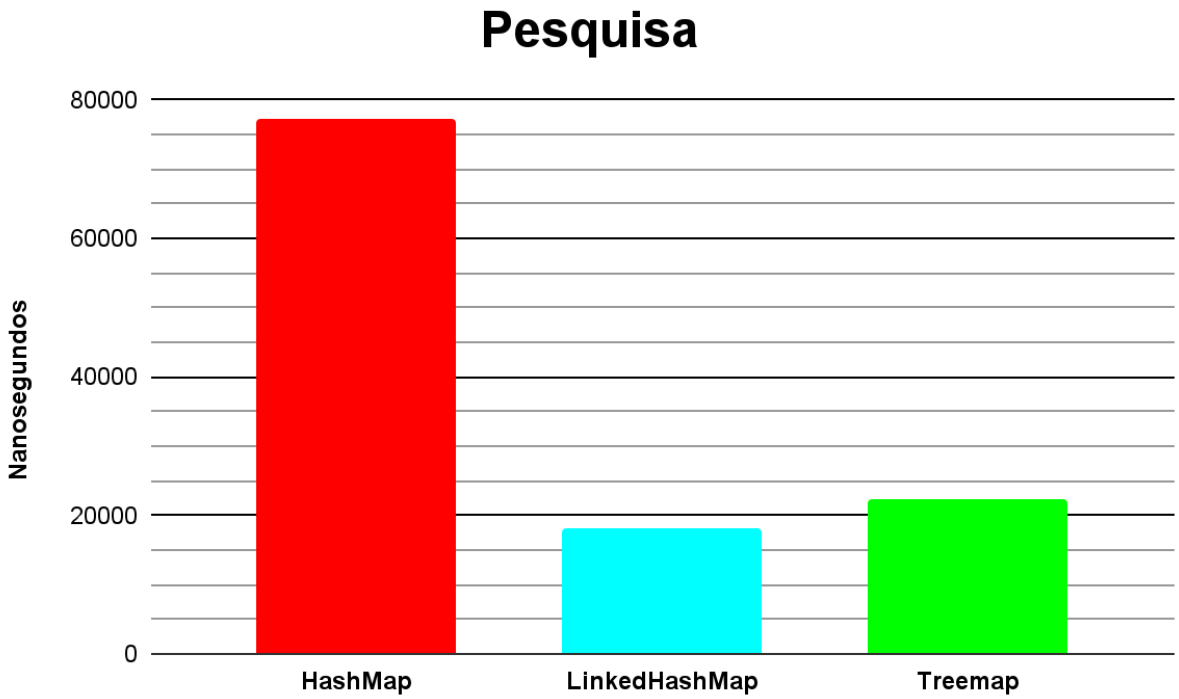
LinkedList: 31.8 milisegundos

Vector: 32.6 milisegundos

HashSet: 60.8 milisegundos

LinkedHashSet: 23 milisegundos

TreeSet: 66.4 milisegundos



HashMap: 77161 nanosegundos

LinkedHashMap: 18168 nanosegundos

TreeMap: 22337 nanosegundos

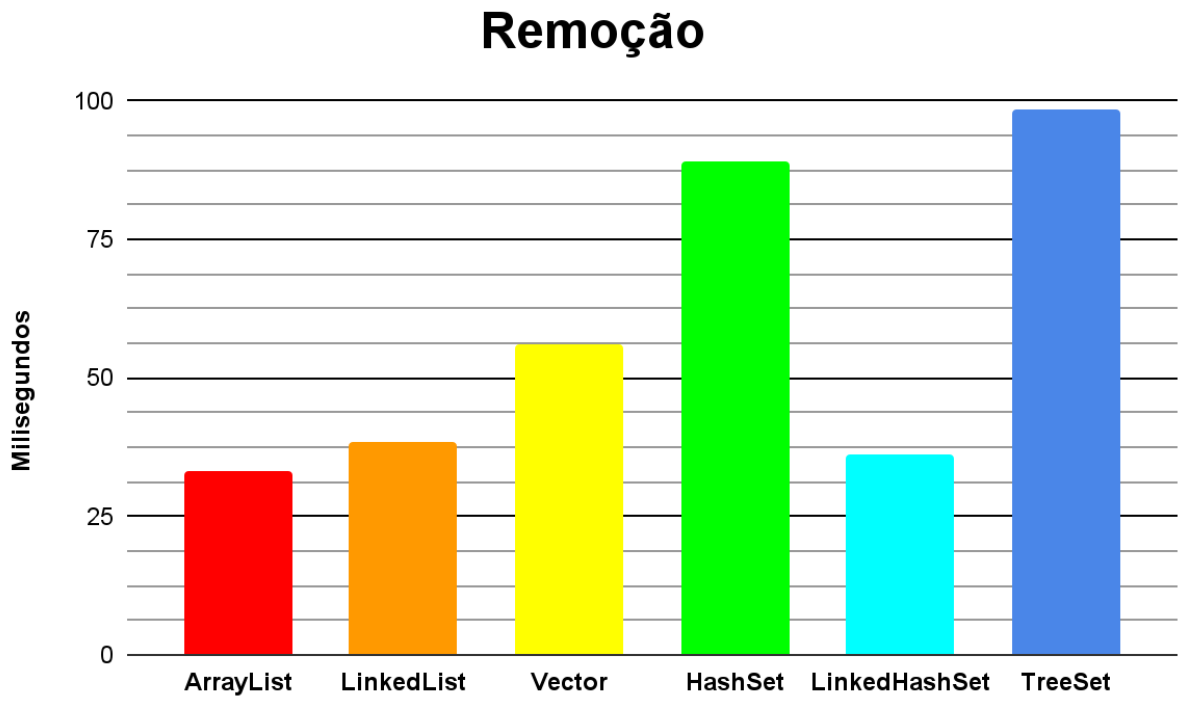
7 - Método de remoção

A implementação do método de remoção é semelhante nas seis primeiras classes, o método funciona recebendo uma chave como elemento a ser pesquisado, é implementado um laço de repetição para percorrer os elementos da tabela de símbolos, se a chave é encontrada o método retorna o valor a ela associado, se a chave não é encontrada o método retorna null. Nas últimas classes é usado o método interno da classe para pesquisar um elemento. Seguem os exemplos abaixo:

```java
public void pop(K key){
    for (Entry<K, V> entry : table) {
        if (entry.getKey().equals(key)) {
            table.remove(entry);
            break;
        }
    }
}
```

```java
public void remove(K key){
    table.remove(key);
}
```

8 - resultado das remoções



ArrayList: 33.2 milisegundos

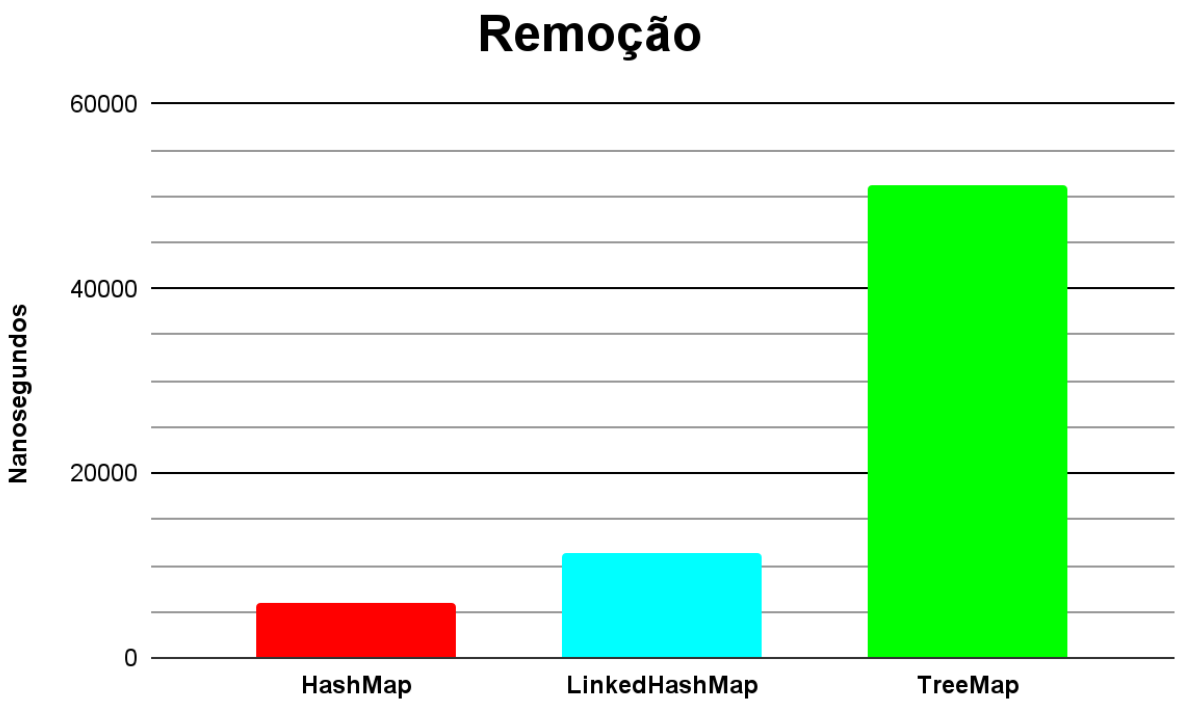
LinkedList: 38.4 milisegundos

Vector: 56.2 milisegundos

HashSet: 89 milisegundos

LinkedHashSet: 36.2 milisegundos

TreeSet: 98.6 milisegundos



HashMap: 5935 nanosegundos

LinkedHashMap: 11331 nanosegundos

TreeMap: 51112 nanosegundos