Questo documento fornisce implementazioni Java per le strutture dati menzionate nei tuoi appunti, seguendo un ordine di complessità crescente.

Indice

Definizioni delle Interfacce

Iniziamo definendo le interfacce di base che le nostre strutture dati implementeranno:

```
/**
 * Interfaccia di base per le strutture dati contenitore
 */
public interface Contenitore {
    /**
     * Verifica se il contenitore è vuoto
     * @return true se vuoto, false altrimenti
     */
    boolean isEmpty();

    /**
     * Restituisce il numero di elementi nel contenitore
     * @return la dimensione del contenitore
     */
    int size();
}

/**
 * Interfaccia per le strutture dati di tipo pila (LIFO)
 */
public interface Pila<T> extends Contenitore {
    /**
     * Aggiunge un elemento in cima alla pila
     * @param elem l'elemento da aggiungere
```

```
     */
    void push(T elem);

    /**
     * Rimuove e restituisce l'elemento in cima
     * @return l'elemento in cima
     * @throws EmptyStackException se la pila è vuota
     */
    T pop();

    /**
     * Restituisce l'elemento in cima senza rimuoverlo
     * @return l'elemento in cima
     * @throws EmptyStackException se la pila è vuota
     */
    T top();
}

/**
 * Interfaccia per le strutture dati di tipo coda (FIFO)
 */
public interface Coda<T> extends Contenitore {
    /**
     * Aggiunge un elemento alla fine della coda
     * @param elem l'elemento da aggiungere
     */
    void enqueue(T elem);

    /**
     * Rimuove e restituisce l'elemento in testa
     * @return l'elemento in testa
     * @throws NoSuchElementException se la coda è vuota
     */
    T dequeue();

    /**
     * Restituisce l'elemento in testa senza rimuoverlo
     * @return l'elemento in testa
     * @throws NoSuchElementException se la coda è vuota
     */
    T getFront();
}
```

Implementazioni di ArrayList

Implementazione di ArrayListInteger

Una semplice implementazione di ArrayList specificamente per interi:

```java
public class ArrayListInteger {
    private int[] elements;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * Costruisce una lista vuota con capacità predefinita
     */
    public ArrayListInteger() {
        elements = new int[DEFAULT_CAPACITY];
        size = 0;
    }

    /**
     * Costruisce una lista vuota con capacità specificata
     */
    public ArrayListInteger(int initialCapacity) {
        if (initialCapacity < 0) {
            throw new IllegalArgumentException("Capacità illegale: " +
initialCapacity);
        }
        elements = new int[initialCapacity];
        size = 0;
    }

    /**
     * Restituisce il numero di elementi nella lista
     */
    public int size() {
        return size;
    }

    /**
     * Verifica se la lista è vuota
     */
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * Aggiunge un intero alla fine della lista
     */
    public void add(int e) {
        ensureCapacity(size + 1);
        elements[size++] = e;
    }

    /**
     * Aggiunge un intero alla posizione specificata
```

```
     */
    public void add(int index, int e) {
        rangeCheckForAdd(index);
        ensureCapacity(size + 1);

        // Sposta gli elementi a destra
        System.arraycopy(elements, index, elements, index + 1, size -
index);
        elements[index] = e;
        size++;
    }

    /**
     * Ottiene l'elemento alla posizione specificata
     */
    public int get(int index) {
        rangeCheck(index);
        return elements[index];
    }

    /**
     * Sostituisce l'elemento alla posizione specificata
     */
    public int set(int index, int e) {
        rangeCheck(index);
        int oldValue = elements[index];
        elements[index] = e;
        return oldValue;
    }

    /**
     * Rimuove l'elemento alla posizione specificata
     */
    public int remove(int index) {
        rangeCheck(index);

        int oldValue = elements[index];

        // Sposta gli elementi a sinistra
        int numMoved = size - index - 1;
        if (numMoved > 0) {
            System.arraycopy(elements, index + 1, elements, index,
numMoved);
        }

        size--;
        return oldValue;
    }

    /**
```

```java
     * Assicura la capacità per aggiungere elementi
     */
    private void ensureCapacity(int minCapacity) {
        if (minCapacity > elements.length) {
            int newCapacity = Math.max(elements.length * 3/2, minCapacity);
            int[] newArray = new int[newCapacity];
            System.arraycopy(elements, 0, newArray, 0, size);
            elements = newArray;
        }
    }

    /**
     * Verifica se l'indice fornito è nell'intervallo valido
     */
    private void rangeCheck(int index) {
        if (index >= size || index < 0) {
            throw new IndexOutOfBoundsException("Indice: " + index + ",
Dimensione: " + size);
        }
    }

    /**
     * Verifica se l'indice fornito è nell'intervallo valido per
l'operazione di aggiunta
     */
    private void rangeCheckForAdd(int index) {
        if (index > size || index < 0) {
            throw new IndexOutOfBoundsException("Indice: " + index + ",
Dimensione: " + size);
        }
    }

    /**
     * Restituisce una rappresentazione in stringa di questa lista
     */
    @Override
    public String toString() {
        if (size == 0) {
            return "[]";
        }

        StringBuilder sb = new StringBuilder();
        sb.append('[');
        for (int i = 0; i < size; i++) {
            sb.append(elements[i]);
            if (i < size - 1) {
                sb.append(", ");
            }
        }
        sb.append(']');
```

```
        return sb.toString();
    }
}
```

Implementazioni di Stack

Implementazione di PilaArrayList

Un'implementazione di stack utilizzando ArrayList:

```
import java.util.ArrayList;
import java.util.EmptyStackException;

public class PilaArrayList<T> implements Pila<T> {
    private ArrayList<T> elements;

    /**
     * Costruisce uno stack vuoto
     */
    public PilaArrayList() {
        elements = new ArrayList<>();
    }

    /**
     * Verifica se lo stack è vuoto
     */
    @Override
    public boolean isEmpty() {
        return elements.isEmpty();
    }

    /**
     * Restituisce il numero di elementi nello stack
     */
    @Override
    public int size() {
        return elements.size();
    }

    /**
     * Aggiunge un elemento in cima allo stack
     */
    @Override
    public void push(T elem) {
        elements.add(elem);
    }

    /**
```

```
     * Rimuove e restituisce l'elemento in cima
     */
    @Override
    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return elements.remove(elements.size() - 1);
    }

    /**
     * Restituisce l'elemento in cima senza rimuoverlo
     */
    @Override
    public T top() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return elements.get(elements.size() - 1);
    }

    /**
     * Restituisce una rappresentazione in stringa di questo stack
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Stack: [");

        for (int i = 0; i < elements.size(); i++) {
            sb.append(elements.get(i));
            if (i < elements.size() - 1) {
                sb.append(", ");
            }
        }

        sb.append("] <- cima");
        return sb.toString();
    }
}
```

Implementazioni di Code

Implementazione di CodaArrayList

Un'implementazione di coda utilizzando ArrayList:

```java
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class CodaArrayList<T> implements Coda<T> {
    private ArrayList<T> elements;

    /**
     * Costruisce una coda vuota
     */
    public CodaArrayList() {
        elements = new ArrayList<>();
    }

    /**
     * Verifica se la coda è vuota
     */
    @Override
    public boolean isEmpty() {
        return elements.isEmpty();
    }

    /**
     * Restituisce il numero di elementi nella coda
     */
    @Override
    public int size() {
        return elements.size();
    }

    /**
     * Aggiunge un elemento alla fine della coda
     */
    @Override
    public void enqueue(T elem) {
        elements.add(elem);
    }

    /**
     * Rimuove e restituisce l'elemento frontale
     */
    @Override
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("La coda è vuota");
        }
        return elements.remove(0);
    }

    /**
```

```
     * Restituisce l'elemento frontale senza rimuoverlo
     */
    @Override
    public T getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("La coda è vuota");
        }
        return elements.get(0);
    }

    /**
     * Restituisce una rappresentazione in stringa di questa coda
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Coda: [");

        for (int i = 0; i < elements.size(); i++) {
            sb.append(elements.get(i));
            if (i < elements.size() - 1) {
                sb.append(", ");
            }
        }

        sb.append("] <- fine");
        return sb.toString();
    }
}
```

Implementazione di CodaArrayListEstremoFissoLiberato

Un'implementazione di coda utilizzando ArrayList con estremità frontale fissa e ottimizzata per operazioni di dequeue O(1):

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class CodaArrayListEstremoFissoLiberato<T> implements Coda<T> {
    private ArrayList<T> elements;
    private int front;  // Indice dell'elemento frontale

    /**
     * Costruisce una coda vuota
     */
    public CodaArrayListEstremoFissoLiberato() {
        elements = new ArrayList<>();
        front = 0;
```

```
    }

    /**
     * Verifica se la coda è vuota
     */
    @Override
    public boolean isEmpty() {
        return front >= elements.size();
    }

    /**
     * Restituisce il numero di elementi nella coda
     */
    @Override
    public int size() {
        return elements.size() - front;
    }

    /**
     * Aggiunge un elemento alla fine della coda
     */
    @Override
    public void enqueue(T elem) {
        elements.add(elem);

        // Compatta la lista se ci sono troppe posizioni nulle/inutilizzate
        if (front > elements.size() / 2 && front > 10) {
            compact();
        }
    }

    /**
     * Rimuove e restituisce l'elemento frontale
     */
    @Override
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("La coda è vuota");
        }

        T value = elements.get(front);
        elements.set(front, null);  // Aiuta con la garbage collection
        front++;

        return value;
    }

    /**
     * Restituisce l'elemento frontale senza rimuoverlo
     */
```

```
    @Override
    public T getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("La coda è vuota");
        }
        return elements.get(front);
    }

    /**
     * Compatta la lista rimuovendo gli elementi null all'inizio
     */
    private void compact() {
        int size = elements.size();
        for (int i = 0; i < size - front; i++) {
            elements.set(i, elements.get(i + front));
        }

        for (int i = size - front; i < size; i++) {
            elements.remove(size - front);
        }

        front = 0;
    }

    /**
     * Restituisce una rappresentazione in stringa di questa coda
     */
    @Override
    public String toString() {
        if (isEmpty()) {
            return "Coda: []";
        }

        StringBuilder sb = new StringBuilder();
        sb.append("Coda: [");

        for (int i = front; i < elements.size(); i++) {
            sb.append(elements.get(i));
            if (i < elements.size() - 1) {
                sb.append(", ");
            }
        }

        sb.append("] <- fine");
        return sb.toString();
    }
}
```

Coda Circolare

Implementazione di CodaCircolareArrayInteger

Un'implementazione di coda circolare specifica per interi:

```
import java.util.NoSuchElementException;

public class CodaCircolareArrayInteger implements Coda<Integer> {
    private int[] elements;
    private int front;  // Indice dell'elemento frontale
    private int rear;   // Indice dell'ultimo elemento
    private int size;   // Numero attuale di elementi
    private static final int DEFAULT_CAPACITY = 10;

    /**
     * Costruisce una coda circolare vuota con capacità predefinita
     */
    public CodaCircolareArrayInteger() {
        elements = new int[DEFAULT_CAPACITY];
        front = 0;
        rear = -1;
        size = 0;
    }

    /**
     * Costruisce una coda circolare vuota con capacità specificata
     */
    public CodaCircolareArrayInteger(int capacity) {
        elements = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    /**
     * Verifica se la coda è vuota
     */
    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * Verifica se la coda è piena
     */
    public boolean isFull() {
        return size == elements.length;
    }

    /**
```

```java
     * Restituisce il numero di elementi nella coda
     */
    @Override
    public int size() {
        return size;
    }

    /**
     * Aggiunge un elemento alla fine della coda
     */
    @Override
    public void enqueue(Integer elem) {
        if (isFull()) {
            throw new IllegalStateException("Coda piena");
        }

        rear = (rear + 1) % elements.length;
        elements[rear] = elem;
        size++;
    }

    /**
     * Rimuove e restituisce l'elemento frontale
     */
    @Override
    public Integer dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("Coda vuota");
        }

        int value = elements[front];
        front = (front + 1) % elements.length;
        size--;

        return value;
    }

    /**
     * Restituisce l'elemento frontale senza rimuoverlo
     */
    @Override
    public Integer getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("Coda vuota");
        }
        return elements[front];
    }

    /**
     * Restituisce una rappresentazione in stringa di questa coda
```

```
     */
    @Override
    public String toString() {
        if (isEmpty()) {
            return "Coda Circolare: []";
        }

        StringBuilder sb = new StringBuilder();
        sb.append("Coda Circolare: [");

        int count = 0;
        int i = front;

        while (count < size) {
            sb.append(elements[i]);
            i = (i + 1) % elements.length;
            count++;

            if (count < size) {
                sb.append(", ");
            }
        }

        sb.append("]");
        return sb.toString();
    }
}
```

Coda Doppia (Deque)

Implementazione di un metodo per verificare i palindromi usando Deque

```
import java.util.ArrayDeque;
import java.util.Deque;

public class PalindromeChecker {

    /**
     * Verifica se una stringa è un palindromo utilizzando un Deque
     * @param input la stringa da verificare
     * @return true se la stringa è un palindromo, false altrimenti
     */
    public static boolean isPalindrome(String input) {
        if (input == null) {
            return false;
        }
```

```
        // Rimuove spazi e caratteri non alfanumerici e converte in
minuscolo
        String cleanInput = input.replaceAll("[^a-zA-Z0-9]",
"").toLowerCase();

        if (cleanInput.isEmpty()) {
            return true;  // Una stringa vuota è palindroma per definizione
        }

        Deque<Character> deque = new ArrayDeque<>();

        // Inserisce tutti i caratteri nel deque
        for (char c : cleanInput.toCharArray()) {
            deque.addLast(c);
        }

        // Confronta i caratteri dalle due estremità
        while (deque.size() > 1) {
            char first = deque.removeFirst();
            char last = deque.removeLast();

            if (first != last) {
                return false;
            }
        }

        return true;
    }

    // Esempio di utilizzo
    public static void main(String[] args) {
        System.out.println("'radar' è palindromo? " +
isPalindrome("radar"));
        System.out.println("'A man, a plan, a canal: Panama' è palindromo? "
+
                isPalindrome("A man, a plan, a canal: Panama"));
        System.out.println("'hello' è palindromo? " +
isPalindrome("hello"));
    }
}
```

Coda Prioritaria

Implementazione di una classe Coppia e un Comparator per PriorityQueue

```
import java.util.Comparator;
import java.util.PriorityQueue;
```

```java
/**
 * Classe che rappresenta una coppia (descrizione, priorità)
 */
public class Coppia {
    private String descrizione;
    private int priorita;

    /**
     * Costruisce una coppia con descrizione e priorità
     */
    public Coppia(String descrizione, int priorita) {
        this.descrizione = descrizione;
        this.priorita = priorita;
    }

    /**
     * Restituisce la descrizione
     */
    public String getDescrizione() {
        return descrizione;
    }

    /**
     * Restituisce la priorità
     */
    public int getPriorita() {
        return priorita;
    }

    /**
     * Restituisce una rappresentazione in stringa di questa coppia
     */
    @Override
    public String toString() {
        return "(" + descrizione + ", " + priorita + ")";
    }
}

/**
 * Comparatore per ordinare le coppie in base alla priorità
 */
class ComparatoreCoppia implements Comparator<Coppia> {

    /**
     * Confronta due coppie in base alla priorità
     * @return negativo se o1 ha priorità maggiore di o2, zero se uguali,
positivo altrimenti
     */
    @Override
```

```
    public int compare(Coppia o1, Coppia o2) {
        // Priorità minore = priorità più alta (min-heap)
        return o1.getPriorita() - o2.getPriorita();
    }
}

/**
 * Classe di esempio per dimostrare l'uso di PriorityQueue con il
comparatore
 */
class EsempioPriorityQueue {
    public static void main(String[] args) {
        // Crea una coda prioritaria utilizzando il comparatore
personalizzato
        PriorityQueue<Coppia> codaPrioritaria = new PriorityQueue<>(new
ComparatoreCoppia());

        // Aggiunge coppie alla coda
        codaPrioritaria.add(new Coppia("Task normale", 3));
        codaPrioritaria.add(new Coppia("Task urgente", 1));
        codaPrioritaria.add(new Coppia("Task bassa priorità", 5));
        codaPrioritaria.add(new Coppia("Task alta priorità", 2));

        // Estrae gli elementi in ordine di priorità
        System.out.println("Elementi in ordine di priorità:");
        while (!codaPrioritaria.isEmpty()) {
            System.out.println(codaPrioritaria.poll());
        }
    }
}
```

Lista Collegata

Esempio di utilizzo di LinkedList

```
import java.util.LinkedList;
import java.util.Iterator;

public class EsempioLinkedList {
    public static void main(String[] args) {
        // Crea una LinkedList di stringhe
        LinkedList<String> lista = new LinkedList<>();

        // Aggiunge elementi alla lista
        lista.add("Primo");
        lista.add("Secondo");
        lista.add("Quarto");
```

```
        // Inserisce un elemento a un indice specifico
        lista.add(2, "Terzo");

        // Aggiunge elementi all'inizio e alla fine
        lista.addFirst("Inizio");
        lista.addLast("Fine");

        // Stampa la lista
        System.out.println("Lista completa: " + lista);

        // Accede agli elementi
        System.out.println("Primo elemento: " + lista.getFirst());
        System.out.println("Ultimo elemento: " + lista.getLast());
        System.out.println("Elemento all'indice 2: " + lista.get(2));

        // Rimuove elementi
        lista.removeFirst();
        lista.removeLast();
        lista.remove(1);

        System.out.println("Lista dopo le rimozioni: " + lista);

        // Verifica se un elemento è presente
        System.out.println("Contiene 'Terzo'? " + lista.contains("Terzo"));

        // Itera sulla lista
        System.out.println("Iterazione sulla lista:");
        Iterator<String> iterator = lista.iterator();
        while (iterator.hasNext()) {
            System.out.println("- " + iterator.next());
        }

        // Itera sulla lista in modo più conciso
        System.out.println("Iterazione concisa:");
        for (String elemento : lista) {
            System.out.println("- " + elemento);
        }

        // Svuota la lista
        lista.clear();
        System.out.println("La lista è vuota? " + lista.isEmpty());
    }
}
```

Insiemi

Esempio di utilizzo di HashSet

```java
import java.util.HashSet;

public class EsempioHashSet {
    public static void main(String[] args) {
        // Crea un HashSet di stringhe
        HashSet<String> set = new HashSet<>();

        // Aggiunge elementi al set
        set.add("Mela");
        set.add("Banana");
        set.add("Arancia");
        set.add("Mela");  // Non verrà aggiunto perché è un duplicato

        // Stampa il set
        System.out.println("Set: " + set);

        // Verifica la dimensione
        System.out.println("Dimensione del set: " + set.size());

        // Verifica se un elemento è presente
        System.out.println("Contiene 'Banana'? " + set.contains("Banana"));
        System.out.println("Contiene 'Pera'? " + set.contains("Pera"));

        // Rimuove un elemento
        set.remove("Banana");
        System.out.println("Set dopo la rimozione: " + set);

        // Itera sul set
        System.out.println("Elementi nel set:");
        for (String frutto : set) {
            System.out.println("- " + frutto);
        }

        // Crea un altro set
        HashSet<String> altroSet = new HashSet<>();
        altroSet.add("Kiwi");
        altroSet.add("Mela");

        // Unione di set
        HashSet<String> unione = new HashSet<>(set);
        unione.addAll(altroSet);
        System.out.println("Unione: " + unione);

        // Intersezione di set
        HashSet<String> intersezione = new HashSet<>(set);
        intersezione.retainAll(altroSet);
        System.out.println("Intersezione: " + intersezione);

        // Differenza di set
```

```
        HashSet<String> differenza = new HashSet<>(set);
        differenza.removeAll(altroSet);
        System.out.println("Differenza: " + differenza);

        // Svuota il set
        set.clear();
        System.out.println("Il set è vuoto? " + set.isEmpty());
    }
}
```

Mappe

Implementazione del metodo nRicorrenze usando HashMap

```
import java.util.HashMap;
import java.util.Map;

public class ContaRicorrenze {

    /**
     * Conta e stampa le ricorrenze di ogni carattere in una stringa
     * @param s la stringa da analizzare
     */
    public static void nRicorrenze(String s) {
        if (s == null || s.isEmpty()) {
            System.out.println("La stringa è vuota o null");
            return;
        }

        // Crea una HashMap per memorizzare i caratteri e le loro ricorrenze
        HashMap<Character, Integer> ricorrenze = new HashMap<>();

        // Conta le ricorrenze di ciascun carattere
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);

            // Aggiorna il conteggio per il carattere corrente
            ricorrenze.put(c, ricorrenze.getOrDefault(c, 0) + 1);
        }

        // Stampa i risultati
        System.out.println("Ricorrenze dei caratteri in \"" + s + "\":");
        for (Map.Entry<Character, Integer> entry : ricorrenze.entrySet()) {
            System.out.println("'" + entry.getKey() + "': " +
entry.getValue());
        }
    }
```

```java
    /**
     * Conta e stampa le ricorrenze di ogni parola in una stringa
     * @param s la stringa da analizzare
     */
    public static void nRicorrenzeParole(String s) {
        if (s == null || s.isEmpty()) {
            System.out.println("La stringa è vuota o null");
            return;
        }

        // Dividi la stringa in parole
        String[] parole = s.split("\\s+");

        // Crea una HashMap per memorizzare le parole e le loro ricorrenze
        HashMap<String, Integer> ricorrenze = new HashMap<>();

        // Conta le ricorrenze di ciascuna parola
        for (String parola : parole) {
            // Normalizza la parola (rimuovi punteggiatura e converti in
minuscolo)
            parola = parola.replaceAll("[^a-zA-Z]", "").toLowerCase();

            if (!parola.isEmpty()) {
                ricorrenze.put(parola, ricorrenze.getOrDefault(parola, 0) +
1);
            }
        }

        // Stampa i risultati
        System.out.println("Ricorrenze delle parole:");
        for (Map.Entry<String, Integer> entry : ricorrenze.entrySet()) {
            System.out.println("\"" + entry.getKey() + "\": " +
entry.getValue());
        }
    }

    // Esempio di utilizzo
    public static void main(String[] args) {
        nRicorrenze("programmazione java");
        System.out.println();
        nRicorrenzeParole("Java è un linguaggio di programmazione. Java è
molto usato.");
    }
}
```

Esempi di TreeMap e TreeSet

```java
import java.util.TreeMap;
import java.util.TreeSet;

public class EsempioTreeMapTreeSet {
    public static void main(String[] args) {
        // Esempio di TreeSet
        System.out.println("=== TreeSet ===");
        TreeSet<String> treeSet = new TreeSet<>();

        // Aggiunge elementi (saranno automaticamente ordinati)
        treeSet.add("Banana");
        treeSet.add("Mela");
        treeSet.add("Arancia");
        treeSet.add("Kiwi");
        treeSet.add("Mela");  // I duplicati non vengono aggiunti

        // Stampa il set (l'ordine sarà alfabetico)
        System.out.println("TreeSet ordinato: " + treeSet);

        // Operazioni specifiche di TreeSet
        System.out.println("Primo elemento: " + treeSet.first());
        System.out.println("Ultimo elemento: " + treeSet.last());
        System.out.println("Elementi prima di 'Kiwi': " +
treeSet.headSet("Kiwi"));
        System.out.println("Elementi dopo o uguali a 'Kiwi': " +
treeSet.tailSet("Kiwi"));

        // Esempio di TreeMap
        System.out.println("\n=== TreeMap ===");
        TreeMap<String, Integer> treeMap = new TreeMap<>();

        // Aggiunge coppie chiave-valore (le chiavi saranno ordinate
automaticamente)
        treeMap.put("Mela", 10);
        treeMap.put("Banana", 5);
        treeMap.put("Arancia", 8);
        treeMap.put("Kiwi", 12);

        // Stampa la mappa (le chiavi saranno in ordine alfabetico)
        System.out.println("TreeMap ordinata: " + treeMap);

        // Operazioni di base
        System.out.println("Valore per 'Mela': " + treeMap.get("Mela"));
        System.out.println("La mappa contiene 'Banana'? " +
treeMap.containsKey("Banana"));
        System.out.println("La mappa contiene il valore 15? " +
treeMap.containsValue(15));

        // Operazioni specifiche di TreeMap
```

```
        System.out.println("Prima chiave: " + treeMap.firstKey());
        System.out.println("Ultima chiave: " + treeMap.lastKey());
        System.out.println("Sottomap fino a 'Kiwi' (escluso): " +
treeMap.headMap("Kiwi"));
        System.out.println("Sottomap da 'Kiwi' in poi: " +
treeMap.tailMap("Kiwi"));

        // Iterazione sulle coppie chiave-valore
        System.out.println("\nInventario frutta:");
        for (java.util.Map.Entry<String, Integer> entry :
treeMap.entrySet()) {
            System.out.println("- " + entry.getKey() + ": " +
entry.getValue() + " pezzi");
        }
    }
}
```

Implementazione di una Tabella Hash Semplice

Ecco un'implementazione semplificata di una tabella hash:

```
/**
 * Implementazione di una tabella hash semplice, che usa il concatenamento
 * per la gestione delle collisioni
 */
public class SimpleHashTable<K, V> {
    private static class Entry<K, V> {
        K key;
        V value;
        Entry<K, V> next;

        Entry(K key, V value, Entry<K, V> next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }

    private static final int DEFAULT_CAPACITY = 16;
    private static final float DEFAULT_LOAD_FACTOR = 0.75f;

    private Entry<K, V>[] buckets;
    private int size;
    private float loadFactor;

    /**
     * Costruisce una tabella hash vuota con capacità e fattore di carico
predefiniti
```

```java
     */
    @SuppressWarnings("unchecked")
    public SimpleHashTable() {
        this.buckets = new Entry[DEFAULT_CAPACITY];
        this.size = 0;
        this.loadFactor = DEFAULT_LOAD_FACTOR;
    }

    /**
     * Costruisce una tabella hash vuota con capacità specificata
     */
    @SuppressWarnings("unchecked")
    public SimpleHashTable(int initialCapacity) {
        this.buckets = new Entry[initialCapacity];
        this.size = 0;
        this.loadFactor = DEFAULT_LOAD_FACTOR;
    }

    /**
     * Calcola l'indice del bucket per una chiave
     */
    private int hash(K key) {
        return key == null ? 0 : Math.abs(key.hashCode() % buckets.length);
    }

    /**
     * Inserisce o aggiorna una coppia chiave-valore
     * @return il valore precedente associato alla chiave, o null se non
c'era
     */
    public V put(K key, V value) {
        if (size >= loadFactor * buckets.length) {
            resize();
        }

        int index = hash(key);
        Entry<K, V> entry = buckets[index];

        // Cerca se la chiave esiste già
        while (entry != null) {
            if ((key == null && entry.key == null) ||
                (key != null && key.equals(entry.key))) {
                // Aggiorna il valore esistente
                V oldValue = entry.value;
                entry.value = value;
                return oldValue;
            }
            entry = entry.next;
        }
```

```
        // Aggiunge una nuova entry all'inizio della lista
        buckets[index] = new Entry<>(key, value, buckets[index]);
        size++;
        return null;
    }

    /**
     * Recupera il valore associato a una chiave
     * @return il valore associato alla chiave, o null se non trovato
     */
    public V get(K key) {
        int index = hash(key);
        Entry<K, V> entry = buckets[index];

        while (entry != null) {
            if ((key == null && entry.key == null) ||
                (key != null && key.equals(entry.key))) {
                return entry.value;
            }
            entry = entry.next;
        }

        return null;
    }

    /**
     * Rimuove una coppia chiave-valore dalla tabella
     * @return il valore associato alla chiave, o null se non trovato
     */
    public V remove(K key) {
        int index = hash(key);
        Entry<K, V> prev = null;
        Entry<K, V> curr = buckets[index];

        while (curr != null) {
            if ((key == null && curr.key == null) ||
                (key != null && key.equals(curr.key))) {
                // Rimuove l'entry
                if (prev == null) {
                    buckets[index] = curr.next;
                } else {
                    prev.next = curr.next;
                }
                size--;
                return curr.value;
            }
            prev = curr;
            curr = curr.next;
        }
```

```
        return null;
    }

    /**
     * Verifica se la tabella contiene una chiave
     */
    public boolean containsKey(K key) {
        return get(key) != null;
    }

    /**
     * Restituisce il numero di coppie chiave-valore
     */
    public int size() {
        return size;
    }

    /**
     * Verifica se la tabella è vuota
     */
    public boolean isEmpty() {
        return size == 0;
    }

    /**
     * Rimuove tutte le coppie chiave-valore
     */
    @SuppressWarnings("unchecked")
    public void clear() {
        this.buckets = new Entry[buckets.length];
        this.size = 0;
    }

    /**
     * Ridimensiona la tabella hash
     */
    @SuppressWarnings("unchecked")
    private void resize() {
        Entry<K, V>[] oldBuckets = buckets;
        buckets = new Entry[oldBuckets.length * 2];
        size = 0;

        // Inserisce tutte le entry nella nuova tabella
        for (Entry<K, V> bucket : oldBuckets) {
            Entry<K, V> entry = bucket;
            while (entry != null) {
                put(entry.key, entry.value);
                entry = entry.next;
            }
        }
```

```
    }

    /**
     * Restituisce una rappresentazione in stringa della tabella
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("{");
        boolean first = true;

        for (Entry<K, V> bucket : buckets) {
            Entry<K, V> entry = bucket;
            while (entry != null) {
                if (!first) {
                    sb.append(", ");
                }
                sb.append(entry.key).append("=").append(entry.value);
                first = false;
                entry = entry.next;
            }
        }

        sb.append("}");
        return sb.toString();
    }

    /**
     * Esempio di utilizzo
     */
    public static void main(String[] args) {
        SimpleHashTable<String, Integer> table = new SimpleHashTable<>();

        table.put("uno", 1);
        table.put("due", 2);
        table.put("tre", 3);

        System.out.println("Tabella: " + table);
        System.out.println("Valore di 'due': " + table.get("due"));

        table.put("due", 22);  // Aggiorna un valore
        System.out.println("Tabella dopo aggiornamento: " + table);

        table.remove("uno");
        System.out.println("Tabella dopo rimozione: " + table);

        System.out.println("La tabella contiene 'tre'? " +
table.containsKey("tre"));
        System.out.println("Dimensione della tabella: " + table.size());
```

```
    }
}
```

Esercizi Aggiuntivi

Esercizio 1: Verifica Parentesi Bilanciate

```
import java.util.Stack;

public class VerificaParentesi {
    /**
     * Verifica se una stringa ha parentesi bilanciate
     * @param str la stringa da verificare
     * @return true se le parentesi sono bilanciate, false altrimenti
     */
    public static boolean parentesiBilanciate(String str) {
        if (str == null || str.isEmpty()) {
            return true;
        }

        Stack<Character> stack = new Stack<>();

        for (char c : str.toCharArray()) {
            if (c == '(' || c == '[' || c == '{') {
                stack.push(c);
            } else if (c == ')' || c == ']' || c == '}') {
                if (stack.isEmpty()) {
                    return false;
                }

                char top = stack.pop();

                if ((c == ')' && top != '(') ||
                    (c == ']' && top != '[') ||
                    (c == '}' && top != '{')) {
                    return false;
                }
            }
        }

        return stack.isEmpty();
    }

    // Test
    public static void main(String[] args) {
        String[] tests = {
            "(((())))",
            "{[()]}",
```

```
            "({[]})",
            "([)]",
            "(((",
            ")))",
            "{[(])}"
        };

        for (String test : tests) {
            System.out.println("'" + test + "' è bilanciata? " +
parentesiBilanciate(test));
        }
    }
}
```

Esercizio 2: Implementazione di una Coda con Due Stack

```
import java.util.Stack;
import java.util.NoSuchElementException;

public class CodaConDueStack<T> implements Coda<T> {
    private Stack<T> stackPush;  // Per enqueue
    private Stack<T> stackPop;   // Per dequeue

    /**
     * Costruisce una coda vuota implementata con due stack
     */
    public CodaConDueStack() {
        stackPush = new Stack<>();
        stackPop = new Stack<>();
    }

    /**
     * Verifica se la coda è vuota
     */
    @Override
    public boolean isEmpty() {
        return stackPush.isEmpty() && stackPop.isEmpty();
    }

    /**
     * Restituisce il numero di elementi nella coda
     */
    @Override
    public int size() {
        return stackPush.size() + stackPop.size();
    }

    /**
```

```
     * Aggiunge un elemento alla fine della coda
     */
    @Override
    public void enqueue(T elem) {
        stackPush.push(elem);
    }

    /**
     * Rimuove e restituisce l'elemento frontale
     */
    @Override
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("La coda è vuota");
        }

        // Se lo stack per la rimozione è vuoto, trasferisci tutti gli
elementi
        // dallo stack di inserimento
        if (stackPop.isEmpty()) {
            trasferisciElementi();
        }

        return stackPop.pop();
    }

    /**
     * Restituisce l'elemento frontale senza rimuoverlo
     */
    @Override
    public T getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("La coda è vuota");
        }

        // Se lo stack per la rimozione è vuoto, trasferisci tutti gli
elementi
        // dallo stack di inserimento
        if (stackPop.isEmpty()) {
            trasferisciElementi();
        }

        return stackPop.peek();
    }

    /**
     * Trasferisce tutti gli elementi dallo stack di inserimento allo stack
di rimozione
     */
    private void trasferisciElementi() {
```

```
        while (!stackPush.isEmpty()) {
            stackPop.push(stackPush.pop());
        }
    }

    /**
     * Restituisce una rappresentazione in stringa di questa coda
     */
    @Override
    public String toString() {
        if (isEmpty()) {
            return "Coda: []";
        }

        // Crea copie temporanee degli stack per non modificare la coda
        Stack<T> tempPush = new Stack<>();
        tempPush.addAll(stackPush);

        Stack<T> tempPop = new Stack<>();
        tempPop.addAll(stackPop);

        StringBuilder sb = new StringBuilder();
        sb.append("Coda: [");

        // Prima gli elementi dello stack di rimozione (sono in ordine LIFO
rispetto alla coda)
        boolean first = true;
        while (!tempPop.isEmpty()) {
            if (!first) {
                sb.append(", ");
            }
            sb.append(tempPop.pop());
            first = false;
        }

        // Poi gli elementi dello stack di inserimento (bisogna invertirli)
        Stack<T> reversed = new Stack<>();
        while (!tempPush.isEmpty()) {
            reversed.push(tempPush.pop());
        }

        while (!reversed.isEmpty()) {
            if (!first) {
                sb.append(", ");
            }
            sb.append(reversed.pop());
            first = false;
        }

        sb.append("]");
```

```
        return sb.toString();
    }

    // Test
    public static void main(String[] args) {
        CodaConDueStack<Integer> coda = new CodaConDueStack<>();

        // Aggiunge elementi
        coda.enqueue(1);
        coda.enqueue(2);
        coda.enqueue(3);

        System.out.println(coda);
        System.out.println("Elemento in testa: " + coda.getFront());

        // Rimuove un elemento
        System.out.println("Rimuovo: " + coda.dequeue());
        System.out.println(coda);

        // Aggiunge altri elementi
        coda.enqueue(4);
        coda.enqueue(5);
        System.out.println(coda);

        // Svuota la coda
        while (!coda.isEmpty()) {
            System.out.println("Rimuovo: " + coda.dequeue());
        }

        System.out.println("La coda è vuota? " + coda.isEmpty());
    }
}
```

Esercizio 3: Implementazione di un Set Utilizzando HashMap

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * Implementazione di un Set utilizzando HashMap
 */
public class HashMapSet<E> implements Iterable<E> {
    private HashMap<E, Object> map;
    private static final Object DUMMY = new Object();

    /**
     * Costruisce un set vuoto
```

```
     */
    public HashMapSet() {
        map = new HashMap<>();
    }

    /**
     * Aggiunge un elemento al set
     * @return true se l'elemento è stato aggiunto, false se era già
presente
     */
    public boolean add(E e) {
        return map.put(e, DUMMY) == null;
    }

    /**
     * Rimuove un elemento dal set
     * @return true se l'elemento è stato rimosso, false se non era presente
     */
    public boolean remove(Object o) {
        return map.remove(o) != null;
    }

    /**
     * Verifica se un elemento è presente nel set
     */
    public boolean contains(Object o) {
        return map.containsKey(o);
    }

    /**
     * Restituisce il numero di elementi nel set
     */
    public int size() {
        return map.size();
    }

    /**
     * Verifica se il set è vuoto
     */
    public boolean isEmpty() {
        return map.isEmpty();
    }

    /**
     * Rimuove tutti gli elementi dal set
     */
    public void clear() {
        map.clear();
    }
```

```java
    /**
     * Restituisce un iteratore per gli elementi del set
     */
    @Override
    public Iterator<E> iterator() {
        return map.keySet().iterator();
    }

    /**
     * Restituisce una rappresentazione in stringa del set
     */
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");

        boolean first = true;
        for (E e : this) {
            if (!first) {
                sb.append(", ");
            }
            sb.append(e);
            first = false;
        }

        sb.append("]");
        return sb.toString();
    }

    // Test
    public static void main(String[] args) {
        HashMapSet<String> set = new HashMapSet<>();

        set.add("Mela");
        set.add("Banana");
        set.add("Arancia");
        set.add("Mela");  // Non verrà aggiunto perché è un duplicato

        System.out.println("Set: " + set);
        System.out.println("Dimensione: " + set.size());
        System.out.println("Contiene 'Banana'? " + set.contains("Banana"));

        set.remove("Banana");
        System.out.println("Set dopo la rimozione: " + set);

        System.out.println("Elementi nel set:");
        for (String frutto : set) {
            System.out.println("- " + frutto);
        }
```

```
    }
}
```

Esercizio 4: Unione di Due Liste Ordinate

```
import java.util.LinkedList;
import java.util.List;

public class UnioneListe {
    /**
     * Unisce due liste ordinate in una nuova lista ordinata
     * @param lista1 prima lista ordinata
     * @param lista2 seconda lista ordinata
     * @return nuova lista contenente tutti gli elementi di entrambe le
liste, in ordine
     */
    public static <T extends Comparable<T>> List<T>
unisciListeOrdinate(List<T> lista1, List<T> lista2) {
        List<T> risultato = new LinkedList<>();

        int i = 0, j = 0;

        // Finché entrambe le liste hanno elementi
        while (i < lista1.size() && j < lista2.size()) {
            T elem1 = lista1.get(i);
            T elem2 = lista2.get(j);

            if (elem1.compareTo(elem2) <= 0) {
                risultato.add(elem1);
                i++;
            } else {
                risultato.add(elem2);
                j++;
            }
        }

        // Aggiungi i rimanenti elementi della prima lista
        while (i < lista1.size()) {
            risultato.add(lista1.get(i));
            i++;
        }

        // Aggiungi i rimanenti elementi della seconda lista
        while (j < lista2.size()) {
            risultato.add(lista2.get(j));
            j++;
        }
```

```
        return risultato;
    }

    // Test
    public static void main(String[] args) {
        List<Integer> lista1 = new LinkedList<>();
        lista1.add(1);
        lista1.add(3);
        lista1.add(5);
        lista1.add(7);

        List<Integer> lista2 = new LinkedList<>();
        lista2.add(2);
        lista2.add(4);
        lista2.add(6);
        lista2.add(8);

        List<Integer> unione = unisciListeOrdinate(lista1, lista2);
        System.out.println("Lista 1: " + lista1);
        System.out.println("Lista 2: " + lista2);
        System.out.println("Unione: " + unione);

        // Test con liste di lunghezza diversa
        lista1.add(9);
        lista1.add(11);

        unione = unisciListeOrdinate(lista1, lista2);
        System.out.println("\nLista 1 modificata: " + lista1);
        System.out.println("Lista 2: " + lista2);
        System.out.println("Unione: " + unione);
    }
}
```

Esercizio 5: Implementazione di una Coda a Priorità con ArrayList

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.NoSuchElementException;

/**
 * Implementazione di una coda a priorità utilizzando ArrayList
 */
public class PriorityQueueArrayList<T> {
    private ArrayList<T> elements;
    private Comparator<? super T> comparator;

    /**
```

```
     * Costruisce una coda a priorità vuota con l'ordinamento naturale degli
elementi
     */
    public PriorityQueueArrayList() {
        this(null);
    }

    /**
     * Costruisce una coda a priorità vuota con l'ordinamento specificato
dal comparatore
     */
    public PriorityQueueArrayList(Comparator<? super T> comparator) {
        this.elements = new ArrayList<>();
        this.comparator = comparator;
    }

    /**
     * Confronta due elementi usando il comparatore o l'ordinamento naturale
     */
    @SuppressWarnings("unchecked")
    private int compare(T o1, T o2) {
        if (comparator != null) {
            return comparator.compare(o1, o2);
        } else {
            return ((Comparable<? super T>) o1).compareTo(o2);
        }
    }

    /**
     * Verifica se la coda è vuota
     */
    public boolean isEmpty() {
        return elements.isEmpty();
    }

    /**
     * Restituisce il numero di elementi nella coda
     */
    public int size() {
        return elements.size();
    }

    /**
     * Aggiunge un elemento alla coda
     */
    public void offer(T elem) {
        if (elem == null) {
            throw new NullPointerException();
        }
```

```
        elements.add(elem);

        // Mantiene l'ordine ristabilendo la proprietà di heap
        int current = elements.size() - 1;
        int parent = (current - 1) / 2;

        while (current > 0 && compare(elements.get(current),
elements.get(parent)) < 0) {
            // Scambia l'elemento corrente con il genitore
            T temp = elements.get(current);
            elements.set(current, elements.get(parent));
            elements.set(parent, temp);

            current = parent;
            parent = (current - 1) / 2;
        }
    }

    /**
     * Restituisce ma non rimuove l'elemento con la più alta priorità
     */
    public T peek() {
        if (isEmpty()) {
            return null;
        }
        return elements.get(0);
    }

    /**
     * Restituisce e rimuove l'elemento con la più alta priorità
     */
    public T poll() {
        if (isEmpty()) {
            return null;
        }

        T result = elements.get(0);

        // Sostituisce la radice con l'ultimo elemento
        T lastElement = elements.remove(elements.size() - 1);

        if (!elements.isEmpty()) {
            elements.set(0, lastElement);
            heapify(0);
        }

        return result;
    }

    /**
```

```
     * Ristabilisce la proprietà di heap a partire dall'indice specificato
     */
    private void heapify(int index) {
        int size = elements.size();
        int smallest = index;
        int left = 2 * index + 1;
        int right = 2 * index + 2;

        // Trova il più piccolo tra l'elemento corrente e i suoi figli
        if (left < size && compare(elements.get(left),
elements.get(smallest)) < 0) {
            smallest = left;
        }

        if (right < size && compare(elements.get(right),
elements.get(smallest)) < 0) {
            smallest = right;
        }

        // Se uno dei figli è più piccolo, scambia e continua il processo
        if (smallest != index) {
            T temp = elements.get(index);
            elements.set(index, elements.get(smallest));
            elements.set(smallest, temp);

            heapify(smallest);
        }
    }

    /**
     * Restituisce e rimuove l'elemento con la più alta priorità
     * @throws NoSuchElementException se la coda è vuota
     */
    public T remove() {
        T element = poll();
        if (element == null) {
            throw new NoSuchElementException();
        }
        return element;
    }

    /**
     * Restituisce una rappresentazione in stringa della coda
     */
    @Override
    public String toString() {
        return elements.toString();
    }

    // Test
```

```
    public static void main(String[] args) {
        // Coda a priorità con ordinamento naturale (minore ha priorità più
alta)
        PriorityQueueArrayList<Integer> pq = new PriorityQueueArrayList<>();

        pq.offer(5);
        pq.offer(2);
        pq.offer(8);
        pq.offer(1);
        pq.offer(10);

        System.out.println("Coda a priorità: " + pq);
        System.out.println("Elemento con priorità più alta: " + pq.peek());

        System.out.println("\nEstrazione degli elementi in ordine di
priorità:");
        while (!pq.isEmpty()) {
            System.out.println(pq.poll());
        }

        // Coda a priorità con ordinamento personalizzato (maggiore ha
priorità più alta)
        Comparator<Integer> reverseOrder = (a, b) -> b.compareTo(a);
        PriorityQueueArrayList<Integer> pqReverse = new
PriorityQueueArrayList<>(reverseOrder);

        pqReverse.offer(5);
        pqReverse.offer(2);
        pqReverse.offer(8);
        pqReverse.offer(1);
        pqReverse.offer(10);

        System.out.println("\nCoda a priorità (ordine inverso): " +
pqReverse);
        System.out.println("Elemento con priorità più alta: " +
pqReverse.peek());

        System.out.println("\nEstrazione degli elementi in ordine di
priorità (inverso):");
        while (!pqReverse.isEmpty()) {
            System.out.println(pqReverse.poll());
        }
    }
}
```