Data Structures TABA 2023 Jiaying Yu

Q1. Queue's are a common data structure. Provide an example of three real world examples of where you would implement Queue.

1. Airport security checkpoints: In airports, passengers go through security checkpoints before boarding their flights. Queues are used to manage the flow of passengers waiting to go through the security checkpoint. Passengers are added to the queue in the order they arrive, and are screened by security personnel in the same order.
2. Restaurants: In busy restaurants, customers may need to wait for a table to become available. Queues are used to manage the flow of customers waiting for a table. Customers are added to the queue in the order they arrive, and are seated at a table when one becomes available.
3. Supermarket checkout: In a supermarket, customers are served on a first-come, first-served basis. Queues are used to manage the flow of customers waiting to be served at the checkout counter. Customers join the queue at the back and are served in the order they arrived. This ensures that everyone is treated fairly and no one is given preferential treatment.

Q2. Differentiate between linear and hierarchical data structures, giving suitable examples of each type of data structure.

1. Linear Data Structures:

Linear data structures are those where the data elements are arranged in a linear sequence, with each element having exactly one predecessor and one successor, except for the first and last elements.

The elements of a linear data structure can be traversed in a linear fashion, from the first element to the last, or from the last to the first.

Examples of linear data structures include arrays, linked lists, stacks, and queues. There are some details of these examples:

Arrays: An array is a collection of elements of the same data type stored in contiguous memory locations. It allows for constant-time access to any element in the array, but inserting or deleting elements can be time-consuming if done in the middle of the array.

Linked Lists: A linked list is a sequence of nodes, where each node contains a data element and a reference to the next node in the list.

Linked lists are easy to insert or delete elements from, but accessing a specific element can be time-consuming as we have to traverse the list sequentially.

Stacks: A stack is a linear data structure where elements are inserted and removed from one end only, called the top of the stack. The last element inserted into the stack is the first element to be removed from the stack, it should be Last In First Out.

Queues: A queue is a linear data structure where elements are inserted from one end and removed from the other end, called the front and the rear of the queue, respectively. The first element inserted into the queue is the first element to be removed from the queue, it should be First In First Out.

2. Hierarchical Data Structures:

Hierarchical data structures are those where the data elements are arranged in a tree-like structure, with each element having one parent and zero or more children, except for the root node, which has no parent.
The elements of a hierarchical data structure can be traversed in a non-linear fashion, moving up or down the tree to visit different branches or nodes.
Examples of hierarchical data structures include trees, binary trees, heaps, and graphs. There are some details of these examples:
Trees: A tree is a hierarchical data structure where each node has zero or more child nodes. The topmost node of a tree is called the root node, and nodes with no children are called leaf nodes. Examples of trees include the file system structure of a computer and a company's organizational chart.
Binary Trees: A binary tree is a tree where each node has at most two child nodes, known as the left child and the right child. Binary trees are used in many applications such as searching and sorting algorithms, and expression trees.
Heaps: A heap is a specialized tree-based data structure where the tree satisfies the heap property, which states that the parent node is either greater than or less than both of its child nodes. Heaps are commonly used to implement priority queues.
Graphs: A graph is a collection of nodes and edges, where each node represents a data element, and each edge represents a relationship between the nodes. Graphs can be used to model a wide range of real-world scenarios, such as social networks, transportation networks, and computer networks.

Q3.
The program entry is the Main class, below is the output.

```
Tree is empty: true
Add 10 person
Tree is empty: false
Tree size: 10
Oldest person: Eva
Youngest person: Fiona
Gender ratio (Male/Female): 1.00
Tree representation:
            Eva(60)
                Igor(55)
        Charlie(50)
    Bob(45)
        George(42)
Alice(35)
        Jasmine(31)
```

```
    David(28)
            Helen(26)
        Fiona(18)
Removing Alice: true
New tree size: 9
Tree representation:
            Eva(60)
                Igor(55)
        Charlie(50)
    Bob(45)
George(42)
        Jasmine(31)
    David(28)
            Helen(26)
        Fiona(18)
```

Main.java

```
/**
 * Jiaying Yu
 * The Main class demonstrates the functionality of the binary search tree for storing Person objects.
 * It creates a binary search tree, adds ten persons to it, and performs various operations.
 */
public class Main {
    public static void main(String[] args) {
        // Create a new binary search tree to store Person objects
        BinaryTree<Person> familyTree = new BinarySearchTree<>();

        // Check if the tree is empty
        System.out.println("Tree is empty: " + familyTree.isEmpty());
        System.out.println("Add 10 person");

        // Add Person objects to the family tree
        familyTree.add(new Person("Alice", 35, "Software Engineer", "Software Engineer", "Married", "Female", "123 Elm St"));
        familyTree.add(new Person("Bob", 45, "Doctor", "Doctor", "Single", "Male", "456 Maple St"));
        familyTree.add(new Person("Charlie", 50, "Teacher", "Teacher", "Married", "Male", "789 Oak St"));
        familyTree.add(new Person("David", 28, "Designer", "Designer", "Single", "Male", "321 Birch St"));
        familyTree.add(new Person("Eva", 60, "Nurse", "Nurse", "Married", "Female", "654 Willow St"));
        familyTree.add(new Person("Fiona", 18, "Student", "Student", "Single", "Female", "987 Pine St"));
```

```
        familyTree.add(new Person("George", 42, "Lawyer", "Lawyer", "Married",
"Male", "147 Cherry St"));
        familyTree.add(new Person("Helen", 26, "Accountant", "Accountant", "Single",
"Female", "369 Cedar St"));
        familyTree.add(new Person("Igor", 55, "Architect", "Architect", "Divorced",
"Male", "951 Apple St"));
        familyTree.add(new Person("Jasmine", 31, "Chef", "Chef", "Married", "Female",
"753 Peach St"));

        // Check if the tree is empty and print the tree size
        System.out.println("Tree is empty: " + familyTree.isEmpty());
        System.out.println("Tree size: " + familyTree.size());

        // Find and print the oldest and youngest persons' name
        System.out.println("Oldest person: " + familyTree.findOldest().getName());
        System.out.println("Youngest person: " + familyTree.findYoungest().getName());

        // Calculate and print the gender ratio (Male/Female)
        System.out.printf("Gender ratio (Male/Female): %.2f\n",
familyTree.genderRatio());
        System.out.println("Tree representation:");

        // Print the tree representation
        familyTree.printTree();
        /*
         * The leftmost side of the console represents the root of the tree.
         * The rightmost side of the console represents the leaf nodes of the tree.
         * The person with the highest age is located at the top-right of the console (the
rightmost leaf node).
         * The person with the lowest age is located at the bottom-left of the console
(the leftmost leaf node).
         */

        // Remove a person by name and print the new tree size
        System.out.println("Removing Alice: " + familyTree.removeByName("Alice"));
        System.out.println("New tree size: " + familyTree.size());

        // Print the updated tree representation
        System.out.println("Tree representation:");
        familyTree.printTree();
    }
}
```

BinaryTree.java

```
/**
 * A generic BinaryTree interface that defines the basic operations for a binary tree.
 * The interface is parameterized with a type T which must implement the
Comparable<T> interface.
```

```
 * This ensures that the BinaryTree interface can work with any type that implements Comparable,
 * providing type safety and code reusability.
 *
 * @param <T> The type of the values stored in the BinaryTree, which must implement Comparable<T>
 */
public interface BinaryTree<T extends Comparable<T>> {
    /**
     * Checks if the binary tree is empty.
     *
     * @return true if the tree is empty, false otherwise
     */
    boolean isEmpty();

    /**
     * Adds a value to the binary tree.
     *
     * @param value The value to be added
     */
    void add(T value);
    /**
     * Returns the number of elements in the binary tree.
     *
     * @return The size of the binary tree
     */
    int size();

    /**
     * Finds the oldest person in the binary tree.
     *
     * @return The oldest person
     */
    T findOldest();

    /**
     * Finds the youngest person in the binary tree.
     *
     * @return The youngest person
     */
    T findYoungest();

    /**
     * Calculates the gender ratio (Male/Female) in the binary tree.
     *
     * @return The gender ratio
     */
    double genderRatio();
```

```
    /**
     * Removes a person from the binary tree by their name, if they exist.
     *
     * @param name The name of the person to be removed
     * @return true if the person was removed, false otherwise
     */
    boolean removeByName(String name);

    /**
     * Prints the binary tree representation, typically in an indented format.
     */
    void printTree();
}
```

BinarySearchTree.java

```
/**
 * Jiaying Yu
 * The BinarySearchTree class implements the BinaryTree interface using a binary
search tree data structure.
 * It allows storing elements of type T, which must implement the Comparable
interface for comparison.
 * The binary search tree maintains the elements in sorted order based on their
natural ordering.
 */
public class BinarySearchTree<T extends Comparable<T>> implements
BinaryTree<T> {
    private Node<T> root;
    private int size;

    public BinarySearchTree() {
        root = null;
        size = 0;
    }

    // Checks if the tree is empty
    @Override
    public boolean isEmpty() {
        return root == null;
    }

    // Adds a new value to the tree
    @Override
    public void add(T value) {
        if (root == null) {
            root = new Node<>(value);
        } else {
            addRecursive(root, value);
```

```
    }
    size++;
}

// Helper method for adding a new value recursively
private Node<T> addRecursive(Node<T> current, T value) {
    if (current == null) {
        return new Node<>(value);
    }

    if (value.compareTo(current.value) < 0) {
        current.left = addRecursive(current.left, value);
    } else if (value.compareTo(current.value) > 0) {
        current.right = addRecursive(current.right, value);
    } else {
        return current;
    }

    return current;
}

// Returns the size of the tree
@Override
public int size() {
    return size;
}

// Finds the oldest person in the tree
@Override
public T findOldest() {
    if (isEmpty()) {
        return null;
    }

    Node<T> current = root;
    while (current.right != null) {
        current = current.right;
    }
    return current.value;
}

// Finds the youngest person in the tree
@Override
public T findYoungest() {
    if (isEmpty()) {
        return null;
    }
```

```
    Node<T> current = root;
    while (current.left != null) {
        current = current.left;
    }
    return current.value;
}

// Calculates the gender ratio (male / female) in the tree
@Override
public double genderRatio() {
    int[] counts = genderCounts(root);
    return (double) counts[0] / counts[1];
}

// Helper method for counting genders recursively
private int[] genderCounts(Node<T> current) {
    if (current == null) {
        return new int[]{0, 0};
    }

    int[] leftCounts = genderCounts(current.left);
    int[] rightCounts = genderCounts(current.right);

    int maleCount = leftCounts[0] + rightCounts[0];
    int femaleCount = leftCounts[1] + rightCounts[1];

    if (((Person) current.value).getGender().equalsIgnoreCase("male")) {
        maleCount++;
    } else {
        femaleCount++;
    }

    return new int[]{maleCount, femaleCount};
}

// Removes a person from the tree by their name
public boolean removeByName(String name) {
    if (isEmpty()) {
        return false;
    }
    int initialSize = size;
    root = removeByNameRecursive(root, name);
    return size < initialSize;
}

// Helper method for removing a person by name recursively
private Node<T> removeByNameRecursive(Node<T> current, String name) {
    if (current == null) {
```

```
            return null;
        }

        Person person = (Person) current.value;
        if (person.getName().equals(name)) {
            size--;
            if (current.left == null && current.right == null) {
                return null;
            } else if (current.right == null) {
                return current.left;
            } else if (current.left == null) {
                return current.right;
            } else {
                T smallestValue = findSmallestValue(current.right);
                current.value = smallestValue;
                current.right = removeByNameRecursive(current.right, ((Person)
smallestValue).getName());
                size++;
                return current;
            }
        }
        current.left = removeByNameRecursive(current.left, name);
        current.right = removeByNameRecursive(current.right, name);
        return current;
    }

    // Helper method for finding the smallest value in a subtree
    private T findSmallestValue(Node<T> current) {
        if (current.left == null) {
            return current.value;
        }
        return findSmallestValue(current.left);
    }

    // Prints the tree with indentation representing the tree structure
    @Override
    public void printTree() {
        printTreeRecursive(root, 0);
    }

    // Helper method for printing the tree recursively
    private void printTreeRecursive(Node<T> current, int depth) {
        if (current == null) {
            return;
        }

        // Print the right subtree
        printTreeRecursive(current.right, depth + 1);
```

```
        // Print the current node value with appropriate indentation
        for (int i = 0; i < depth; i++) {
            System.out.print("\t");
        }
        System.out.println(current.value);

        // Print the left subtree
        printTreeRecursive(current.left, depth + 1);
    }
}
```

Node.java

```
/**
 * Q3.C.
 * Jiaying Yu
 * A generic Node class for a binary tree.
 * The class is parameterized with a type T which must implement the
Comparable<T> interface.
 * This ensures that the Node class can store any type that implements Comparable,
providing
 * type safety and code reusability.
 *
 * @param <T> The type of the value stored in the Node, which must implement
Comparable<T>
 */
public class Node<T extends Comparable<T>> {
    T value; // The value stored in the Node
    Node<T> left; // Reference to the left child Node
    Node<T> right; // Reference to the right child Node

    /**
     * Constructor for the Node class.
     * Initializes the value, and sets the left and right child Nodes to null.
     *
     * @param value The value to be stored in the Node
     */
    public Node(T value) {
        this.value = value;
        left = null;
        right = null;
    }
}
```

Person.java

```
import java.util.Objects;
/**
```

```
 * Person class representing a person with basic attributes such as name, age,
occupation, job,
 * marital status, gender, and address. This class implements the
Comparable<Person> interface
 * which allows Person objects to be compared based on their age.
 */
public class Person implements Comparable<Person> {
    private String name;
    private int age;
    private String occupation;
    private String job;
    private String maritalStatus;
    private String gender;
    private String address;

    /**
     * Constructs a new Person object with the specified attributes.
     *
     * @param name The name of the person
     * @param age The age of the person
     * @param occupation The occupation of the person
     * @param job The job of the person
     * @param maritalStatus The marital status of the person
     * @param gender The gender of the person
     * @param address The address of the person
     */
    public Person(String name, int age, String occupation, String job, String
maritalStatus, String gender, String address) {
        this.name = name;
        this.age = age;
        this.occupation = occupation;
        this.job = job;
        this.maritalStatus = maritalStatus;
        this.gender = gender;
        this.address = address;
    }

    // Getter and setter methods for each attribute
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
```

```
}

public void setAge(int age) {
    this.age = age;
}

public String getOccupation() {
    return occupation;
}

public void setOccupation(String occupation) {
    this.occupation = occupation;
}

public String getJob() {
    return job;
}

public void setJob(String job) {
    this.job = job;
}

public String getMaritalStatus() {
    return maritalStatus;
}

public void setMaritalStatus(String maritalStatus) {
    this.maritalStatus = maritalStatus;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

@Override
public boolean equals(Object o) {
```

```
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Person person = (Person) o;
        return Objects.equals(name, person.name);
    }

    /**
     * Compares this Person object with the specified Person object for order.
     * The comparison is based on the age of the Person objects.
     *
     * @param other The Person object to be compared
     * @return A negative integer, zero, or a positive integer as this Person's age is
less than,
     *         equal to, or greater than the specified Person's age
     */
    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age);
    }

    /**
     * Returns a string representation of the Person object, displaying the name and
age.
     *
     * @return A string representation of the Person object
     */
    @Override
    public String toString() {
        return name+"("+age+")" ;
    }
}
```