Riassunto completo

Indice

Introduzione

Le strutture dati sono formati specializzati per organizzare, elaborare, recuperare e memorizzare dati in modo efficiente. In Java, molte di queste strutture sono parte del Collections Framework, che fornisce implementazioni riutilizzabili dei principali tipi di dati astratti.

Interfacce di Base

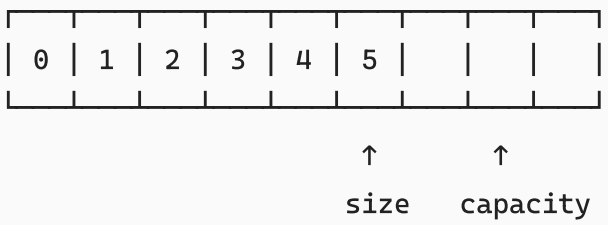
```
// Interfaccia base per tutte le strutture dati container
public interface Contenitore {
    boolean isEmpty();  // Verifica se il contenitore è vuoto
    int size();         // Restituisce il numero di elementi
}

// Interfaccia per le pile (LIFO)
public interface Pila<T> extends Contenitore {
    void push(T elem);  // Aggiunge elemento in cima
    T pop();            // Rimuove e restituisce elemento in cima
    T top();            // Visualizza elemento in cima senza rimuoverlo
}

// Interfaccia per le code (FIFO)
public interface Coda<T> extends Contenitore {
```

```
    void enqueue(T elem);  // Aggiunge elemento in coda
    T dequeue();           // Rimuove e restituisce elemento frontale
    T getFront();          // Visualizza elemento frontale senza rimuoverlo
}
```

ArrayList

ArrayList è un array ridimensionabile che può crescere o diminuire dinamicamente a seconda delle necessità.

Schema di Funzionamento



Implementazione di ArrayListInteger

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

    // Costruttore
    public ArrayListInteger() {
        elements = new int[DEFAULT_CAPACITY];
        size = 0;
    }

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

    // Ottiene l'elemento all'indice specificato
    public int get(int index) {
        if (index >= size || index < 0)
            throw new IndexOutOfBoundsException();
        return elements[index];
    }

    // Rimuove l'elemento all'indice specificato
    public int remove(int index) {
```

```
        if (index >= size || index < 0)
            throw new IndexOutOfBoundsException();
        int oldValue = elements[index];
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elements, index + 1, elements, index,
numMoved);
        size--;
        return oldValue;
    }

    // Aumenta la capacità se necessario
    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;
        }
    }

    // Altri metodi (size, isEmpty, ecc.)
}
```

Complessità Computazionale

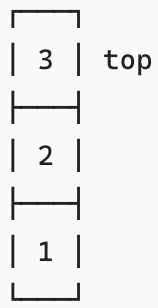
Operazione	Complessità	Note
add(e)	O(1) ammortizzato	Può essere O(n) se è necessario ridimensionare l'array
get(index)	O(1)	Accesso diretto tramite indice
remove(index)	O(n)	Richiede spostamento degli elementi successivi
size()	O(1)	Restituisce il campo size
contains(obj)	O(n)	Richiede scansione dell'array

Pile (LIFO)

Le pile (Stack) seguono il principio Last-In-First-Out (LIFO): l'ultimo elemento inserito sarà il primo ad essere rimosso.

Schema di Funzionamento

```
       push ↓   ↑ pop
            |   |
            ▼   |
```



Implementazione di PilaArrayList

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

    // Costruttore
    public PilaArrayList() {
        elements = new ArrayList<>();
    }

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

    // Restituisce la dimensione della pila
    @Override
    public int size() {
        return elements.size();
    }

    // Aggiunge un elemento in cima
    @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);
    }

    // Visualizza l'elemento in cima senza rimuoverlo
    @Override
    public T top() {
```

```
        if (isEmpty())
            throw new EmptyStackException();
        return elements.get(elements.size() - 1);
    }
}
```

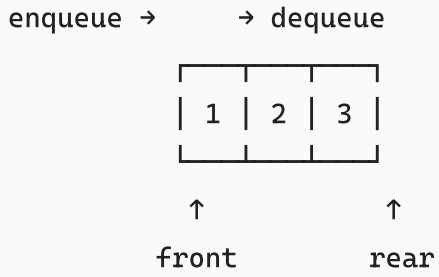
Complessità Computazionale

Operazione	Complessità	Note
push(e)	O(1) ammortizzato	Può essere O(n) se ArrayList deve ridimensionarsi
pop()	O(1)	Rimozione dall'ultimo elemento
top()	O(1)	Accesso all'ultimo elemento
isEmpty()	O(1)	Verifica se è vuota
size()	O(1)	Restituisce la dimensione

Code (FIFO)

Le code (Queue) seguono il principio First-In-First-Out (FIFO): il primo elemento inserito sarà il primo ad essere rimosso.

Schema di Funzionamento



Implementazione di CodaArrayList

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

    // Costruttore
    public CodaArrayList() {
        elements = new ArrayList<>();
    }

    // Verifica se la coda è vuota
    @Override
    public boolean isEmpty() {
```

```
        return elements.isEmpty();
    }

    // Restituisce la dimensione della 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);
    }

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

Implementazione di CodaArrayListEstremoFissoLiberato

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

    // Costruttore
    public CodaArrayListEstremoFissoLiberato() {
        elements = new ArrayList<>();
        front = 0;
    }

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

```java
    }

    // 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 "liberate"
        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;
    }

    // Visualizza 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 le posizioni liberate
    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;
    }
}
```

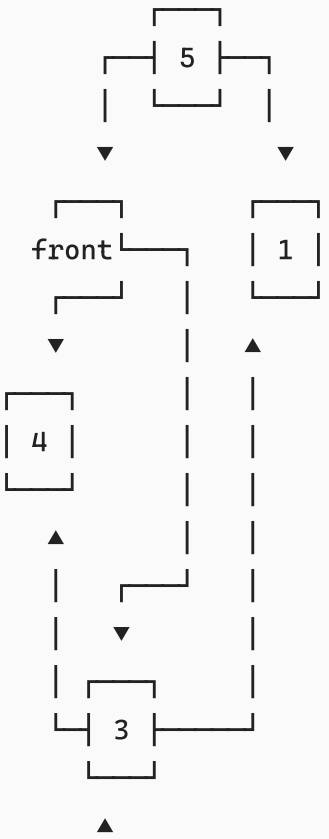
Complessità Computazionale

Operazione	CodaArrayList	CodaArrayListEstremoFissoLiberato	Note
enqueue(e)	O(1) ammortizzato	O(1) ammortizzato	Può essere O(n) se necessario ridimensionare
dequeue()	O(n)	O(1)	In CodaArrayList richiede spostamento di tutti gli elementi
getFront()	O(1)	O(1)	
isEmpty()	O(1)	O(1)	
size()	O(1)	O(1)	

Coda Circolare

Una coda circolare usa un array di dimensione fissa in modo più efficiente, "avvolgendosi" all'inizio quando raggiunge la fine.

Schema di Funzionamento



```
          |
        rear
```

Implementazione di CodaCircolareArrayInteger

```
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;

    // Costruttore
    public CodaCircolareArrayInteger() {
        elements = new int[DEFAULT_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;
    }

    // Restituisce il numero di elementi
    @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;
    }

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

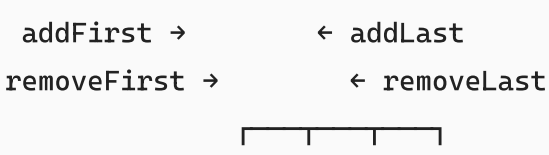
Complessità Computazionale

Operazione	Complessità	Note
enqueue(e)	O(1)	Operazione costante
dequeue()	O(1)	Operazione costante
getFront()	O(1)	Operazione costante
isEmpty()	O(1)	Operazione costante
isFull()	O(1)	Operazione costante
size()	O(1)	Operazione costante

Coda Doppia (Deque)

Una Deque (Double-ended Queue) permette di inserire e rimuovere elementi da entrambe le estremità.

Schema di Funzionamento



```
| 1 | 2 | 3 |
└───┴───┴───┘
```

Implementazione del metodo isPalindrome usando Deque

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

public class PalindromeChecker {
    /**
     * Verifica se una stringa è un palindromo utilizzando un Deque
     */
    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;
    }
}
```

Complessità Computazionale

Operazione	Complessità (ArrayDeque)	Note
addFirst(e)	O(1)	Operazione costante
addLast(e)	O(1)	Operazione costante
removeFirst()	O(1)	Operazione costante
removeLast()	O(1)	Operazione costante
getFirst()	O(1)	Operazione costante
getLast()	O(1)	Operazione costante
size()	O(1)	Operazione costante

Coda Prioritaria

Una coda prioritaria è un tipo di coda dove gli elementi hanno una "priorità" e vengono serviti in base ad essa anziché in ordine FIFO.

Schema di Funzionamento

```
Coda (minimo in testa):

          1
        /   \
       2     3
      / \
     4   5

Inserimento di un elemento: O(log n)
Estrazione del minimo: O(log n)
```

Implementazione di Coppia e ComparatoreCoppia per PriorityQueue

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

class Coppia {
    private String descrizione;
    private int priorita;

    public Coppia(String descrizione, int priorita) {
        this.descrizione = descrizione;
```

```
        this.priorita = priorita;
    }

    public String getDescrizione() {
        return descrizione;
    }

    public int getPriorita() {
        return priorita;
    }

    @Override
    public String toString() {
        return "(" + descrizione + ", " + priorita + ")";
    }
}

class ComparatoreCoppia implements Comparator<Coppia> {
    // Confronta due coppie in base alla priorità
    // Priorità minore = priorità più alta (min-heap)
    @Override
    public int compare(Coppia o1, Coppia o2) {
        return o1.getPriorita() - o2.getPriorita();
    }
}

class EsempioPriorityQueue {
    public static void main(String[] args) {
        // Crea una coda prioritaria con 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());
        }
    }
}
```

Complessità Computazionale

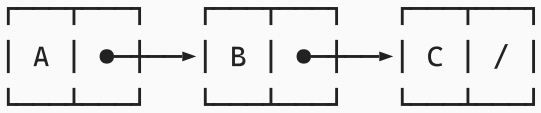
Operazione	Complessità	Note
add(e) / offer(e)	O(log n)	Richiede riequilibrio dell'heap
poll() / remove()	O(log n)	Richiede riequilibrio dell'heap
peek()	O(1)	Accede solo all'elemento in testa
contains(o)	O(n)	Richiede ricerca nell'heap
size()	O(1)	Operazione costante

Liste Concatenate

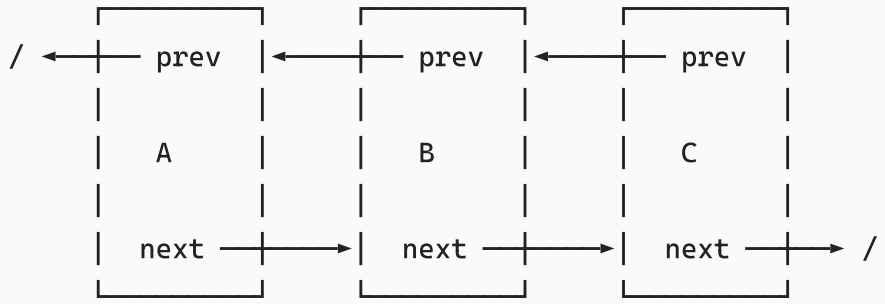
Una lista concatenata è una sequenza di nodi, dove ogni nodo contiene dati e un riferimento al nodo successivo.

Schema di Funzionamento

```
Lista semplicemente concatenata:
```




```
Lista doppiamente concatenata:
```



Esempio di Utilizzo di LinkedList

```java
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);

        // Itera sulla lista
        System.out.println("Elementi nella lista dopo le rimozioni:");
        for (String elemento : lista) {
            System.out.println("- " + elemento);
        }
    }
}
```

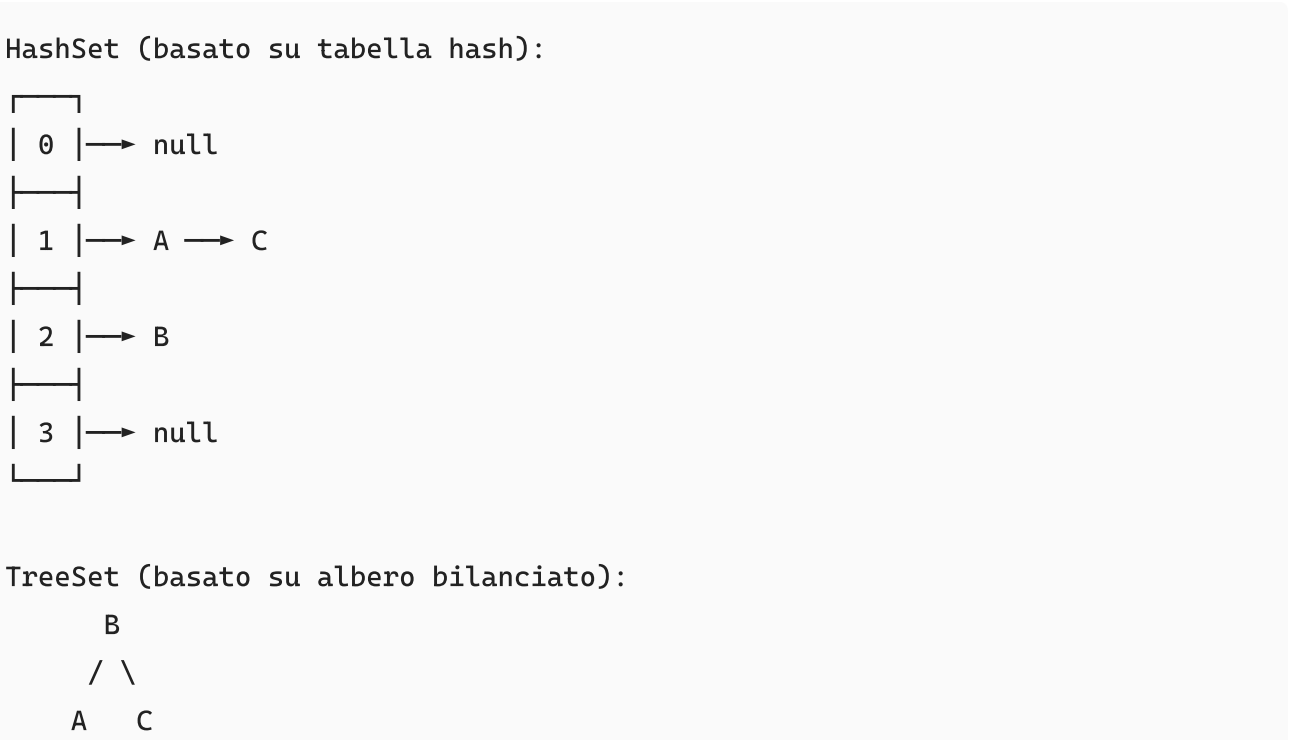
Complessità Computazionale

Operazione	Complessità	Note
add(e) / addLast(e)	O(1)	Operazione costante
addFirst(e)	O(1)	Operazione costante
add(index, e)	O(n)	Richiede navigazione fino all'indice
get(index)	O(n)	Richiede navigazione fino all'indice
getFirst() / getLast()	O(1)	Operazione costante
remove(index)	O(n)	Richiede navigazione fino all'indice
removeFirst() / removeLast()	O(1)	Operazione costante
contains(o)	O(n)	Richiede scansione della lista

Insiemi (Set)

Un Set è una collezione che non contiene elementi duplicati.

Schema di Funzionamento



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

        // Rimuove un elemento
        set.remove("Banana");

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

```
        // Operazioni sugli insiemi

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

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

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

Complessità Computazionale

Operazione	HashSet	TreeSet	Note
add(e)	O(1)	O(log n)	HashSet usa tabella hash, TreeSet usa albero RB
remove(o)	O(1)	O(log n)	
contains(o)	O(1)	O(log n)	
size()	O(1)	O(1)	
iteration	O(n)	O(n)	TreeSet itera in ordine

Mappe

Una Map è un oggetto che associa chiavi a valori, senza chiavi duplicate.

Schema di Funzionamento

```
| 3 |──► null
└───┘


TreeMap:
     K2:V2
    /    \
K1:V1    K3:V3
```

Esempio di nRicorrenze usando HashMap

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

public class ContaRicorrenze {
    /**
     * Conta e stampa le ricorrenze di ogni carattere in una stringa
     */
    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());
        }
    }

    public static void main(String[] args) {
        nRicorrenze("programmazione");
    }
}
```

Complessità Computazionale

Operazione	HashMap	TreeMap	Note
put(k, v)	O(1)	O(log n)	
get(k)	O(1)	O(log n)	
remove(k)	O(1)	O(log n)	
containsKey(k)	O(1)	O(log n)	
containsValue(v)	O(n)	O(n)	Richiede scansione di tutti i valori
size()	O(1)	O(1)	
iteration	O(n)	O(n)	TreeMap itera in ordine di chiave

Tabelle di Hash

Una tabella hash è una struttura dati che associa chiavi a valori utilizzando una funzione di hash.

Schema di Funzionamento

```
Tabella hash con risoluzione delle collisioni per concatenamento:

hash(k1) = hash(k3) = 1
hash(k2) = 2

┌─────┐
│  0  │──→ null
├─────┤
│  1  │──→ [k1:v1] ──→ [k3:v3]
├─────┤
│  2  │──→ [k2:v2]
├─────┤
│  3  │──→ null
└─────┘
```

Tabelle di Hash (continuazione)

Complessità Computazionale

Operazione	Caso medio	Caso peggiore	Note
insert(k, v)	O(1)	O(n)	Dipende dalla funzione hash e dal fattore di carico

Operazione	Caso medio	Caso peggiore	Note
search(k)	O(1)	O(n)	Dipende dalla funzione hash e dal fattore di carico
delete(k)	O(1)	O(n)	Dipende dalla funzione hash e dal fattore di carico

Implementazione Semplificata di una Tabella Hash

```
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;

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

    private int hash(K key) {
        return key == null ? 0 : Math.abs(key.hashCode() % buckets.length);
    }

    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;
    }

    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;
    }

    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;
    }

    public boolean containsKey(K key) {
        return get(key) != null;
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    @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;
            }
        }
    }
}
```

Esercizi aggiuntivi

Verifica Parentesi Bilanciate

```
import java.util.Stack;

public class VerificaParentesi {
    /**
     * Verifica se una stringa ha parentesi bilanciate
     */
    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();
    }
}
```

Coda implementata 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

    public CodaConDueStack() {
        stackPush = new Stack<>();
        stackPop = new Stack<>();
    }

    @Override
    public boolean isEmpty() {
        return stackPush.isEmpty() && stackPop.isEmpty();
    }

    @Override
    public int size() {
        return stackPush.size() + stackPop.size();
    }
```

```
    @Override
    public void enqueue(T elem) {
        stackPush.push(elem);
    }

    @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
        if (stackPop.isEmpty()) {
            trasferisciElementi();
        }

        return stackPop.pop();
    }

    @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
        if (stackPop.isEmpty()) {
            trasferisciElementi();
        }

        return stackPop.peek();
    }

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

Riepilogo Complessità Computazionale

Di seguito, una tabella riassuntiva delle complessità computazionali per le principali operazioni delle strutture dati trattate:

Struttura Dati	Accesso	Ricerca	Inserimento	Cancellazione
ArrayList	O(1)	O(n)	O(1)/O(n)	O(n)
Stack (ArrayList)	O(1)	O(n)	O(1)	O(1)
Queue (LinkedList)	O(1)	O(n)	O(1)	O(1)
CodaArrayList	O(1)	O(n)	O(1)	O(n)
CodaArrayListEstremoFissoLiberato	O(1)	O(n)	O(1)	O(1)
Circular Queue	O(1)	O(n)	O(1)	O(1)
Deque (ArrayDeque)	O(1)	O(n)	O(1)	O(1)
Priority Queue	O(1)*	O(n)	O(log n)	O(log n)
LinkedList	O(n)	O(n)	O(1)**	O(1)**
HashSet	N/A	O(1)	O(1)	O(1)
TreeSet	N/A	O(log n)	O(log n)	O(log n)
HashMap	N/A	O(1)	O(1)	O(1)
TreeMap	N/A	O(log n)	O(log n)	O(log n)
Hash Table	N/A	O(1)	O(1)	O(1)

- Solo per l'elemento con priorità più alta
  ** Quando la posizione è già nota

Linee guida per la scelta della struttura dati

Quando si seleziona una struttura dati, è importante considerare:

1. **Tipo di operazioni**: Quali operazioni verranno eseguite più frequentemente?
   - Inserimento/cancellazione frequente: LinkedList
   - Accesso casuale frequente: ArrayList
   - LIFO: Stack
   - FIFO: Queue
   - Operazioni su entrambe le estremità: Deque
   - Elementi ordinati per priorità: PriorityQueue
2. **Complessità temporale**: Quali caratteristiche di prestazioni sono richieste?
   - Ricerca veloce: HashSet, HashMap
   - Elementi ordinati: TreeSet, TreeMap
   - Accesso indicizzato veloce: ArrayList
3. **Vincoli di memoria**: Quanta memoria è disponibile?
   - Dimensione fissa: array, code circolari
   - Dimensione dinamica: ArrayList, LinkedList
4. **Requisiti di ordinamento**:

- Mantenere l'ordine di inserimento: LinkedList, LinkedHashMap
- Ordinamento naturale o personalizzato: TreeSet, TreeMap, PriorityQueue

5. **Caratteristiche specifiche**:
   - Eliminazione dei duplicati: Set
   - Mappature chiave-valore: Map
   - Equilibrio tra complessità e spazio: scegliere in base alle esigenze