**Portfolio Project**

Jeremy Matthews

Colorado State University Global

CSC400 – Data Structures and Algorithms

Herbert Pensado

Due Date: 10/06/2024

**Portfolio Project**

In this Portfolio Project we are tasked with correcting all previous Critical Thinking Assignments, writing a 2 - 3 page summary of the lessons learned in the Data Structures and Algorithms course, and creating a final program that creates a Person class that contains a string that represents the first and last name of a person and their age. A Queue class is created to store each person in the queue and is able to sort the queue based on the last name or age. The program prompts the user to add five people to the queue. The program then prints the queue and then sorts the queue descending order by name and age. In this document you will find the reflections and lessons learned in this class, the source code, and screenshots of the application executing.

**Reflections and Lessons Learned**

In this 8-week Data Structures and Algorithms course, we explored key concepts related to data structures, algorithm efficiency, and their applications in Java. Here’s a summary of each week’s lessons:

**Week 1**

In the first week we began with Abstract Data Types (ADTs) like lists, queues, stacks, and bags, focusing on the Bag ADT. Bags allow for duplicates and are commonly used in systems like inventory management or word frequency counters. Understanding ADTs laid a strong foundation for the weeks that followed.

**Week 2**

Then in the second week we studied implementing Bags with fixed-size arrays, dynamically resizable arrays, and linked lists. Each data structure offers different benefits, such as the fast access times of arrays or the flexibility of linked lists for frequent insertions and deletions.

**Week 3**

In week 3 we learned about algorithm efficiency and Big O Notation, crucial for evaluating performance, especially with large datasets. By analyzing time complexities like O(1), O(n), and O(n²), we better understood how to choose the right algorithms for various scenarios.

**Week 4**

Then in the fourth week of the course the focus was on the Stack ADT and its applications in algebraic expression evaluation and program execution. Stacks operate on a Last-In, First-Out (LIFO) principle, making them ideal for managing function calls and understanding recursion in Java.

**Week 5**

In week 5 recursion was the key topic, exploring both direct and indirect recursion. We discussed the importance of base cases to avoid infinite loops and analyzed the performance of recursive methods, emphasizing the balance between elegance and efficiency.

**Week 6**

In week 6 we examined sorting algorithms like Selection Sort, Insertion Sort, and Shell Sort. While Selection and Insertion Sorts have O(n²) time complexity, Shell Sort can achieve better performance with O(n log n) in certain cases, making it more suitable for larger datasets.

**Week 7**

In week 7 faster sorting methods like Merge Sort, Quick Sort, and Radix Sort were covered. Merge Sort and Quick Sort are divide-and-conquer algorithms with O(n log n) complexity, while Radix Sort is efficient for sorting numbers based on their individual digits. Each method has specific use cases depending on memory needs and data characteristics.

**Week 8**

In the final week we concluded with Queues, Deques, and Priority Queues. Queues use a First-In, First-Out (FIFO) structure, useful for task scheduling. Deques allow element insertion and removal from both ends, offering more versatility. Priority Queues, implemented with heaps, enable prioritized processing and are essential in algorithms like Dijkstra's shortest path.

**Conclusion**

This 8-week Data Structures and Algorithms course provided a comprehensive understanding of fundamental data structures like ADTs, stacks, queues, and sorting algortihms, along with critical concepts like recursion and algorithm efficiency. Each week’s focus on different structures and techniques equipped me with the skills to analyze and optimize code, making me more proficient in choosing the right tools for various programming challenges. The knowledge gained has deepened my understanding of how data is stored, processed, and managed, which is essential for writing efficient and scalable software. This course has not only enhanced my Java programming skills but also broadened my perspective on solving complex problems, preparing me to tackle real-world programming scenarios with greater confidence and precision.

**Final Program**

**Source Code**

**Main.java**

package com.spiritwisestudios.person;

// Main.java

import java.util.Comparator; // For user input

import java.util.Scanner; // For sorting

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in); // For user input

Queue queue = new Queue(); // Create a new queue

// Prompt the user to add five people to the queue

for (int i = 0; i < 5; i++) {

System.out.println("Enter details for person " + (i + 1));

System.out.print("First Name: ");

String firstName = scanner.nextLine(); // Get the first name

System.out.print("Last Name: ");

String lastName = scanner.nextLine(); // Get the last name

System.out.print("Age: ");

int age = Integer.parseInt(scanner.nextLine()); // Get the age

Person person = new Person(firstName, lastName, age); // Create a new person

queue.enqueue(person); // Add the person to the queue

System.out.println(); // Add a new line

}

// Display the contents of the queue

System.out.println("Contents of the queue:");

queue.displayQueue(); // Display the queue

// Sort the queue descending by last name

queue.quickSort(new Comparator<Person>() {

@Override

public int compare(Person p1, Person p2) { // Compare two people

return p2.getLastName().compareTo(p1.getLastName()); // Compare last names

}

});

// Display the queue after sorting by last name

System.out.println("\nQueue sorted descending by last name:");

queue.displayQueue(); // Display the queue

// Sort the queue descending by age

queue.quickSort(new Comparator<Person>() {

@Override

public int compare(Person p1, Person p2) { // Compare two people

return Integer.compare(p2.getAge(), p1.getAge()); // Compare ages

}

});

// Display the queue after sorting by age

System.out.println("\nQueue sorted descending by age:");

queue.displayQueue(); // Display the queue

scanner.close(); // Close the scanner

}

}

**Person.java**

package com.spiritwisestudios.person;

// Person.java

// This class represents a person with a first name, last name, and age

public class Person {

private String firstName; // First name

private String lastName; // Last name

private int age; // Age

// Constructor

public Person(String firstName, String lastName, int age) {

this.firstName = firstName; // Set the first name

this.lastName = lastName; // Set the last name

this.age = age; // Set the age

}

// Getters

public String getFirstName() {

return firstName; // Return the first name

}

public String getLastName() {

return lastName; // Return the last name

}

public int getAge() {

return age; // Return the age

}

// toString method for easy printing

@Override

public String toString() {

return firstName + " " + lastName + ", Age: " + age; // Return the person's details

}

}

**Queue.java**

package com.spiritwisestudios.person;

import java.util.ArrayList; // For the ArrayList class

import java.util.Comparator; // For sorting

public class Queue {

private ArrayList<Person> queue; // ArrayList to store the queue

public Queue() {

queue = new ArrayList<>(); // Initialize the ArrayList

}

// Enqueue method

public void enqueue(Person p) { // Add a person to the queue

queue.add(p);

}

// Dequeue method

public Person dequeue() { // Remove a person from the queue

if(queue.isEmpty()){ // Check if the queue is empty

return null; // Return null if the queue is empty

}

return queue.remove(0); // Remove and return the first person in the queue

}

// Method to display the queue

public void displayQueue(){

for(Person p : queue) { // Loop through the queue

System.out.println(p); // Print the person

}

}

// QuickSort method

public void quickSort(Comparator<Person> comparator) {

quickSortHelper(0, queue.size() - 1, comparator); // Call the helper method

}

private void quickSortHelper(int low, int high, Comparator<Person> comparator){

if(low < high){ // Check if the low index is less than the high index

int pi = partition(low, high, comparator); // Get the partition index

quickSortHelper(low, pi - 1, comparator); // Recursively sort the left side

quickSortHelper(pi + 1, high, comparator); // Recursively sort the right side

}

}

private int partition(int low, int high, Comparator<Person> comparator) {

Person pivot = queue.get(high); // Set the pivot to the last element

int i = low - 1; // Set the index of the smaller element

for(int j = low; j < high; j++) { // Loop through the elements

if(comparator.compare(queue.get(j), pivot) <= 0) { // Compare the elements

i++; // Increment the index of the smaller element

// Swap queue[i] and queue[j]

Person temp = queue.get(i);

queue.set(i, queue.get(j));

queue.set(j, temp);

}

}

// Swap queue[i+1] and queue[high]

Person temp = queue.get(i + 1);

queue.set(i + 1, queue.get(high)); // Set the pivot element in the correct position

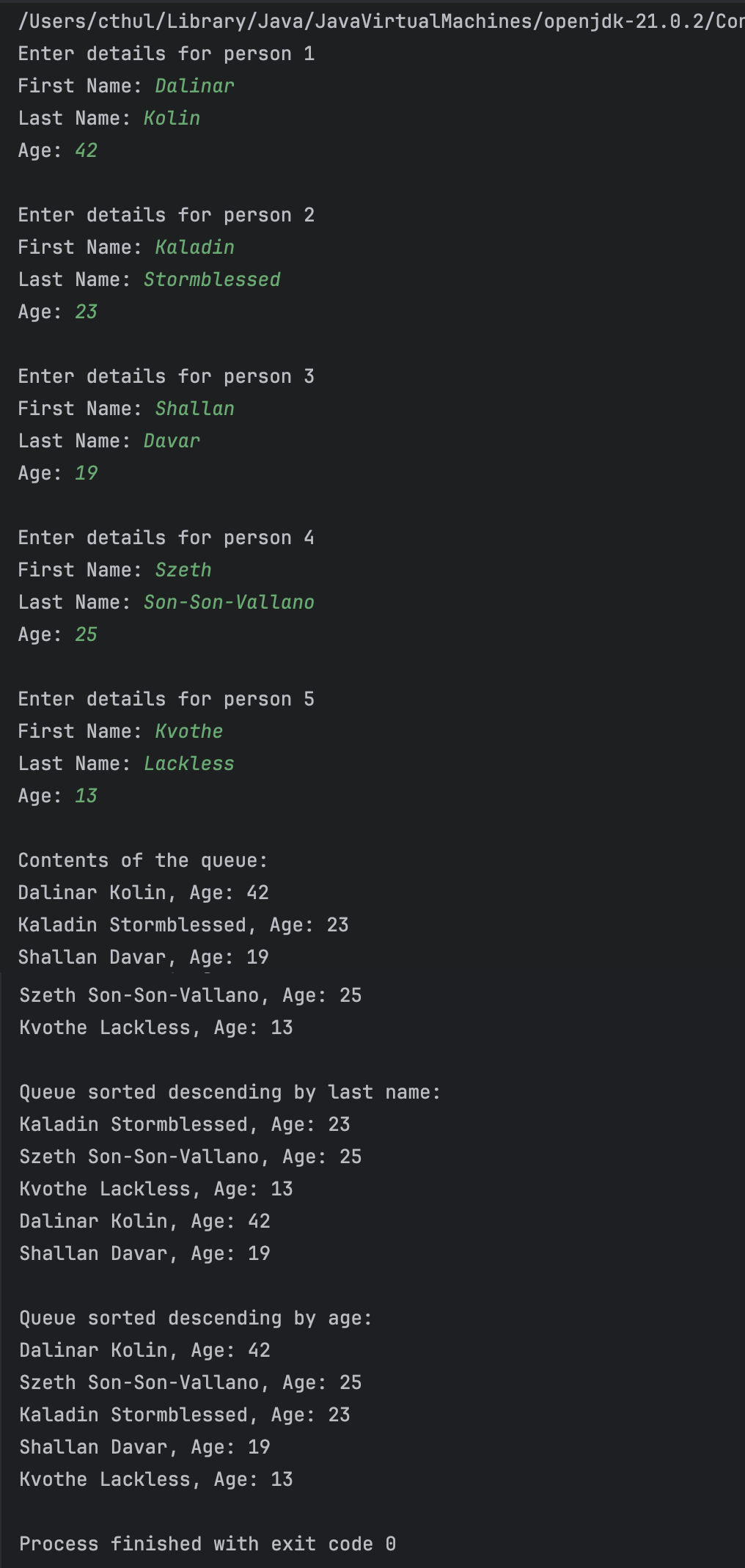
queue.set(high, temp);

return i + 1; // Return the partition index

}

}

**Output Screenshot**

****