Mapas

Dicionários e tabelas

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados
Universidade do Estado de Santa Catarina

UDESC

Material de apoio

Leitura principal:

- Capítulo 10 de Goodrich et al. (2014)[1] – Mapas, tabelas hash e skip lists.

Leitura complementar:

- Capítulo 3 (3.1) de Sedgewick e Wayne (2011)[2] – Tabelas de símbolos.
- Capítulo 8 de Preiss (2001)[3] – Dispersão, tabelas de dispersão e tabelas de espalhamento.

[1] Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

[2] Robert Sedgewick e Kevin Wayne (2011). *Algorithms*. Addison-Wesley Professional.

[3] Bruno R Preiss (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.

Mapas

Conceitos básicos

Um **mapa** armazena entradas compostas por uma **chave** e um **valor**.

- A chave serve para buscar o registro.
- O valor armazena o registro associado à chave.

Mapas

Conceitos básicos

Um **mapa** armazena entradas compostas por uma **chave** e um **valor**.

- A chave serve para buscar o registro.
- O valor armazena o registro associado à chave.

Também chamado de **dicionário**, **tabela** ou **array associativo**, o mapa organiza e acessa as entradas pelas suas **chaves**, em vez das suas posições, i.e. a chave é o índice da estrutura.

Mapas

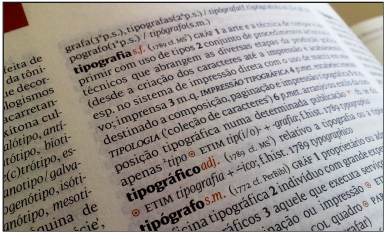
Conceitos básicos

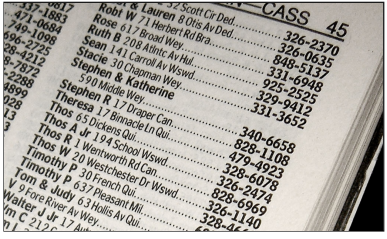
Um **mapa** armazena entradas compostas por uma **chave** e um **valor**.

- A chave serve para buscar o registro.
- O valor armazena o registro associado à chave.

Também chamado de **dicionário**, **tabela** ou **array associativo**, o mapa organiza e acessa as entradas pelas suas **chaves**, em vez das suas posições, i.e. a chave é o índice da estrutura.

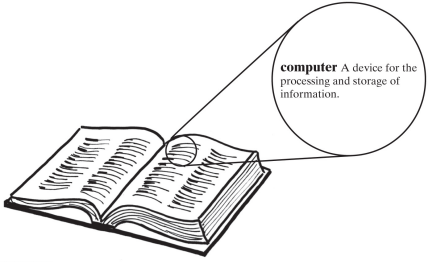
Exemplos no cotidiano: dicionários, listas telefônicas, cardápios, índices remissivos, ...





Características:

- Na maioria das implementações, **a chave é única**.
  - Ao inserir uma entrada com chave existente, o valor atual é substituído pelo novo valor.
- A chave pode ser um objeto de **qualquer classe**.
  - Necessário garantir a comparação de chaves.



Mapas

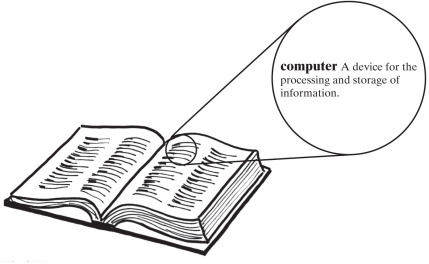
Conceitos básicos

Características:

- Na maioria das implementações, **a chave é única**.
  - Ao inserir uma entrada com chave existente, o valor atual é substituído pelo novo valor.
- A chave pode ser um objeto de **qualquer classe**.
  - Necessário garantir a comparação de chaves.

Operações:

- `get(k)`: retorna o valor associado à chave `k`.
- `put(k, v)`: insere uma entrada com chave `k` e valor v.
- `remove(k)`: remove a entrada com chave `k`.



Mapas

Análise de complexidade (diferentes implementações)

Podemos implementar um mapa usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

Mapas

Análise de complexidade (diferentes implementações)

Podemos implementar um mapa usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

- Qual a melhor escolha? ... vamos **analisar primeiro**, e escolher uma opção!

Mapas

Análise de complexidade (diferentes implementações)

Podemos implementar um mapa usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

- Qual a melhor escolha? ... vamos **analisar primeiro**, e escolher uma opção!

Operação	**Arranjo**		**Encadeamento**	
	Não ordenado	**Ordenado**	**Não ordenado**	**Ordenado**
`get`	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`put`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

— a inserção (`put`) em um arranjo ordenado é feita em $\mathcal{O}(\log n)$ na substituição.

Mapas

Análise de complexidade (diferentes implementações)

Podemos implementar um mapa usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

- Qual a melhor escolha? ... vamos **analisar primeiro**, e escolher uma opção!

Operação	**Arranjo**		**Encadeamento**	
	Não ordenado	**Ordenado**	**Não ordenado**	**Ordenado**
`get`	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`put`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$

— a inserção (`put`) em um arranjo ordenado é feita em $\mathcal{O}(\log n)$ na substituição.

Conclusão: usaremos implementações baseadas em **arranjo** (ordenado e não ordenado).

Mapas

Implementação (interface)

Interface `Map`:

```
1  public interface Map<K, V> {
2    int size();
3    boolean isEmpty();
4    V get(K key);
5    V put(K key, V value);
6    V remove(K key);
7  }
```

Mapas

Implementação (interface)

Interface `Map`:

```
1  public interface Map<K, V> {
2    int size();
3    boolean isEmpty();
4    V get(K key);
5    V put(K key, V value);
6    V remove(K key);
7  }
```

- O mapa é uma coleção de entradas, que armazenam a chave e o valor (definidos pelos tipos `K` e `V`, respectivamente).
- As operações sempre retornam o valor associado à chave sendo consultada/alterada.
- Não acessamos as entradas externamente, portanto sua definição é encapsulada no mapa.

Mapas

Implementação usando um arranjo não ordenado

Classes `UnsortedArrayMap` e `Entry`:

```
1  public class UnsortedArrayMap<K,V> implements Map<K,V> {
2    private static class Entry<K,V> {
3      private K k;
4      private V v;
5
6      public Entry(K key, V value) { k = key; v = value; }
7
8      public K getKey() { return k; }
9      public V getValue() { return v; }
10
11     public V setValue(V value) {
12       V old = v;
13       v = value;
14       return old;
15     }
16   }
17
18   private ArrayList<Entry<K,V>> data = new ArrayList<>();
19
20   //...
21 }
```

Mapas

Implementação usando um arranjo não ordenado

Classes `UnsortedArrayMap` e `Entry`:

```
1  public class UnsortedArrayMap<K,V> implements Map<K,V> {
2    private static class Entry<K,V> {
3      private K k;
4      private V v;
5
6      public Entry(K key, V value) { k = key; v = value; }
7
8      public K getKey() { return k; }
9      public V getValue() { return v; }
10
11     public V setValue(V value) {
12       V old = v;
13       v = value;
14       return old;
15     }
16   }
17
18   private ArrayList<Entry<K,V>> data = new ArrayList<>();
19
20   //...
21 }
```

- A classe `Entry` define somente um construtor parametrizando, impedindo a criação de uma entrada sem chave e valor.
- Essa classe não permite alterar a chave; para isso, uma nova entrada deve ser criada.
- Ao alterar o valor associado à entrada, o valor antigo é retornado (linhas 11 a 15).
- O mapa usa um `ArrayList` para armazenar as entradas (linha 18).

Mapas

Implementação usando um arranjo não ordenado

Método auxiliar `findKey`:

```
1  private int findIndex(K key) {
2    int n = data.size();
3    for (int j=0; j < n; j++)
4      if (data.get(j).getKey().equals(key))
5        return j;
6    return -1;
7  }
```

Mapas

Implementação usando um arranjo não ordenado

Método auxiliar `findKey`:

```
1  private int findIndex(K key) {
2    int n = data.size();
3    for (int j=0; j < n; j++)
4      if (data.get(j).getKey().equals(key))
5        return j;
6    return -1;
7  }
```

- O método implementa uma busca sequencial, retornando o índice que contém a entrada com a chave buscada (ou -1).
- **Importante:** a classe da chave deve sobrescrever o método `equals` para implementar a comparação de chaves.

Mapas

Implementação usando um arranjo não ordenado

Método auxiliar `findKey`:

```
1  private int findIndex(K key) {
2    int n = data.size();
3    for (int j=0; j < n; j++)
4      if (data.get(j).getKey().equals(key))
5        return j;
6    return -1;
7  }
```

Método `get`:

```
1  public V get(K key) {
2    int j = findIndex(key);
3    if (j == -1) return null;
4    return data.get(j).getValue();
5  }
```

- O método implementa uma busca sequencial, retornando o índice que contém a entrada com a chave buscada (ou -1).
- **Importante:** a classe da chave deve sobrescrever o método `equals` para implementar a comparação de chaves.

Mapas

Implementação usando um arranjo não ordenado

Método auxiliar `findKey`:

```
1  private int findIndex(K key) {
2    int n = data.size();
3    for (int j=0; j < n; j++)
4      if (data.get(j).getKey().equals(key))
5        return j;
6    return -1;
7  }
```

- O método implementa uma busca sequencial, retornando o índice que contém a entrada com a chave buscada (ou -1).
- **Importante:** a classe da chave deve sobrescrever o método `equals` para implementar a comparação de chaves.

Método `get`:

```
1  public V get(K key) {
2    int j = findIndex(key);
3    if (j == -1) return null;
4    return data.get(j).getValue();
5  }
```

- O método usa o `findIndex` para buscar a entrada com a chave desejada, retornando seu valor (ou `null`, caso a chave não seja encontrada).

Mapas

Implementação usando um arranjo não ordenado

Método put:

```
1  public V put(K key, V value) {
2    int j = findIndex(key);
3    if (j == -1) {
4      data.add(new Entry<>(key, value));
5      return null;
6    } else
7      return data.get(j).setValue(value);
8  }
```

Mapas

Implementação usando um arranjo não ordenado

Método `put`:

```
1  public V put(K key, V value) {
2    int j = findIndex(key);
3    if (j == -1) {
4      data.add(new Entry<>(key, value));
5      return null;
6    } else
7      return data.get(j).setValue(value);
8  }
```

- Caso a chave não tenha sido encontrada, adiciona uma nova entrada e retorna `null` (linhas 3 a 5).
- Caso contrário, substitui o valor da entrada e retorna o valor antigo (linha 7).

Mapas

Implementação usando um arranjo não ordenado

Método `put`:

```
1  public V put(K key, V value) {
2    int j = findIndex(key);
3    if (j == -1) {
4      data.add(new Entry<>(key, value));
5      return null;
6    } else
7      return data.get(j).setValue(value);
8  }
```

- Caso a chave não tenha sido encontrada, adiciona uma nova entrada e retorna `null` (linhas 3 a 5).
- Caso contrário, substitui o valor da entrada e retorna o valor antigo (linha 7).

Método `get`:

```
1  public V remove(K key) {
2    int j = findIndex(key);
3    if (j == -1) return null;
4    V answer = data.get(j).getValue();
5    int n = size();
6    if (j != n - 1) data.set(j, data.get(n-1));
7    data.remove(n-1);
8    return answer;
9  }
```

Mapas

Implementação usando um arranjo não ordenado

Método `put`:

```
1  public V put(K key, V value) {
2    int j = findIndex(key);
3    if (j == -1) {
4      data.add(new Entry<>(key, value));
5      return null;
6    } else
7      return data.get(j).setValue(value);
8  }
```

- Caso a chave não tenha sido encontrada, adiciona uma nova entrada e retorna `null` (linhas 3 a 5).
- Caso contrário, substitui o valor da entrada e retorna o valor antigo (linha 7).

Método `get`:

```
1  public V remove(K key) {
2    int j = findIndex(key);
3    if (j == -1) return null;
4    V answer = data.get(j).getValue();
5    int n = size();
6    if (j != n - 1) data.set(j, data.get(n-1));
7    data.remove(n-1);
8    return answer;
9  }
```

- O método troca de posição a entrada a ser removida e a última entrada (se não forem a mesma), e então remove a atual entrada da última posição, cuja operação é mais eficiente. Ao final, retorna o valor da entrada removida.

Mapas

Implementação usando um arranjo ordenado

Classe `SortedArrayMap`:

```
1 public class SortedArrayMap<K extends Comparable<? super K>,V> implements Map<K,V> {
2   // Definição da classe Entry
3
4   private ArrayList<Entry<K,V>> data = new ArrayList<>();
5   //...
6 }
```

Para **comparar as chaves**:

1. usamos uma chave naturalmente comparável (e.g. `Integer`, `String`); ou
2. implementamos a comparação na classe da chave.
   - i.e., interface `Comparable` e método `compareTo`.

Na definição do tipo genérico `K`, a classe já exige que a chave seja `Comparable` (linha 1).

Mapas

Implementação usando um arranjo ordenado

O método `findIndex` agora implementa uma busca binária:

```
1  private int findIndex(K key) { return findIndex(key, 0, data.size() - 1); }
2
3  private int findIndex(K key, int low, int high) {
4    if (high < low) return high + 1;
5    int mid = (low + high) / 2;
6    int result = key.compareTo(data.get(mid).getKey());
7    if (result == 0)
8      return mid;
9    else if (result < 0)
10     return findIndex(key, low, mid - 1);
11   else
12     return findIndex(key, mid + 1, high);
13 }
```

- Trata-se de uma implementação recursiva do algoritmo.
- Quando o método não encontra a chave, retorna a posição de inserção.

Mapas

Implementação usando um arranjo ordenado

Métodos `get` e `remove`:

```
1  public V get(K key) {
2    int j = findIndex(key);
3    if (j == size() || key.compareTo(data.get(j).getKey()) != 0) return null;
4    return data.get(j).getValue();
5  }
6
7  public V remove(K key) {
8    int j = findIndex(key);
9    if (j == size() || key.compareTo(data.get(j).getKey()) != 0) return null;
10   return data.remove(j).getValue();
11 }
```

Mapas

Implementação usando um arranjo ordenado

Métodos `get` e `remove`:

```
1  public V get(K key) {
2    int j = findIndex(key);
3    if (j == size() || key.compareTo(data.get(j).getKey()) != 0) return null;
4    return data.get(j).getValue();
5  }
6
7  public V remove(K key) {
8    int j = findIndex(key);
9    if (j == size() || key.compareTo(data.get(j).getKey()) != 0) return null;
10   return data.remove(j).getValue();
11 }
```

- Ambos os métodos têm o mesmo comportamento geral, usando a busca binária de chave.
- O condicional verifica o insucesso da busca, quando o índice (de inserção) é inválido ou a chave não se encontra naquela posição (de inserção), retornando `null` (linhas 3 e 9).
- Em caso de sucesso, retorna o valor (lin. 4) ou remove a entrada e retorna seu valor (lin. 10).

Mapas

Implementação usando um arranjo ordenado

Método put:

```
1  public V put(K key, V value) {
2    int j = findIndex(key);
3    if (j < size() && key.compareTo(data.get(j).getKey()) == 0)
4      return data.get(j).setValue(value);
5    data.add(j, new Entry<K,V>(key,value));
6    return null;
7  }
```

Mapas

Implementação usando um arranjo ordenado

Método `put`:

```
1 public V put(K key, V value) {
2   int j = findIndex(key);
3   if (j < size() && key.compareTo(data.get(j).getKey()) == 0)
4     return data.get(j).setValue(value);
5   data.add(j, new Entry<K,V>(key,value));
6   return null;
7 }
```

- A busca binária (`findIndex`) já retorna a posição de inserção.
- Caso essa posição já contenha uma entrada com a mesma chave a ser inserida, só substitui o valor da entrada e retorna o valor antigo (linhas 3 e 4).
- Caso contrário, adiciona uma nova entrada e retorna `null` (linhas 5 e 6).

45EST – Algoritmos e Estruturas de Dados
Prof. Marcelo de Souza