Харківський національний університет імені В. Н. Каразіна

Факультет комп’ютерних наук

Кафедра теоретичної та прикладної системотехніки

ЕКЗАМЕНАЦІЙНА РОБОТА

дисципліна: «Мови Прикладного Програмування»

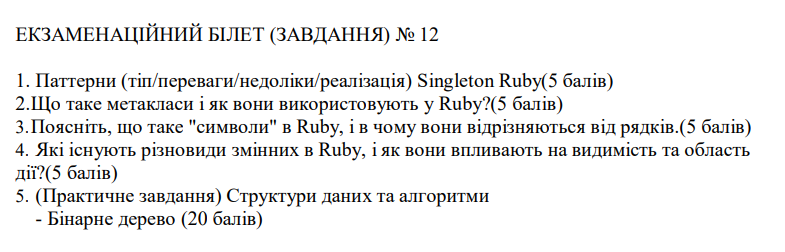
Виконала студентка групи КС31

Второва Яна Вячеславівна

Перевірив

Паршенцев Б.В.

Харків - 2023



**№1. Паттерни (тіп/переваги/недоліки/реалізація) Singleton Ruby**

* *Тип:*

Паттерн Singleton - це породжений породжуючий паттерн, який гарантує, що клас має лише один екземпляр і забезпечує глобальну точку доступу до цього екземпляра.

* *Переваги:*

Гарантія єдиного екземпляра: Singleton забезпечує, що у вас буде тільки один екземпляр класу в системі.

Глобальний доступ: Легкий доступ до єдиного екземпляра з будь-якої точки програми.

* *Недоліки Singleton в Ruby:*

Схильність до перевантаження: Іноді використання Singleton може призвести до поганого дизайну, оскільки клас стає глобальною змінною.

Труднощі в тестуванні : Тестування може бути ускладненим, оскільки Singleton може зберігати стан між тестами.

* *Реалізація Singleton в Ruby:*

В Ruby можна використовувати простий підхід з використанням модуля і змінної класу.

Приклад:

class SingletonClass

attr\_accessor :data

def self.instance

@instance ||= new

end

private\_class\_method :new

end

singleton\_instance = SingletonClass.instance

singleton\_instance.data = "Hello, Singleton!"

puts singleton\_instance.data

**№2. Що таке метакласи і як вони використовують у Ruby?**

Метакласи в Ruby:

Метакласи - це концепція в мові програмування Ruby, яка використовується для управління класами та їх поведінкою. Кожен об'єкт в Ruby є екземпляром класу, і кожен клас також є об'єктом. Однак для управління класами і їх поведінкою Ruby використовує спеціальний механізм, який називається метакласами.

Основні концепції метакласів в Ruby:

1. Одиничний екземпляр метакласу:

Для кожного класу у Ruby існує тільки один об'єкт метакласу. Цей метаклас використовується для зберігання методів, які специфічні лише для цього класу.

1. Класи - об'єкти:

Кожен клас в Ruby є об'єктом класу Class. Отже, клас може мати свій власний метаклас.

1. Робота з методами:

Метакласи використовуються для зберігання методів, які визначаються тільки для конкретного об'єкта класу. Тобто, методи, які не є спільними для всіх об'єктів класу, можуть бути розміщені в метакласі.

Застосування метакласів в Ruby:

* Модифікація класів під час виконання: Метакласи дозволяють додавати, змінювати або видаляти методи класу під час виконання програми.
* Реалізація Singleton-методів: Singleton-методи - це методи, які визначаються тільки для конкретного об'єкта, і їх реалізація здійснюється за допомогою метакласів.
* Динамічна зміна поведінки об'єктів: Метакласи дозволяють динамічно змінювати поведінку об'єктів класу під час виконання програми.

**№3. Поясніть, що таке "символи" в Ruby, і в чому вони відрізняються від рядків.**

Символи в Ruby:

У Ruby символ - це літерал, що виглядає як ім'я змінної, але починається з двокрапки. Наприклад, ‘: symbol\_name’. Символи є незмінними, тобто один і той же символ завжди вказує на один і той же об'єкт пам'яті протягом всього життя програми.

Основні відмінності між символами і рядками:

1. Незмінність:

Символи незмінні, їхні значення не можна змінити. Рядки можуть бути змінені.

1. Пам'ять:

Символи використовують менше пам'яті, оскільки вказують на той самий об'єкт пам'яті. Рядки створюють новий об'єкт пам'яті кожного разу, коли вони оголошуються.

1. Ідентифікація:

Символи використовуються для представлення імен або лейблів. Рядки використовуються для роботи з текстовим змістом.

1. Оголошення:

Оголошення символу виглядає як :symbol\_name, де symbol\_name - це ім'я символу. Оголошення рядка виглядає як "string". Символи зручніші, коли потрібно просто представити ім'я чи лейбл, а не використовувати його для маніпуляцій зі змістом.

1. Використання в хешах:

Символи часто використовуються як ключі в хешах, оскільки швидко представляють лейбли чи імена.

**№4. Які існують різновиди змінних в Ruby, і як вони впливають на видимість та область дії?**

У Ruby існують два основних типи змінних: локальні та інстанційні. Крім того, є ще змінні класу та глобальні змінні. Кожен тип змінної має власний обсяг видимості і область дії.

1. Локальні змінні:

* Видимість: Локальні змінні обмежені областю свого визначення і не доступні за її межами.
* Область дії: Локальні змінні визначаються в межах методу, блока або іншої конструкції та видаляються при завершенні цієї конструкції.

1. Інстанційні змінні:

* Видимість: Інстанційні змінні належать об'єкту та можуть бути доступні в усіх методах цього об'єкта.
* Область дії: Інстанційні змінні зберігаються протягом усього життя об'єкта і доступні всім методам об'єкта.

1. Змінні класу:

* Видимість: Змінні класу належать класу та можуть бути доступні всім об'єктам цього класу та його підкласів.
* Область дії: Змінні класу зберігають значення між різними об'єктами класу, але вони належать класу, а не конкретному об'єкту.

1. Глобальні змінні:

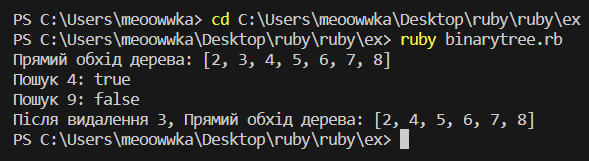
* Видимість: Глобальні змінні доступні в усьому коді програми.
* Область дії: Глобальні змінні зберігають значення протягом усього виконання програми.

1. Константи:

* Видимість: Константи подібні до глобальних змінних, але вони не повинні змінюватись.
* Область дії: Константи доступні в усьому коді програми і зазвичай використовуються для представлення незмінюваних значень.

**№5. Структури даних та алгоритми - Бінарне дерево**

Результат виконання:



class TreeNode

  attr\_accessor :value, :left, :right

  def initialize(value)

    @value = value

    @left = nil

    @right = nil

  end

end

class BinaryTree

  attr\_accessor :root

  def initialize

    @root = nil

  end

  def insert(value)

    @root = insert\_recursive(@root, value)

  end

  def search(value)

    search\_recursive(@root, value)

  end

  def delete(value)

    @root = delete\_recursive(@root, value)

  end

  def inorder

    inorder\_recursive(@root)

  end

  private

  def insert\_recursive(node, value)

    return TreeNode.new(value) if node.nil?

    if value < node.value

      node.left = insert\_recursive(node.left, value)

    elsif value > node.value

      node.right = insert\_recursive(node.right, value)

    end

    node

  end

  def search\_recursive(node, value)

    return false if node.nil?

    if value == node.value

      return true

    elsif value < node.value

      return search\_recursive(node.left, value)

    else

      return search\_recursive(node.right, value)

    end

  end

  def delete\_recursive(node, value)

    return node if node.nil?

    if value < node.value

      node.left = delete\_recursive(node.left, value)

    elsif value > node.value

      node.right = delete\_recursive(node.right, value)

    else

      if node.left.nil?

        return node.right

      elsif node.right.nil?

        return node.left

      end

      node.value = find\_min\_value(node.right)

      node.right = delete\_recursive(node.right, node.value)

    end

    node

  end

  def find\_min\_value(node)

    current = node

    current = current.left until current.left.nil?

    current.value

  end

  def inorder\_recursive(node)

    return [] if node.nil?

    left\_result = inorder\_recursive(node.left)

    current\_result = [node.value]

    right\_result = inorder\_recursive(node.right)

    left\_result + current\_result + right\_result

  end

end

tree = BinaryTree.new

tree.insert(5)

tree.insert(3)

tree.insert(7)

tree.insert(2)

tree.insert(4)

tree.insert(6)

tree.insert(8)

puts "Прямий обхід дерева: #{tree.inorder}"

puts "Пошук 4: #{tree.search(4)}"

puts "Пошук 9: #{tree.search(9)}"

tree.delete(3)

puts "Після видалення 3, Прямий обхід дерева: #{tree.inorder}"