Universidade São Judas Tadeu Butantã

Gestão e qualidade de software - CCP1AN-BUE1-6507138

Guilherme de Camargo Leite Eubank Pereira - 822142574 - 822142574@ulife.com.br
Guilherme Farias Menoci - 822135941 - 822135941@ulife.com.br
João Henrique Bezerra dos Santos RA: 821141558 - 821141558@ulife.com.br

São Paulo 2024

Plano de Testes

Entendendo a Função

A função busca_binaria implementa o algoritmo de busca binária, um método eficiente para encontrar um elemento em um array ordenado. A função recebe um array de inteiros (iVet) e um valor a ser buscado (iK). Ela retorna o índice do elemento encontrado, caso exista, ou -1 caso contrário.

Objetivos dos Testes

- **Corretude:** Verificar se a função retorna o índice correto para elementos presentes e ausentes no array.
- **Robustez:** Testar a função com diferentes tamanhos de arrays, valores extremos (primeiro, último, meio), e valores fora do intervalo.
- **Eficiência:** Avaliar o desempenho da função em diferentes cenários, como arrays grandes e pequenos.
- **Limitações:** Identificar as condições nas quais a função não funciona corretamente (e.g., array não ordenado).

Casos de Teste

1. Casos Básicos

- **Elemento no meio:** Array com número ímpar de elementos, elemento buscado no meio.
- **Elemento no início:** Elemento buscado é o primeiro do array.
- **Elemento no final:** Elemento buscado é o último do array.
- **Elemento não encontrado:** Elemento buscado não está no array.

2. Casos Limite

- **Array vazio:** Verificar o comportamento quando o array de entrada está vazio.
- **Array com um elemento:** Verificar o comportamento quando o array tem apenas um elemento.
- **Elemento menor que todos:** Elemento buscado é menor que todos os elementos do array.
- **Elemento maior que todos:** Elemento buscado é maior que todos os elementos do array.

3. Casos Especiais

- **Elementos duplicados:** Array com elementos duplicados.
- **Array ordenado de forma decrescente:** Verificar o comportamento quando o array não está ordenado de forma crescente.
- **Overflow de índice:** Testar com valores muito grandes para os índices, para verificar se a função lida com esses casos.

Estrutura dos Testes (Exemplo em JUnit)

Java

```
import org.junit.Test;
 import static org.junit.Assert.*;

 public class BuscaBinariaTest {

     @Test
     public void testElementoNoMeio() {
         int[] array = {1, 2, 3, 4, 5};
         int resultado = busca_binaria(array, 3);
         assertEquals(2, resultado);
     }

     // ... outros testes ...
 }
```

Métricas de Qualidade

- **Cobertura de código:** Verificar se todos os caminhos do código foram executados nos testes.
- **Complexidade ciclomática:** Avaliar a complexidade dos testes para garantir que eles são concisos e eficientes.
- **Tempo de execução:** Medir o tempo de execução dos testes para identificar gargalos de desempenho.

Considerações Adicionais

- **Testes unitários:** Cada teste deve verificar uma única funcionalidade da função.
- **Dados de teste:** Utilizar diferentes conjuntos de dados para garantir a robustez dos testes.
- **Ferramentas de teste:** Utilizar frameworks de testes como JUnit para automatizar a execução dos testes.
- **Testes de desempenho:** Utilizar ferramentas de profiling para medir o desempenho da função em diferentes cenários.

Roteiro de Testes

Objetivo: Verificar a corretude, robustez e eficiência da função de busca binária em diferentes cenários.

1. Divisão dos Testes

- **Testes de unidade:** Verificam o comportamento individual da função para diferentes entradas.
- **Testes de integração:** Avaliam a interação da função com outras partes do sistema (se aplicável).
- **Testes de desempenho:** Medem o tempo de execução da função em diferentes condições.

2. Casos de Teste Específicos

2.1 Testes de Unidade

- **Casos básicos:**
  - Elemento no meio do array.
  - Elemento no início do array.
  - Elemento no final do array.
  - Elemento não encontrado no array.
- **Casos limite:**
  - Array vazio.
  - Array com um elemento.
  - Elemento menor que todos os elementos do array.
  - Elemento maior que todos os elementos do array.
- **Casos especiais:**
  - Array com elementos duplicados (em várias posições).
  - Array ordenado de forma decrescente.
  - Array com números negativos.
  - Array com números muito grandes ou muito pequenos.
  - Índices de busca inválidos (negativos, maiores que o tamanho do array).
- **Casos de erro:**
  - Array nulo.
  - Array não ordenado.

2.2 Testes de Integração (se aplicável)

- **Integração com outras funções:** Se a função de busca binária é utilizada em outras partes do sistema, verificar se a integração funciona corretamente.
- **Integração com estruturas de dados:** Se a função utiliza outras estruturas de dados (e.g., listas ligadas), verificar se a interação é correta.

2.3 Testes de Desempenho

- **Arrays de diferentes tamanhos:** Testar com arrays pequenos, médios e grandes.

- **Elementos aleatórios:** Gerar arrays com elementos aleatórios para verificar o desempenho em diferentes distribuições de dados.
- **Elementos ordenados:** Testar com arrays já ordenados para verificar o melhor caso de desempenho.
- **Elementos inversamente ordenados:** Testar com arrays ordenados de forma decrescente para verificar o pior caso de desempenho.

3. Estrutura dos Testes (Exemplo em JUnit)

Java

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class BuscaBinariaTest {

    @Test
    public void testElementoNoMeio() {
        int[] array = {1, 2, 3, 4, 5};
        int resultado = busca_binaria(array, 3);
        assertEquals(2, resultado);
    }

    @Test
    public void testArrayVazio() {
        int[] array = {};
        int resultado = busca_binaria(array, 5);
        assertEquals(-1, resultado);
    }

    // ... outros testes ...
}
```

4. Métricas de Qualidade

- **Cobertura de código:** Utilizar ferramentas como JaCoCo para medir a porcentagem do código que foi executada nos testes.
- **Complexidade ciclomática:** Avaliar a complexidade dos testes para garantir que eles são concisos e eficientes.
- **Tempo de execução:** Medir o tempo de execução dos testes para identificar gargalos de desempenho.

5. Ferramentas

- **JUnit:** Framework de testes unitários para Java.
- **Mockito:** Framework de testes de mock para simular objetos e dependências.
- **JaCoCo:** Ferramenta de cobertura de código.
- **Profilers:** Ferramentas para medir o desempenho da aplicação (e.g., VisualVM).

Considerações Adicionais

- **Testes parametrizados:** Utilizar testes parametrizados para reduzir a redundância de código e aumentar a cobertura de testes.
- **Testes de exceções:** Verificar se a função lança as exceções esperadas para entradas inválidas.
- **Testes de borda:** Testar os limites da função (e.g., valores mínimos e máximos para os índices).