Universidade Presbiteriana Mackenzie

ESTRUTURA DE DADOS II

Tabela Hash
Atividade (máx. três alunos)

Objetivo

Implementar duas tabelas hash com diferentes tratamentos de colisão (endereçamento aberto e encadeamento) e com operações de busca, inserção e remoção, em Java.

Instruções

- A atividade deve ser resolvida usando a linguagem Java.
- A solução não deve usar as estruturas de dados relacionadas à tabela hash oferecidas pela linguagem Java (projetos que usarem tais estruturas serão desconsiderados – zero).
- Inclua a identificação do grupo (nomes completo e TIA de cada integrante) no início de cada arquivo de código, como comentário.
- Inclua todas as referências (livros, artigos, sites, vídeos, entre outros) consultadas para solucionar a atividade, como comentário no arquivo `.java` que contém a `main()`.

Enunciado

1. Crie uma interface para as operações comuns da tabela hash (veja o **Apêndice: Um pouco de teoria...** deste documento). A interface deve estar declarada em seu próprio arquivo java (ex. `HashTable.java`).

- A chave deve ser do tipo `int` e o valor do tipo `String`.
- A busca deve ser feita pela chave e deve retornar o valor associado à chave, se existir.
- A inserção deve retornar um de três valores, indicando "chave-valor inserido", "valor da chave atualizado" ou "erro ao inserir chave-valor".
- A remoção deve ser feita pela chave e deve retornar `true` se foi possível remover a chave-valor ou `false` caso contrário.

2. Crie uma classe que implementa a tabela hash com tratamento de colisão por endereçamento aberto (*open addressing*) com sondagem/tentativa linear. Essa classe deve implementar a interface do item 1 e deve estar em seu próprio arquivo java (ex. `HashTableOA.java`). Inclua um construtor que recebe como parâmetro o tamanho da tabela hash. A escolha da função hash fica a seu critério.

3. Crie uma classe que implementa a tabela hash com tratamento de colisão por encadeamento (*chaining*). Essa classe deve implementar a interface do item 1 e deve estar em seu próprio arquivo java (ex. `HashTableChaining.java`). Inclua um construtor que recebe como parâmetro o tamanho da tabela hash. Use uma função hash diferente da tabela hash do item 2.

4. Para testar a sua implementação das tabelas hash, defina um conjunto de chaves e valores para inserir, consultar e remover em cada tabela hash. Escreva casos de teste e faça a validação dos testes na `main()` (ex. O que acontece quando a pessoa insere uma chave que já existe na tabela hash? O que acontece quando a tabela fica cheia? O que acontece quando a pessoa tenta remover uma chave que não existe?).

5. Realize todas as alterações e consultas primeiro na tabela hash com endereçamento aberto e depois na tabela hash com encadeamento, sempre usando uma mesma variável do tipo da interface criada no item 1.

Universidade Presbiteriana Mackenzie

ESTRUTURA DE DADOS II

Entrega

Compacte o código-fonte (somente arquivos *.java) no **formato zip**.

Atenção: O arquivo zip não deve conter arquivos intermediários e/ou pastas geradas pelo compilador/IDE (ex. arquivos *.class, etc.).

Prazo de entrega: via link do Moodle até 10/11/2023 23:59.

Critérios de avaliação

A nota da atividade é calculada de acordo com os critérios da tabela a seguir.

ITEM AVALIADO	PONTUAÇÃO MÁXIMA
1. Interface para as operações comuns da tabela hash.	1,0
2. Tabela hash com tratamento de colisão por endereçamento aberto (*open addressing*).	3,5
3. Tabela hash com tratamento de colisão por encadeamento (*chaining*).	3,5
4. Testes das alterações e consultas de cada tabela hash.	1,0
5. Funcionamento geral do programa, de acordo com o enunciado.	1,0

Tabela 1 - Critérios de avaliação.

A tabela a seguir contém critérios de avaliação que podem **reduzir** a nota final da atividade.

ITEM INDESEJÁVEL	REDUÇÃO DE NOTA
O projeto é cópia de outro projeto.	Projeto é zerado
O projeto usa estruturas de dados relacionadas à tabela hash oferecidas pela linguagem Java.	Projeto é zerado
Há erros de compilação e/ou o programa trava durante a execução[1].	-1,0
Não há identificação do grupo. Não há indicação de referências. Arquivos enviados em formatos incorretos. Arquivos e/ou pastas intermediárias que são criadas no processo de compilação ou pela IDE foram enviadas junto com o código-fonte.	-1,0

Tabela 2 - Critérios de avaliação (redução de nota).

O código-fonte será compilado com o compilador javac (17.0.8) na plataforma Windows da seguinte forma:

```
> javac *.java
```

O código compilado será executado com java (17.0.8) na plataforma Windows da seguinte forma (`<Classe>` deve ser substituído pelo nome da classe que contém o método `public static void main(String[] args)`):

```
> java <Classe>
```

[1] Sobre erros de compilação: considere apenas erros. Não há problema se o projeto tiver *warnings* (embora *warnings* podem avisar sobre possíveis travamentos em tempo de execução, como loop infinito, divisão por zero, etc.).

ESTRUTURA DE DADOS II

APÊNDICE: Um pouco de teoria...

Conforme apresentado em aula, as operações comuns de uma tabela hash envolvem:

- Busca por chaves;
- Inserção de chaves;
- Remoção de chaves.

O funcionamento e implementação de cada operação listada acima depende da técnica de tratamento de colisão escolhida para a tabela hash, sendo que cada técnica usa diferentes tipos de dados na implementação:

- Endereçamento aberto (*open addressing*) usa um array de um tipo que contém chave e valor (*key-value*);
- Encadeamento (*chaining*) usa um array de lista encadeada, sendo que cada nó da lista contém a chave e o valor.

Pensando no conceito de herança de classes (superclasse/subclasse, classe pai|mãe/classe filho(a), classe base/classe derivada, generalização/especialização), podemos observar que, para implementar uma tabela hash com tratamento de colisão por endereçamento e outra por encadeamento, não é viável criar uma classe base e uma classe derivada: qual técnica seria a classe base e qual seria a derivada?

Embora seja possível ter duas classes que implementam tabelas hash, cada uma com tratamento de colisão diferente, não faz sentido usar herança de classes como fizemos nas atividades anteriores: se a classe base for do tipo endereçamento aberto e a derivada for do tipo encadeamento, a classe derivada nunca usará o array da classe base. O mesmo problema acontece se a classe base for do tipo encadeamento e a derivada for endereçamento aberto.

Porém, sabemos que, independentemente do tratamento de colisão, a tabela hash tem operações de busca, inserção e remoção. E, pensando na pessoa usuária das nossas tabelas hash, é importante que ela saiba que as duas implementações oferecerão as mesmas operações (pelo menos em termos de assinatura do método) e que ela tem como escolher entre uma tabela hash com endereçamento aberto e uma tabela hash com encadeamento.

Seria interessante que a pessoa tivesse a possibilidade de escrever um código do tipo:

```
// Cria uma instância de uma tabela hash com endereçamento aberto.
HashTable ht1 = new HashTableOA();

// Cria uma instância de uma tabela hash com encadeamento.
HashTable ht2 = new HashTableChaining();
```

Ué? Não seria o caso de simplesmente criar uma classe `HashTable` com três métodos vazios (ex. `search()`, `insert()`, `remove()`) e, então, `HashTableOA` e `HashTableChaining` herdariam de `HashTable`, sobrescreveriam os métodos vazios e definiriam os atributos necessários para cada tratamento de colisão?

Sim! Mas... Quem garante que `HashTableOA` e `HashTableChaining` realmente sobrescrevem os métodos da superclasse `HashTable`? Da forma que trabalhamos com herança de classes, não é possível ter essa garantia!

Para este cenário, quem consegue garantir que `HashTableOA` e `HashTableChaining` sobrescrevem os métodos `search()`, `insert()` e `remove()`, implementando as instruções específicas para cada versão da tabela hash é uma...

ESTRUTURA DE DADOS II

Interface

Uma `interface` pode ser vista como um contrato que obriga uma classe a implementar os métodos da interface. Ou seja: se uma classe "herda" uma interface, ela deve implementar todos os métodos que foram descritos na interface – mesmo que o método não tenha nenhuma instrução, ele deve existir na classe.

A palavra "herdar" está entre aspas porque não dizemos que ~~*"uma classe é uma interface"*~~ (herança, *"is-a"*). Afirmar que ~~*"uma classe contém uma interface"*~~ (composição, *"has-a"*) também é errado. O correto é dizer que *uma classe implementa uma interface*.

OK, mas qual é a diferença entre uma classe e uma interface?

- No caso da interface, não há implementação de métodos, apenas assinaturas de métodos[2], sendo que todos os métodos são implicitamente públicos (não há necessidade de usar a palavra-chave `public`).
- Uma interface também não possui construtores – não é possível instanciar uma interface! O que pode ser instanciado é uma classe que *implementa* a interface.
- Uma classe pode implementar uma ou mais interfaces.

Com interfaces, podemos definir um conjunto de métodos que devem ser implementados por várias classes diferentes, independentemente das hierarquias de herança. Também é possível fazer polimorfismo com interfaces (referenciamos os objetos por meio das interfaces). Assim, o código pode se tornar mais flexível e modular.

Como criar uma interface

```
// arquivo: NomeDaInterface.java

public interface NomeDaInterface {
  // Métodos da interface aqui.
}
```

Dois exemplos de interface:

```
// arquivo: KeyListener.java

public interface KeyListener {
  void onKeyUp(int keyCode);
  void onKeyDown(int keyCode);
  void onKeyPressed(int keycode);
}
```

```
// arquivo: MouseListener.java

public interface MouseListener {
  void onClick(int button, int x, int y);
  void onDoubleClick(int button, int x, int y);
}
```

[2] Também podemos incluir atributos e métodos padrão, mas não falaremos sobre isso neste material. Para saber mais, esse artigo é uma boa referência: https://www.freecodecamp.org/portuguese/news/interfaces-em-java-explicadas-com-exemplos.

Como uma classe deve implementar uma interface

```
// arquivo: NomeDaClasse.java

class NomeDaClasse implements NomeDaInterface {
  // Métodos da interface implementados aqui.
}
```

Exemplo de classe implementando a interface `KeyListener` da seção anterior:

```
// arquivo: TextBox.java

public class TextBox implements KeyListener {

  private String name;

  public TextBox(String name) {
    this.name = name;
  }

  @Override
  public void onKeyUp(int keyCode) {
    System.out.println(name + " -> tecla " + keyCode + " foi liberada!");
  }

  @Override
  public void onKeyDown(int keyCode) {
    System.out.println(name + " -> tecla " + keyCode + " foi pressionada!");
  }

  @Override
  public void onKeyPressed(int keyCode) {
    System.out.println(name + " -> tecla " + keyCode + " está pressionada!");
  }

}
```

Exemplo de classe implementando as interfaces `KeyListener` e `MouseListener` da seção anterior:

```
// arquivo: Window.java

public class Window implements KeyListener, MouseListener {

  @Override
  public void onClick(int button, int x, int y) {
    if (button == 0)
      System.out.println("Window: Clicou na janela - LMB(" + x + ", " + y + ")");
  }

  @Override
  public void onDoubleClick(int button, int x, int y) {
    // Do nothing.
  }

  @Override
```

```
  public void onKeyUp(int keyCode) {
    if (keyCode == VK_ESCAPE)
      System.exit(0);
  }

  @Override
  public void onKeyDown(int keyCode) {
    System.out.println("Window: Ignorando tecla " + keyCode + "...");
  }

  @Override
  public void onKeyPressed(int keyCode) {
    // Do nothing.
  }

}
```

Exemplo de polimorfismo com interface

```
// arquivo: MainGUI.java

public class MainGUI {

  KeyListener keyListeners[];
  MouseListener mouseListeners[];

  public MainGUI() {
    Window win = new Window();

    // Polimorfismo!
    keyListeners = new KeyListener[3];
    keyListeners[0] = win;
    keyListeners[1] = new TextBox("login");
    keyListeners[2] = new TextBox("senha");

    mouseListeners = new MouseListener[2];
    mouseListeners[0] = win;
  }

  // Observe que tanto faz o tipo do objeto (Window ou TextBox, no exemplo);
  // o polimorfismo é feito com a interface. Como keyListeners é um array de
  // KeyListener, conseguimos chamar o método onKeyDown() de todas as classes
  // que implementam a interface KeyListener (idem para MouseListener a seguir).
  public void dispatchOnKeyDownEvent(int keyCode) {
    for (var obj : keyListeners)
      if (obj != null)
        obj.onKeyDown(keyCode);
  }

  public void dispatchOnClickEvent(int button, int x, int y) {
    for (var obj : mouseListeners)
      if (obj != null)
        obj.onClick(button, x, y);
  }

}
```

```
// arquivo: Main.java

public class Main {

  public static void main(String[] args) {
    // Apenas uma simulação de notificação de eventos de teclado e mouse...
    MainGUI gui = new MainGUI();
    gui.dispatchOnKeyDownEvent(65);
    gui.dispatchOnClickEvent(0, 100, 200);
  }

}
```

Saída desse exemplo:

```
Window: Ignorando tecla 65...
login -> tecla 65 foi pressionada!
senha -> tecla 65 foi pressionada!
Window: Clicou na janela - LMB(100, 200)
```