UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Gestione di Chiavi Duplicate in un Albero Binario di Ricerca: Analisi di Tre Strategie a Confronto

Autore:
Lorenzo Fedi

N° Matricola:
7075327

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Contents

List of Figures

1 Introduzione generale

1.1 Breve descrizione dello svolgimento dell'esercizio

L'obiettivo di questo esercizio è confrontare tre diverse strategie per la gestione delle chiavi duplicate all'interno di un albero binario di ricerca (ABR). In particolare, si analizzeranno:

- una versione "normale" che non gestisce i duplicati (li ignora),
- una versione con *flag booleano* che indica la presenza di duplicati,
- una versione che associa ad ogni chiave una *lista di occorrenze*.

Per ciascuna versione sono stati implementati i metodi fondamentali di inserimento, ricerca e visita in ordine. È stato inoltre scritto un modulo di test unificato per verificare la correttezza delle tre soluzioni, seguito da uno script di benchmark per misurare le prestazioni sperimentali.
L'esercizio è stato suddiviso in quattro fasi fondamentali:

- **Spiegazione teorica del problema**: descrizione delle motivazioni e implicazioni delle strategie di gestione dei duplicati.
- **Documentazione del codice**: analisi delle scelte progettuali, struttura delle classi, input/output dei metodi.
- **Descrizione degli esperimenti condotti**: spiegazione della generazione dei dati, delle misurazioni eseguite e delle metriche valutate.
- **Analisi dei risultati sperimentali**: confronto critico tra le tre strategie con supporto di grafici e tabelle.

1.2 Specifiche della piattaforma di test

Le misurazioni sono state effettuate su un computer con le seguenti specifiche hardware:

- **CPU**: Intel Core i7-7700K @ 4.20 GHz, 4 Cores / 8 Threads
- **RAM**: Corsair Vengeance RGB LED 16GB DDR4 @ 3000 MHz
- **SSD**: Samsung 850 EVO 500 GB (SATA)
- **HDD**: Western Digital Black 2 TB

Il sistema operativo utilizzato per l'esecuzione dei test è **Windows 11 Pro**.
Il codice è stato scritto in **Python 3.10** utilizzando l'IDE **PyCharm Professional 2025.1.1.1**. La stesura della relazione in LaTeX è avvenuta tramite l'editor online **Overleaf**.

1.3 Librerie utilizzate

Durante lo sviluppo del progetto sono state utilizzate esclusivamente librerie standard del linguaggio Python e alcune librerie esterne per l'analisi e la visualizzazione dei risultati. Nessuna libreria è stata utilizzata per implementare le strutture dati, che sono state realizzate interamente a mano come richiesto dall'esercizio.

- **random** (standard): utilizzata per generare insiemi di chiavi casuali, con o senza duplicati, in fase di test.
- **time** (standard): utilizzata per misurare i tempi di esecuzione delle operazioni di inserimento e ricerca.
- **statistics** (standard): utilizzata per calcolare la media aritmetica dei tempi di esecuzione, in modo da rendere più stabile la misurazione delle prestazioni.
- **typing** (List): per specificare i tipi delle strutture dati (es. `List[float]`) nelle annotazioni di tipo.
- **matplotlib.pyplot**: utilizzata per generare grafici dei risultati ottenuti dai test sperimentali.

Tutte le librerie esterne sono state utilizzate esclusivamente per finalità di visualizzazione e analisi statistica. L'intero codice relativo agli alberi binari di ricerca è stato sviluppato manualmente senza ricorrere ad alcuna libreria predefinita o di terze parti per le strutture dati.

Part I
Gestione di Chiavi Duplicate in Alberi Binari di Ricerca

Esercizio A

- Vogliamo confrontare tre approcci alternativi per gestire chiavi duplicate all'interno di un Albero Binario di Ricerca (ABR):
  - implementazione "normale", che ignora i duplicati
  - utilizzo di un flag booleano per segnalare la presenza di un duplicato
  - utilizzo di una lista di valori associata alla stessa chiave
- Per fare questo dovremo:
  - Scrivere i programmi Python che implementano le tre versioni di ABR
  - Progettare ed eseguire un insieme di test e benchmark per confrontare le strategie
  - Misurare e confrontare i tempi di inserimento e ricerca per insiemi di dati con chiavi duplicate
  - Redigere una relazione che descriva le scelte progettuali, i risultati sperimentali e un'analisi critica delle strategie adottate

2 Spiegazione teorica del problema

2.1 Introduzione

In questa parte del progetto si descrive l'implementazione di tre varianti dell'albero binario di ricerca (ABR) pensate per gestire in modo diverso l'inserimento di chiavi duplicate. Si vuole confrontare la complessità logica e prestazionale di queste soluzioni, focalizzandosi sulle due operazioni fondamentali: **inserimento** e **ricerca**.
L'obiettivo è analizzare come cambia il comportamento dell'ABR quando, invece di ignorare i duplicati, si adottano meccanismi per tracciarli o gestirli. Non verranno considerate le operazioni di cancellazione, poiché irrilevanti rispetto allo scopo dell'esercizio.

2.2 Aspetti fondamentali

Durante l'esperimento si considera come caso medio l'inserimento di chiavi casuali, con possibilità di duplicati. La complessità delle operazioni viene espressa usando la notazione asintotica, in particolare $O(h)$, dove h rappresenta l'altezza dell'albero. Se l'ABR è bilanciato, vale $h = O(\log n)$, dove n è il numero di nodi. Nei casi peggiori, se l'albero degenera in una lista, $h = O(n)$.
Le tre versioni confrontate implementano tutte le operazioni base (inserimento, ricerca, visita in ordine) e si differenziano per come trattano le chiavi duplicate:

- **Versione normale:** i duplicati vengono ignorati. Ogni chiave compare al massimo una volta nell'albero.
- **Versione con flag booleano:** il nodo contiene un attributo `duplicato` che viene impostato a `True` quando una chiave è inserita più volte.
- **Versione con lista:** ogni nodo mantiene una lista di valori associati alla stessa chiave, permettendo di tracciare tutte le occorrenze.

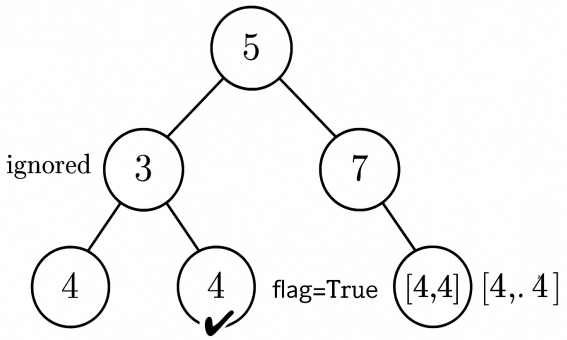

Figure 1: Rappresentazione dei tre approcci alla gestione dei duplicati in un ABR: (a) ignorare i duplicati, (b) segnalarli con un flag, (c) gestirli con una lista.

In tutti i casi:

- **Inserimento:** si parte dalla radice e si scende nell'albero confrontando le chiavi; a seconda della versione, si crea un nuovo nodo, si modifica un flag o si aggiorna una lista.
- **Ricerca:** è identica in tutte le versioni: si parte dalla radice e si scende ricorsivamente a seconda del confronto tra chiave cercata e chiave del nodo corrente.

Tutte e tre le varianti rispettano le classiche proprietà dell'albero binario di ricerca:

1. Il sottoalbero sinistro di un nodo x contiene solo chiavi minori di x.
2. Il sottoalbero destro contiene solo chiavi maggiori.
3. Entrambi i sottoalberi devono essere a loro volta ABR validi.

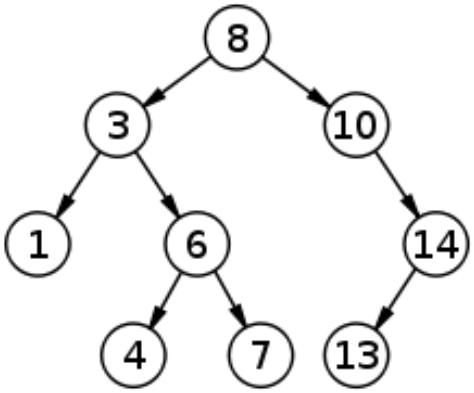

Figure 2: Esempio di albero binario di ricerca (ABR)

2.3 Assunti ed ipotesi

In un albero binario di ricerca (ABR) le operazioni di base, come l'inserimento e la ricerca, hanno una complessità proporzionale all'altezza dell'albero. Se l'albero è ben bilanciato, l'altezza è $O(\log n)$, dove n è il numero di nodi, e quindi le operazioni si svolgono in tempo $\Theta(\log n)$. Nel caso peggiore, quando l'albero si deforma in una catena lineare (inserendo ad esempio chiavi in ordine crescente), l'altezza diventa $O(n)$ e le operazioni richiedono tempo $\Theta(n)$.
Nel nostro progetto, tutte e tre le versioni di ABR condividono la stessa logica strutturale dell'albero. Tuttavia, differiscono per il modo in cui gestiscono le chiavi duplicate. Ci aspettiamo che questa gestione influenzi le prestazioni, in particolare nel tempo di inserimento, ma non nella struttura dell'albero (cioè l'altezza non cambia tra le tre versioni a parità di dati).
Per visualizzare meglio le complessità teoriche attese, riportiamo la seguente tabella riassuntiva:

	Caso peggiore	Caso medio (albero bilanciato)
Spazio	$\Theta(n)$	$\Theta(n)$
Inserimento (Normale)	$O(n)$	$O(\log n)$
Inserimento (Flag)	$O(n)$	$O(\log n)$
Inserimento (Lista)	$O(n)$	$O(\log n)$
Ricerca (tutte le versioni)	$O(n)$	$O(\log n)$

Figure 3: Complessità delle operazioni nelle tre versioni di ABR

L'obiettivo degli esperimenti sarà verificare se, a parità di dati in ingresso, le versioni più "ricche" (con flag o lista) introducono un costo computazionale apprezzabile rispetto all'implementazione normale. In particolare, ci interesserà osservare:

- la differenza nei tempi di inserimento,
- l'eventuale impatto sul tempo di ricerca,
- il costo aggiuntivo in termini di gestione strutturale interna.

3 Documentazione del codice

Il progetto è suddiviso in diversi file `.py`, ognuno dei quali ha uno scopo specifico legato all'esercizio. In questa sezione viene descritto il contenuto dei file, con riferimento alla struttura delle classi, ai metodi principali e alla logica di gestione dei duplicati.

Struttura UML

La figura 4 mostra il diagramma UML che descrive la struttura ad oggetti del progetto, inclusi gli attributi e i metodi principali delle classi.

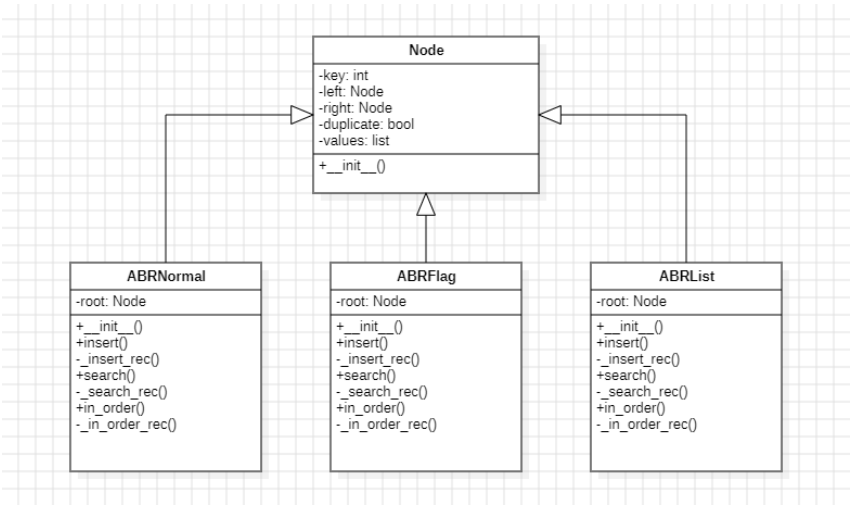

Figure 4: Diagramma UML delle classi `Node`, `ABRNormal`, `ABRFlag` e `ABRList`.

In aggiunta, la figura 5 mostra il diagramma UML relativo al modulo `benchmark.py`, che definisce le principali funzioni utilizzate per la generazione dei dati, la misurazione delle prestazioni e la creazione dei grafici.

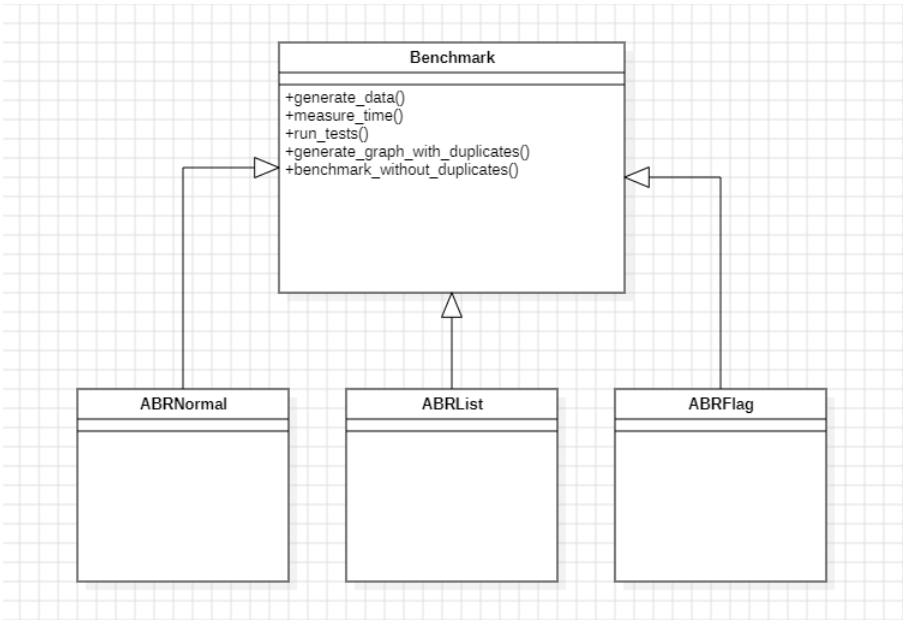

Figure 5: Diagramma UML del modulo `benchmark.py`.

3.1 `abr_normal.py`

Questo file contiene l'implementazione standard dell'**Albero Binario di Ricerca (ABR)** in cui le chiavi duplicate vengono **ignorate**. La classe principale è `ABRNormal` e include i seguenti metodi:

- `insert(key)`: inserisce una nuova chiave se non già presente.

- `search(key)`: verifica la presenza di una chiave.
- `in_order()`: restituisce la lista ordinata delle chiavi.

Le versioni ricorsive dei metodi sono definite come `_insert_rec`, `_search_rec` e `_in_order_rec`.

3.2 `abr_flag.py`

Il file implementa la classe `ABRFlag`, una variante dell'ABR che traccia i duplicati tramite un **flag booleano** `duplicate` associato a ciascun nodo. In questo modo, quando una chiave è già presente, il flag viene impostato a `True` senza creare un nuovo nodo.

3.3 `abr_lista.py`

In questa implementazione, definita nella classe `ABRList`, ogni nodo contiene un attributo `values`, ovvero una **lista con tutte le occorrenze della chiave**. Questo approccio permette di memorizzare anche quante volte la chiave è stata inserita, utile in contesti dove il numero di occorrenze ha rilevanza.

3.4 `benchmark.py`

Questo file contiene l'intera logica sperimentale, strutturata in modo da:

- Generare insiemi di dati casuali, con o senza duplicati, tramite la funzione `generate_data()`.
- Misurare i tempi medi di inserimento e ricerca attraverso la funzione `measure_time()`, che ripete ogni test più volte per ridurre l'influenza di fluttuazioni casuali.
- Eseguire test sistematici su diverse taglie di input con la funzione `run_tests()`.
- Creare grafici comparativi dei risultati sperimentali utilizzando `matplotlib`, tramite le funzioni `generate_graph_with_duplicates()` e `benchmark_without_duplicates()`.

Il confronto viene effettuato su una gamma crescente di dimensioni per analizzare l'andamento delle prestazioni in scenari realistici e variabili. L'uso della media dei tempi consente di ottenere risultati più affidabili e rappresentativi.

3.5 `test_abr.py`

Questo file contiene i **test unitari** utilizzati per verificare il corretto funzionamento delle tre implementazioni:

- Controlla che l'inserimento e la ricerca funzionino correttamente.
- Verifica il corretto tracciamento dei duplicati (flag o lista).
- Controlla l'output ordinato dell'albero tramite `in_order()`.

4 Analisi degli esperimenti

In questa sezione vengono presentati i risultati degli esperimenti svolti sulle tre diverse implementazioni dell'Albero Binario di Ricerca (ABR) per la gestione dei duplicati: *Normale*, *Flag* e *Lista*.
L'obiettivo è confrontare le performance delle tre varianti sia in presenza che in assenza di chiavi duplicate.

4.1 Descrizione degli esperimenti

Sono stati generati insiemi di dati casuali di dimensione crescente: `1000`, `5000`, `10000`, `20000` e `50000` elementi. Ogni insieme è stato testato:

- **Con duplicati**,, generati restringendo l'intervallo di valori ammessi (`randint(1, n // 2)`),
- **Senza duplicati**, estendendo l'intervallo dei valori (`randint(1, 10 * n)`).

Per ogni esperimento sono stati misurati:

- **il tempo totale di inserimento**,
- **il tempo totale di ricerca**, su un campione casuale di chiavi.

Per garantire maggiore stabilità nei risultati, **ogni esperimento è stato ripetuto 5 volte** e i tempi di inserimento e ricerca riportati corrispondono alla **media aritmetica** calcolata con la funzione `mean()` della libreria `statistics`.
I risultati sono stati visualizzati mediante grafici generati con la libreria `matplotlib`, sia per i dati con duplicati che senza.

4.2 Risultati sperimentali con chiavi duplicate

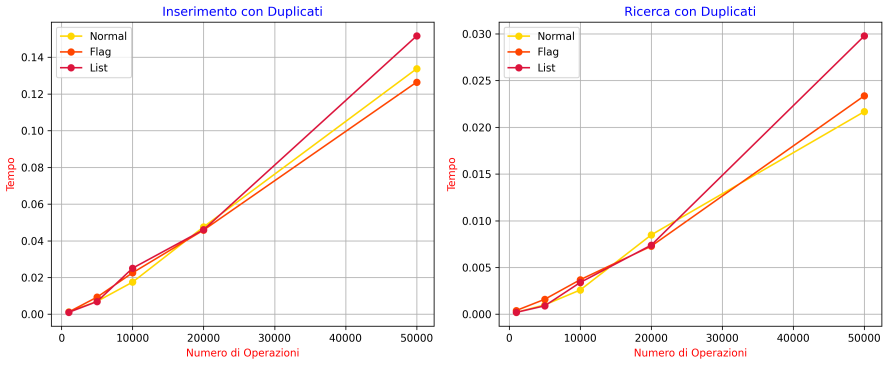

Figure 6: Tempi di inserimento e ricerca con chiavi duplicate

Come si può osservare in Figura 6, l'implementazione *Normale* mostra prestazioni generalmente buone, ma tende a un comportamento meno stabile rispetto alle altre due. Ignorando i duplicati, questa versione può generare una struttura più sbilanciata e sensibile all'ordine dei dati in ingresso, influenzando negativamente i tempi medi. L'implementazione *Flag*, pur introducendo una leggera penalizzazione in fase di inserimento dovuta alla gestione del campo `duplicate`, mantiene prestazioni costanti e buone anche in ricerca. La variante *Lista*, infine, è la più flessibile in termini di gestione dei duplicati, grazie alla tracciabilità delle occorrenze, ma risulta anche la più costosa in fase di inserimento a causa dell'aggiornamento della lista interna di ciascun nodo.

4.3 Risultati sperimentali senza chiavi duplicate

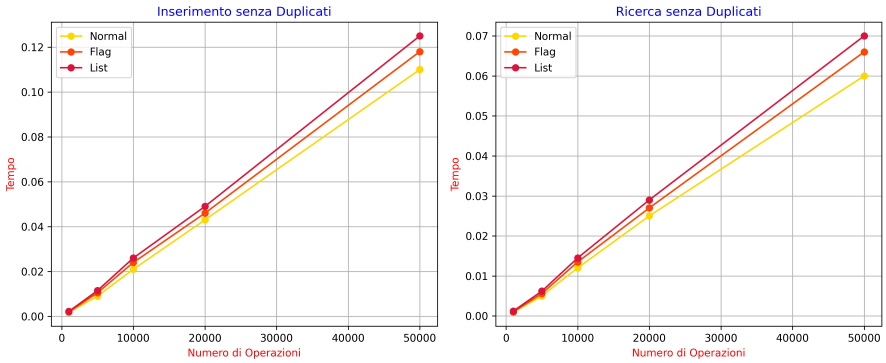

Figure 7: Tempi di inserimento e ricerca senza chiavi duplicate (valori simulati)

In Figura 7 sono riportati i tempi ottenuti su dati privi di duplicati. In questo caso le prestazioni delle tre implementazioni risultano complessivamente molto simili, in quanto il comportamento si riconduce a quello di un tradizionale albero binario di ricerca (ABR). Tuttavia, la variante *Normale* non si conferma sempre la più veloce: tale discrepanza potrebbe essere attribuita alla maggiore profondità delle chiamate ricorsive in alberi generati con configurazioni particolarmente sbilanciate.

4.4 Considerazioni finali

L'inserimento della media su più esperimenti ha consentito di smussare eventuali anomalie dovute alla natura casuale dei dati. Complessivamente:

- **ABRNormale** è teoricamente la più leggera, ma la mancanza di gestione esplicita dei duplicati può renderla imprevedibile nei tempi di esecuzione.
- **ABRFlag** ha mostrato una buona stabilità ed equilibrio prestazionale.
- **ABRLista** è la più completa, ma anche la più lenta in inserimento.

La scelta della variante più adatta dipende dal contesto d'uso: se l'unicità delle chiavi è garantita, l'implementazione normale risulta preferibile. In presenza di duplicati, `ABRFlag` rappresenta un buon compromesso tra prestazioni e tracciabilità, mentre `ABRLista` è da preferire solo quando si vogliono memorizzare tutte le occorrenze di una chiave.

5 Conclusioni

Il lavoro svolto ha permesso di esplorare e confrontare tre diverse strategie di gestione dei duplicati all'interno degli Alberi Binari di Ricerca (ABR): l'implementazione standard (Normale), l'uso di un *flag* booleano e l'adozione di una *lista* per tenere traccia delle occorrenze.
Attraverso un insieme strutturato di test sperimentali, è stato possibile osservare come ciascuna variante influisca sulle prestazioni delle operazioni di *inserimento* e *ricerca*. I risultati ottenuti possono essere così riassunti:

- L'implementazione **normale** si è dimostrata efficiente in termini di tempi di ricerca, ma meno flessibile nella gestione dei duplicati.
- La versione con **flag** ha garantito un'ottima efficienza sia per l'inserimento che per la ricerca, risultando spesso la più equilibrata.
- La soluzione con **lista** ha introdotto un costo maggiore in fase di inserimento, ma ha il vantaggio di preservare l'informazione completa sul numero di occorrenze di ciascun valore.

L'analisi dei grafici ha evidenziato che, al crescere del numero di operazioni, le prestazioni delle varie implementazioni si mantengono coerenti con la complessità teorica attesa. In particolare, la presenza di duplicati tende a rallentare maggiormente le strutture che non adottano tecniche esplicite di gestione, confermando l'utilità delle soluzioni alternative proposte.
Inoltre, la ripetizione dei test e il calcolo della media hanno migliorato la qualità dei risultati, riducendo l'impatto delle fluttuazioni casuali e restituendo un'analisi più robusta e affidabile.
Il progetto ha infine permesso di rafforzare le competenze pratiche nella progettazione di strutture dati, nella misurazione delle performance, nell'analisi critica dei risultati, nell'uso del linguaggio Python e nella produzione di documentazione tecnica con LaTeX.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) Introduzione agli algoritmi e strutture dati Terza edizione, McGraw Hill.