Introduzione

Le strutture dati sono formati specializzati per organizzare, elaborare, recuperare e memorizzare dati in modo efficiente. Java fornisce una ricca collezione di strutture dati attraverso il suo Collections Framework, che include implementazioni per vari tipi di dati astratti.

Questo documento copre le principali strutture dati utilizzate nella programmazione Java, con un focus su:

- Fondamenti teorici
- Operazioni comuni
- Complessità computazionale
- Casi d'uso

1. ArrayList

Fondamenti Teorici

Un ArrayList è un'implementazione di array ridimensionabile che fornisce capacità di ridimensionamento dinamico mantenendo prestazioni simili a un array per la maggior parte delle operazioni.

Caratteristiche Principali

- Ridimensionamento dinamico (cresce automaticamente quando vengono aggiunti elementi)
- Accesso indicizzato (complessità temporale O(1))
- Implementa l'interfaccia List
- Consente elementi duplicati
- Consente elementi null
- Mantiene l'ordine di inserimento

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`add(E e)`	O(1) ammortizzato	Aggiunge elemento alla fine
`add(int index, E e)`	O(n)	Inserisce elemento a indice specifico
`get(int index)`	O(1)	Recupera elemento a indice specifico

Operazione	Complessità Temporale	Descrizione
`remove(int index)`	O(n)	Rimuove elemento a indice specifico
`remove(Object o)`	O(n)	Rimuove prima occorrenza dell'elemento
`size()`	O(1)	Ottiene il numero di elementi
`contains(Object o)`	O(n)	Verifica se l'elemento esiste
`isEmpty()`	O(1)	Verifica se la struttura è vuota

Dettagli Implementativi

- Basato su un array Java standard
- Quando la capacità viene raggiunta, tipicamente crea un nuovo array 1,5 volte più grande dell'originale
- L'allocazione sequenziale della memoria consente un uso efficiente della cache CPU

Casi d'Uso

- Quando è richiesto un accesso frequente per indice
- Quando la dimensione della collezione cambia frequentemente ma l'accesso casuale è ancora importante
- Per implementare semplici pile e code (sebbene siano preferibili classi specializzate)

2. Stack (LIFO)

Fondamenti Teorici

Uno Stack è una struttura dati lineare che segue il principio Last-In-First-Out (LIFO). Gli elementi vengono aggiunti e rimossi dalla stessa estremità, chiamata "cima" dello stack.

Caratteristiche Principali

- Pattern di accesso LIFO
- Gli elementi vengono aggiunti/rimossi solo da un'estremità
- Tipicamente implementato estendendo altre collezioni (ArrayList o LinkedList)

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`push(E e)`	O(1)	Aggiunge elemento in cima
`pop()`	O(1)	Rimuove e restituisce l'elemento in cima

Operazione	Complessità Temporale	Descrizione
`peek()`	O(1)	Visualizza l'elemento in cima senza rimuoverlo
`isEmpty()`	O(1)	Verifica se lo stack è vuoto
`size()`	O(1)	Ottiene il numero di elementi

Dettagli Implementativi

In Java, uno stack può essere implementato in diversi modi:

- Utilizzando la classe legacy `java.util.Stack` (estende Vector, sincronizzata)
- Utilizzando ArrayDeque (approccio moderno preferito)
- Implementazione personalizzata con ArrayList o LinkedList

Casi d'Uso

- Valutazione delle espressioni e parsing sintattico
- Algoritmi di backtracking
- Meccanismi di annullamento nelle applicazioni
- Gestione delle chiamate di funzione (call stack)

3. Queue (FIFO)

Fondamenti Teorici

Una Queue è una struttura dati lineare che segue il principio First-In-First-Out (FIFO). Gli elementi vengono aggiunti alla fine e rimossi dal fronte.

Caratteristiche Principali

- Pattern di accesso FIFO
- Gli elementi vengono aggiunti da un'estremità (coda/retro) e rimossi dall'altra (fronte/testa)
- Esistono varie implementazioni specializzate (code prioritarie, deque)

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`enqueue(E e)` / `add(E)`	O(1)	Aggiunge elemento alla fine
`dequeue()` / `remove()`	O(1)	Rimuove e restituisce l'elemento frontale

Operazione	Complessità Temporale	Descrizione
`peek()`	O(1)	Visualizza l'elemento frontale senza rimuoverlo
`isEmpty()`	O(1)	Verifica se la coda è vuota
`size()`	O(1)	Ottiene il numero di elementi

Dettagli Implementativi

In Java, le code possono essere implementate in diversi modi:

- Utilizzando `java.util.LinkedList` (implementa l'interfaccia Queue)
- Utilizzando `java.util.ArrayDeque` (implementazione più efficiente per uso generale)
- Utilizzando implementazioni specializzate come `PriorityQueue`

Casi d'Uso

- Pianificazione dei task
- Allocazione delle risorse
- Ricerca in ampiezza (BFS)
- Gestione delle code di stampa
- Code di messaggi nei sistemi distribuiti

4. Coda Circolare

Fondamenti Teorici

Una Coda Circolare è una struttura dati lineare che segue il principio FIFO ma utilizza un array di dimensione fissa in modo più efficiente avvolgendosi all'inizio quando raggiunge la fine.

Caratteristiche Principali

- Dimensione fissa, uso efficiente della memoria
- Struttura circolare (quando raggiunge la fine, si avvolge all'inizio)
- Tipicamente implementata con puntatori front e rear

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`enqueue(E e)`	O(1)	Aggiunge elemento alla fine
`dequeue()`	O(1)	Rimuove e restituisce l'elemento frontale
`peek()`	O(1)	Visualizza l'elemento frontale senza rimuoverlo
`isEmpty()`	O(1)	Verifica se la coda è vuota
`isFull()`	O(1)	Verifica se la coda è piena
`size()`	O(1)	Ottiene il numero di elementi

Dettagli Implementativi

- Tipicamente utilizza un array di dimensione fissa
- Mantiene indici front e rear
- Utilizza l'aritmetica modulo per avvolgersi intorno all'array
- Può essere implementata con o senza contatore di dimensione

Casi d'Uso

- Gestione della memoria
- Gestione del traffico
- Scheduling della CPU
- Buffer di streaming multimediali
- Sistemi in tempo reale con requisiti di memoria fissa

5. Deque (Double-ended Queue)

Fondamenti Teorici

Un Deque (double-ended queue) è una collezione lineare che supporta l'inserimento e la rimozione da entrambe le estremità.

Caratteristiche Principali

- Gli elementi possono essere aggiunti/rimossi da entrambe le estremità
- Combina le caratteristiche di stack e code
- Può essere utilizzato sia come struttura FIFO che LIFO

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`addFirst(E e)`	O(1)	Aggiunge elemento all'inizio
`addLast(E e)`	O(1)	Aggiunge elemento alla fine
`removeFirst()`	O(1)	Rimuove e restituisce l'elemento iniziale
`removeLast()`	O(1)	Rimuove e restituisce l'elemento finale
`getFirst()`	O(1)	Visualizza l'elemento iniziale senza rimuoverlo
`getLast()`	O(1)	Visualizza l'elemento finale senza rimuoverlo
`isEmpty()`	O(1)	Verifica se il deque è vuoto
`size()`	O(1)	Ottiene il numero di elementi

Dettagli Implementativi

In Java, i deque possono essere implementati come:

- `java.util.ArrayDeque` (implementazione con array ridimensionabile, non thread-safe)
- `java.util.LinkedList` (implementazione con lista doppiamente collegata)

Casi d'Uso

- Algoritmi di work stealing
- Implementazione di stack e code
- Controllo dei palindromi
- Problemi con finestra scorrevole
- Algoritmo di scheduling A-Steal

6. Coda Prioritaria

Fondamenti Teorici

Una Coda Prioritaria è un tipo di dato astratto simile a una coda regolare, ma dove ogni elemento ha una "priorità" e gli elementi con priorità più alta vengono serviti prima degli elementi con priorità più bassa.

Caratteristiche Principali

- Gli elementi sono ordinati per priorità anziché per ordine di inserimento
- L'elemento con la priorità più alta è sempre in testa
- Tipicamente implementata con una struttura dati heap

- Non permette elementi null

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`add(E e)` / `offer(E e)`	O(log n)	Aggiunge elemento con priorità
`remove()` / `poll()`	O(log n)	Rimuove e restituisce la priorità più alta
`peek()`	O(1)	Visualizza la priorità più alta senza rimuoverla
`contains(Object o)`	O(n)	Verifica se l'elemento esiste
`isEmpty()`	O(1)	Verifica se la coda è vuota
`size()`	O(1)	Ottiene il numero di elementi

Dettagli Implementativi

- La `PriorityQueue` di Java è implementata con un heap binario
- L'ordinamento naturale o un Comparator determina la priorità
- Min heap di default (elemento più piccolo prima)
- Può essere convertito in max heap utilizzando un Comparator personalizzato

Casi d'Uso

- Algoritmo di Dijkstra per il percorso più breve
- Codifica di Huffman
- Algoritmi di ricerca best-first
- Scheduling dei job basato sulla priorità
- Simulazione guidata dagli eventi

7. Liste Collegate

Fondamenti Teorici

Una Lista Collegata è una struttura dati lineare in cui gli elementi sono memorizzati in nodi, e ogni nodo punta al nodo successivo nella sequenza.

Caratteristiche Principali

- Dimensione dinamica
- Inserimenti e cancellazioni efficienti (quando la posizione è nota)

- Accesso sequenziale (no accesso casuale)
- Tipi: singolarmente collegate, doppiamente collegate, circolari

Operazioni Principali e Complessità

Operazione	Complessità Temporale	Descrizione
`add(E e)`	O(1)	Aggiunge elemento alla fine
`add(int index, E e)`	O(n)	Inserisce elemento a indice specifico
`get(int index)`	O(n)	Recupera elemento a indice specifico
`remove(int index)`	O(n)	Rimuove elemento a indice specifico
`remove(Object o)`	O(n)	Rimuove prima occorrenza dell'elemento
`size()`	O(1)	Ottiene il numero di elementi
`contains(Object o)`	O(n)	Verifica se l'elemento esiste
`isEmpty()`	O(1)	Verifica se la lista è vuota

Dettagli Implementativi

- La `LinkedList` di Java è implementata come lista doppiamente collegata
- Mantiene riferimenti ai nodi primo e ultimo
- Implementa sia l'interfaccia List che Deque
- Ogni nodo contiene dati e riferimenti ai nodi successivo (e precedente)

Casi d'Uso

- Quando sono necessarie frequenti operazioni di inserimento/cancellazione
- Implementazione di stack, code e grafi
- Quando la memoria viene allocata dinamicamente
- Aritmetica polinomiale
- Playlist musicali

8. Insiemi (Set)

Fondamenti Teorici

Un Set è una collezione che non contiene elementi duplicati e modella l'astrazione matematica degli insiemi.

Caratteristiche Principali

- Nessun elemento duplicato
- Al massimo un elemento null
- Diverse implementazioni offrono diversi comportamenti di ordinamento
- Tipicamente fornisce ricerca più veloce rispetto alle liste

Implementazioni Comuni di Set

HashSet

- Utilizza una tabella hash per la memorizzazione
- Nessun ordine garantito
- Consente valore null
- O(1) per ricerca, inserimento, cancellazione

TreeSet

- Implementazione con Red-Black Tree
- Elementi memorizzati in ordine
- Non consente null
- O(log n) per ricerca, inserimento, cancellazione

LinkedHashSet

- Tabella hash con lista collegata
- Mantiene l'ordine di inserimento
- Consente valore null
- O(1) per ricerca, inserimento, cancellazione con ordinamento

Operazioni Principali e Complessità

Operazione	Tempo HashSet	Tempo TreeSet	Descrizione
`add(E e)`	O(1)	O(log n)	Aggiunge elemento
`remove(Object o)`	O(1)	O(log n)	Rimuove elemento
`contains(Object o)`	O(1)	O(log n)	Verifica esistenza
`size()`	O(1)	O(1)	Conta elementi
`isEmpty()`	O(1)	O(1)	Verifica se vuoto
`iterator()`	O(1)	O(1)	Ottiene iteratore

Dettagli Implementativi

- HashSet utilizza HashMap internamente

- TreeSet utilizza TreeMap internamente
- LinkedHashSet utilizza LinkedHashMap internamente
- Le prestazioni di HashSet dipendono dalla qualità dell'implementazione di hashCode()

Casi d'Uso

- Rimozione dei duplicati da una collezione
- Test di appartenenza
- Operazioni matematiche su insiemi (unione, intersezione)
- Controllo dell'unicità
- Implementazione di dizionari

9. Mappe (Map)

Fondamenti Teorici

Una Map è un oggetto che mappa chiavi a valori. Non può contenere chiavi duplicate; ogni chiave può essere mappata al massimo a un valore.

Caratteristiche Principali

- Coppie chiave-valore
- Nessuna chiave duplicata
- Diverse implementazioni offrono diversi comportamenti di ordinamento
- Può o meno consentire chiavi e valori null

Implementazioni Comuni di Map

HashMap

- Utilizza una tabella hash per la memorizzazione
- Nessun ordine garantito
- Consente una chiave null e molteplici valori null
- O(1) per ricerca, inserimento, cancellazione

TreeMap

- Implementazione con Red-Black Tree
- Elementi memorizzati in ordine di chiave
- Non consente chiavi null
- O(log n) per ricerca, inserimento, cancellazione

LinkedHashMap

- Tabella hash con lista collegata
- Mantiene l'ordine di inserimento o di accesso
- Consente una chiave null e molteplici valori null
- O(1) per ricerca, inserimento, cancellazione con ordinamento

Operazioni Principali e Complessità

Operazione	Tempo HashMap	Tempo TreeMap	Descrizione
`put(K key, V value)`	O(1)	O(log n)	Aggiunge o aggiorna coppia
`get(Object key)`	O(1)	O(log n)	Recupera valore
`remove(Object key)`	O(1)	O(log n)	Rimuove coppia per chiave
`containsKey(Object)`	O(1)	O(log n)	Verifica esistenza chiave
`containsValue(Object)`	O(n)	O(n)	Verifica esistenza valore
`size()`	O(1)	O(1)	Conta coppie
`isEmpty()`	O(1)	O(1)	Verifica se vuoto

Dettagli Implementativi

- HashMap utilizza un array di bucket, ognuno contenente una lista collegata o un albero di entry
- TreeMap utilizza una struttura Red-Black tree
- LinkedHashMap estende HashMap con una lista doppiamente collegata
- Il fattore di carico influenza prestazioni e utilizzo della memoria

Casi d'Uso

- Dizionari e ricerche
- Caching
- Conteggio occorrenze (mappe di frequenza)
- Indicizzazione dei dati
- Rappresentazione di strutture dati sparse

10. Tabelle Hash

Fondamenti Teorici

Una tabella hash è una struttura dati che implementa un tipo di dato astratto di array associativo, una struttura che può mappare chiavi a valori utilizzando una funzione hash.

Caratteristiche Principali

- Utilizza una funzione hash per calcolare un indice in un array
- Idealmente fornisce complessità temporale media O(1) per le ricerche
- Deve gestire le collisioni quando chiavi diverse producono lo stesso indice
- Richiede buone funzioni hash per minimizzare le collisioni

Strategie di Risoluzione delle Collisioni

Concatenamento

- Ogni bucket contiene una lista di tutti gli elementi che producono lo stesso indice hash
- Tecnica di risoluzione delle collisioni più semplice
- Le prestazioni degradano con l'allungamento delle catene

Indirizzamento Aperto

- Tutti gli elementi sono memorizzati direttamente nell'array della tabella hash
- Quando si verifica una collisione, si cerca il prossimo slot disponibile
- Varianti: Sondaggio lineare, Sondaggio quadratico, Doppio hashing

Operazioni Principali e Complessità

Operazione	Caso Medio	Caso Peggiore	Descrizione
`insert(key, value)`	O(1)	O(n)	Aggiunge o aggiorna coppia
`search(key)`	O(1)	O(n)	Trova valore per chiave
`delete(key)`	O(1)	O(n)	Rimuove coppia chiave-valore

Dettagli Implementativi

- HashMap e HashSet di Java utilizzano internamente tabelle hash
- La capacità iniziale predefinita è 16 bucket
- Il fattore di carico predefinito è 0,75
- Il rehashing si verifica quando viene superata la soglia del fattore di carico
- Da Java 8, i bucket si convertono da liste collegate ad alberi quando diventano grandi

Casi d'Uso

- Implementazione di map e set
- Indicizzazione di database
- Caching

- Tabelle dei simboli nei compilatori
- Interning delle stringhe

Riepilogo della Complessità Computazionale

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

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

Scegliere la Struttura Dati Corretta

Quando si seleziona una struttura dati, considerare:

1. **Operazioni necessarie**: Quali operazioni verranno eseguite più frequentemente?
2. **Complessità temporale**: Quali caratteristiche di prestazioni sono richieste?
3. **Vincoli di memoria**: Quanta memoria è disponibile?
4. **Requisiti di ordinamento**: Gli elementi devono essere in un ordine specifico?
5. **Accesso concorrente**: La struttura sarà acceduta da più thread?

Riferimento Rapido

- **Necessità di accesso casuale veloce per indice?** → ArrayList
- **Necessità di operazioni LIFO?** → Stack o ArrayDeque
- **Necessità di operazioni FIFO?** → Queue, LinkedList, o ArrayDeque
- **Necessità di inserimento/cancellazione a tempo costante da entrambe le estremità?** → ArrayDeque
- **Necessità di elementi ordinati in base alla priorità?** → PriorityQueue
- **Necessità di inserimento/cancellazione veloce nel mezzo?** → LinkedList

- **Necessità di eliminare i duplicati?** → HashSet o TreeSet
- **Necessità di mappature chiave-valore?** → HashMap o TreeMap
- **Necessità di mappature chiave-valore in ordine?** → TreeMap
- **Necessità di preservare l'ordine di inserimento?** → LinkedHashMap o LinkedHashSet