UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

B-alberi vs Alberi Binari di Ricerca

Davide Meta

N° Matricola: 7136156

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Prof. Simone Marinai

Contents

List of Figures

1 Introduzione generale

1.1 Breve descrizione dello svolgimento del progetto

Per questo progetto suddividiamo la descrizione in 5 parti fondamentali:

- **Introduzione generale**: presentazione del progetto e delle specifiche della piattaforma di test.
- **Spiegazione teorica del problema**: descrizione teorica dei B-alberi e degli Alberi Binari di Ricerca, in particolar modo analizzando gli accessi a memoria secondaria.
- **Documentazione del codice**: spiegazione del codice e dei metodi utilizzati.
- **Descrizione degli esperimenti condotti**: presentazione dei test effettuati e verifica delle prestazioni (in termini di accessi a disco) basandosi sulle ipotesi teoriche.
- **Analisi dei risultati sperimentali**: riflessione sui risultati ottenuti e conclusioni finali.

1.2 Specifiche della piattaforma di test

La piattaforma di test utilizzata presenta le seguenti caratteristiche hardware:

- **CPU**: Apple M3 (8-core CPU con 4 performance core e 4 efficiency core)
- **GPU**: Integrata (Apple M3 10-core GPU)
- **RAM**: 16GB Unified Memory
- **SSD**: SSD NVMe integrato 512GB
- **HDD**: Non presente

Il linguaggio di programmazione utilizzato è **Python 3.x** su Visual Studio Code (**VS Code**). La stesura di questa relazione è avvenuta tramite *LaTeX* sull'editor online **Overleaf**.

1.3 Richiesta dell'esercizio

"I B-alberi sono strutture dati per memoria secondaria in cui ogni nodo può avere molti figli. Implementare B-alberi e confrontarli con la memorizzazione in alberi binari di ricerca, sapendo che in memoria secondaria gli algoritmi si confrontano sulla base degli accessi a disco (in questo caso possiamo considerare il numero di nodi letti o scritti)."

1.4 Obiettivi del progetto

Lo scopo pratico del progetto è quello di mettere a confronto B-alberi e Alberi Binari di Ricerca, osservando in particolare quanti accessi al disco servono nelle varie operazioni, in modo tale da capire quale delle due strutture conviene usare quando i dati stanno su memoria di massa.

2 Spiegazione teorica del problema

2.1 Introduzione

In questa parte del progetto andremo a definire e descrivere l'implementazione dei B-alberi. Infatti i **B-alberi** sono strutture dati progettate per l'archiviazione in memoria secondaria, dove il costo degli accessi a disco è decisamente maggiore rispetto agli accessi in memoria primaria. A differenza degli Alberi Binari di Ricerca (**ABR**), i B-alberi sono progettati per minimizzare il numero di accessi a disco mantenendo più chiavi per nodo.
Per semplicità andremo a confrontare ed implementare soltanto i metodi di inserimento e ricerca tra le due strutture dati. Il metodo di cancellazione verrà trattato soltanto da un punto di vista teorico.

2.2 Aspetti fondamentali

2.2.1 Alberi Binari di Ricerca (ABR)

Un ABR è un albero binario dove ogni nodo contiene una chiave e al massimo due figli. Le sue proprietà fondamentali sono:

1. Il sottoalbero sinistro di un nodo contiene solo nodi con chiavi minori.
2. Il sottoalbero destro di un nodo contiene solo nodi con chiavi maggiori.
3. Entrambi i sottoalberi sono ABR.

2.2.2 B-alberi

Un B-albero di ordine t è un albero di ricerca bilanciato dove ogni nodo può contenere al massimo $t-1$ chiavi e t figli. Le proprietà sono:

1. Ogni nodo ha al massimo t figli.
2. Ogni nodo interno (non foglia) ha almeno $\lceil t/2 \rceil$ figli.
3. La radice ha almeno 2 figli (se non è una foglia).
4. Tutte le foglie sono allo stesso livello.
5. Un nodo con k figli contiene $k-1$ chiavi.

2.3 Confronto teorico sulla complessità asintotica

In un **ABR** le operazioni di base richiedono un tempo proporzionale all'altezza dell'albero. L'altezza attesa di un ABR costruito in modo casuale è $O(\log n)$, mentre nel caso peggiore (catena lineare) l'altezza è $O(n)$.
Invece i **B-alberi** sono auto-bilanciati per costruzione, garantendo di conseguenza un'altezza $h = \log_t n$. Quindi le operazioni di inserimento, ricerca e cancellazione hanno sempre complessità logaritmica che dipende dal numero di chiavi n e dalla base t.

Operazione	**B-alberi**	**ABR**	
		Caso medio	**Caso peggiore**
Ricerca	$O(\log_t n)$	$O(\log n)$	$O(n)$
Inserimento	$O(\log_t n)$	$O(\log n)$	$O(n)$
Cancellazione	$O(\log_t n)$	$O(\log n)$	$O(n)$

Table 1: Confronto delle complessità asintotiche tra ABR e B-alberi.

2.4 Assunti ed ipotesi

Per i nostri esperimenti assumiamo:

- Ogni accesso a un nodo corrisponde a un accesso a disco
- Il costo di accesso a disco è uniforme
- Confrontiamo B-alberi di ordine *t=[5, 10, 15]* con ABR
- Testiamo con dataset di dimensioni *sizes=[10, 100, 1000]* elementi

3 Documentazione del codice

3.1 Schema del contenuto e interazione tra i moduli

L'implementazione è strutturata nei seguenti moduli:

- **BTree.py**: Implementazione del B-albero
- **ABR.py**: Implementazione dell'Albero Binario di Ricerca
- **test.py**: Framework per l'esecuzione dei test e generatore di grafici per l'analisi
- **main.py**: Main per la simulazione degli accessi a disco

3.2 Analisi delle scelte implementative

Il B-albero è stato implementato rappresentando i nodi come liste di chiavi in ordine crescente ed ogni nodo ha un'altra lista che contiene i riferimenti (puntatori) ai nodi figli. Quando un nodo supera la capacità massima permessa, viene eseguito lo *split*, così da mantenere la struttura bilanciata. Per simulare in modo realistico gli accessi a disco, è stato introdotto un contatore *disk_access_log* che viene incrementato a ogni accesso a un nodo.

3.2.1 Classe BTree

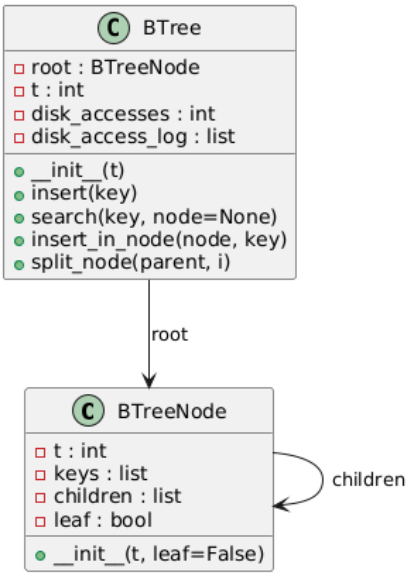


Figure 1: Diagramma UML della classe BTree

I principali metodi implementati nella classe **BTree** sono:

- **__init__(self, t)**: Inizializza il B-albero di ordine *t*.
- **insert(self, key)**: Inserisce una chiave nell'albero.
- **search(self, key, node)**: Ricerca una chiave nell'albero.
- **insert_in_node(self, node, key)**: Inserisce una chiave in un nodo specifico.
- **split_node(self, parent, i)**: Divide un nodo pieno.

Mentre **BTreeNode** ha soltanto il metodo **__init__(self, t, leaf)** che serve per inizializzare il nodo del B-albero.

3.2.2 Classe ABR

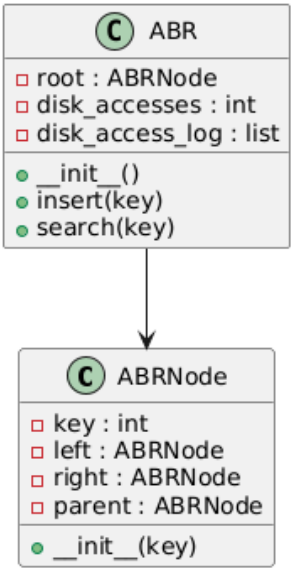


Figure 2: Diagramma UML della classe ABR

I principali metodi implementati nella classe **ABR** sono:

- **__init__(self)**: Inizializza un ABR vuoto.
- **insert(self, key)**: Inserisce un nodo nell'albero.
- **search(self, key)**: Ricerca un nodo nell'albero.

Analogamente a *BTreeNode*, anche **ABRNode** possiede soltanto il metodo **__init__(self, key)** per inizializzare il nodo dell'ABR.

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali

4.1 Dati utilizzati

Gli esperimenti sugli ABR e B-alberi sono stati condotti con diverse configurazioni sul numero di nodi inseriti, sull'ordine del B-albero e sul tipo di inserimento e ricerca.
I valori che sono stati scelti per questo test sono i seguenti:

- **Dimensioni dei dataset**: *sizes = [10, 100, 1000]*
- **Configurazioni B-albero**: *t_values = [5, 10, 15]*
- **Tipi di test**: *scelte = ["insert_sequenziale", "search_sequenziale", "insert_random", "search_random"]*
  - **Inserimento sequenziale**: chiavi inserite in ordine crescente (caso peggiore per ABR)
  - **Inserimento casuale**: chiavi inserite in ordine casuale (caso medio per ABR)
  - **Ricerca sequenziale**: ricerca di tutte le chiavi in ordine
  - **Ricerca casuale**: ricerca di chiavi in ordine casuale

Per la misurazione del numero totale di accessi a disco è stata utilizzata la variabile *disk_access_log*, che ad ogni operazione di inserimento o ricerca andata a buon fine, incrementava di uno il suo valore.

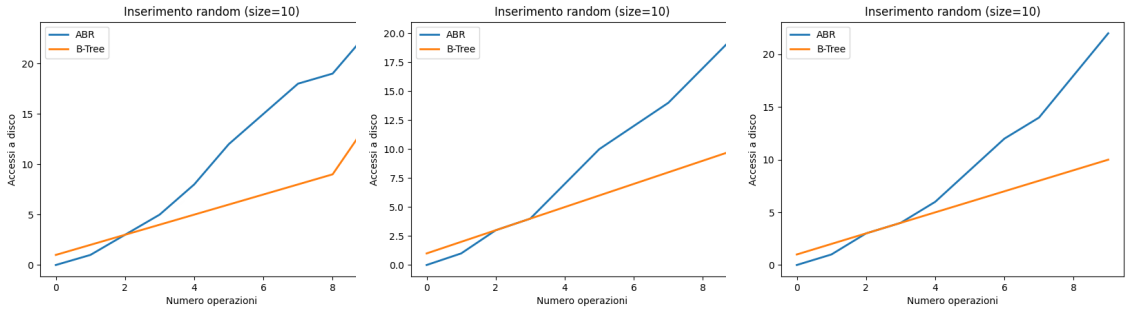
4.2 Risultati sperimentali

Per valutare le prestazioni delle due strutture dati, all'interno del file *test.py*, sono stati effettuati diversi esperimenti andando a cambiare i seguenti parametri:

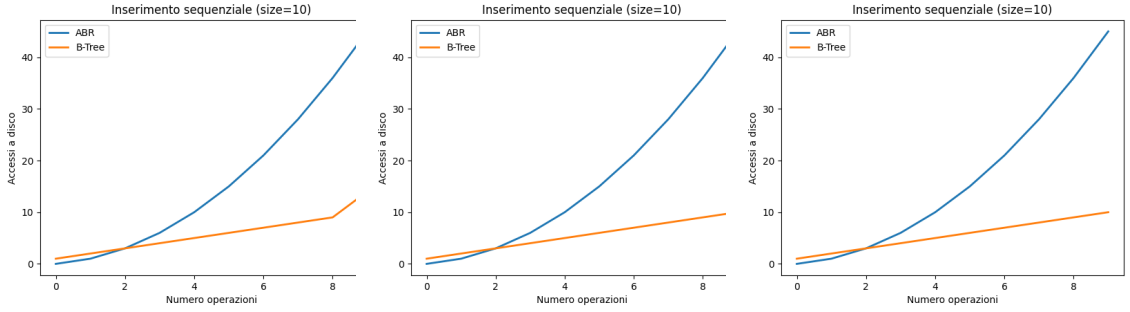
- **Dimensione dell'input (sizes)**: diversi ordini di grandezza di dati inseriti.
- **Valore di t (t_values)**: grado del B-albero.
- **Scelta delle operazioni**: inserimento sequenziale, ricerca sequenziale, inserimento casuale, ricerca casuale.

Grafici

4.2.1 Inserimento



(a) Random, $t = 5$, $n = 10$ (b) Random, $t = 10$, $n = 10$ (c) Random, $t = 15$, $n = 10$

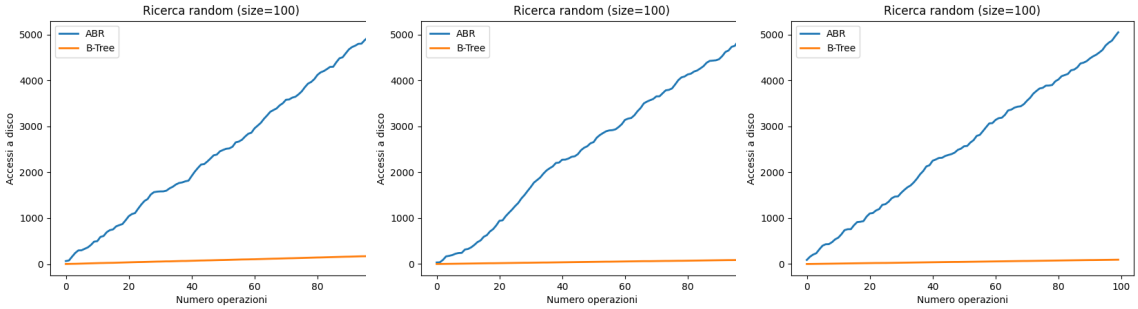


(d) Seq, $t = 5$, $n = 10$ (e) Seq, $t = 10$, $n = 10$ (f) Seq, $t = 15$, $n = 10$

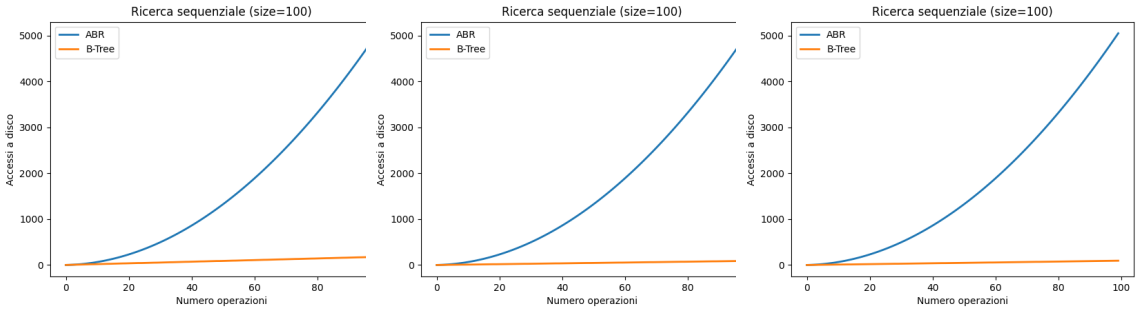
Figure 3: Confronto accessi a disco durante l'inserimento al variare di t con $n = 10$.

Commento: Dai grafici si osserva che aumentando t diminuisce il numero di accessi a disco, confermando l'ipotesi teorica che un grado maggiore riduce la profondità dell'albero.

4.2.2 Ricerca



(a) Random, $t = 5$, $n = 100$ (b) Random, $t = 10$, $n = 100$ (c) Random, $t = 15$, $n = 100$

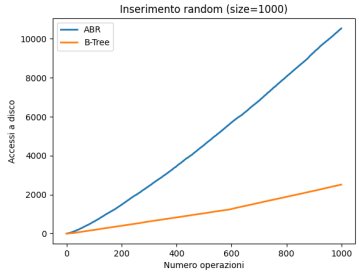


(d) Seq, $t = 5$, $n = 100$ (e) Seq, $t = 10$, $n = 100$ (f) Seq, $t = 15$, $n = 100$

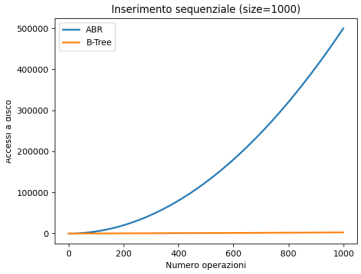
Figure 4: Confronto accessi a disco durante la ricerca al variare di t con $n = 100$.

Commento: Come si può notare dai grafici, la ricerca sequenziale su ABR conferma il comportamento lineare $O(n)$, mentre nei B-alberi il numero di accessi cresce in modo logaritmico, visibile attraverso un incremento contenuto, soprattutto al crescere dei valori di t.

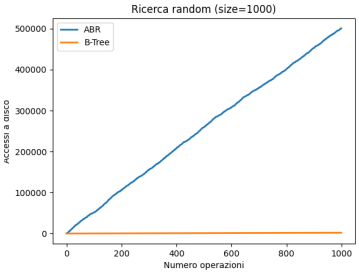
4.2.3 Scalabilità



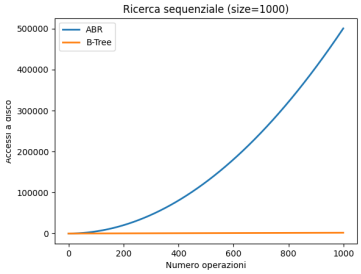
(a) Inserimento Random, $t = 15$, $n = 1000$



(b) Inserimento Sequenziale, $t = 15$, $n = 1000$



(c) Ricerca Random, $t = 15$, $n = 1000$



(d) Ricerca Sequenziale, $t = 15$, $n = 1000$

Figure 5: Analisi di scalabilità per $t = 15$ e $n = 1000$.

Commento: Nei test più grandi ($n = 1000$ e $t = 15$) si vede chiaramente dai grafici che i B-alberi richiedono molti meno accessi a disco rispetto agli ABR, soprattutto nelle operazioni sequenziali, confermando l'efficienza della struttura bilanciata e la scalabilità per grandi dimensioni dei dataset.

4.3 Osservazioni

Dall'analisi complessiva dei grafici emerge che:

- Nei B-alberi si può notare che se aumento il grado t, l'albero diventa meno profondo e di conseguenza servono meno accessi a disco.
- L'inserimento casuale mostra più variabilità rispetto a quello sequenziale, ma generalmente il tutto rimane coerente con la teoria.
- Con $n = 1000$ è possibile notare la vera forza dei B-alberi; la *scalabilità*. Sia l'inserimento sia la ricerca rimangono efficienti anche con grandi dataset, specialmente con valori maggiori di t.

5 Tesi e sintesi finale

5.1 Risultati principali

Dall'analisi sperimentale condotta nel ***capitolo 4***, possiamo dire ciò sulle prestazioni degli accessi a disco:

1. **B-alberi complessivamente migliori**: I B-alberi riescono a dimostrare prestazioni superiori agli ABR in termini di accessi a disco sia nel caso medio che nel caso peggiore del tipo $O(\log_t n)$, là dove l'ABR soffre maggiormente con una complessità $O(n)$.
2. **Consistenza delle prestazioni**: I B-alberi offrono prestazioni consistenti in qualsiasi caso, indipendentemente dal tipo di inserimento delle chiavi, mentre gli ABR sono influenzati dall'ordine di inserimento.
3. **Scalabilità**: Il vantaggio dei B-alberi aumenta all'aumentare della dimensione del dataset, confermando la sua superiorità nel caso di grandi volumi di dati.

5.2 Implicazioni pratiche

5.2.1 Quando utilizzare B-alberi

I B-alberi sono perfetti quando ho:

- Applicazioni con frequenti accessi a memoria secondaria
- Casi dove l'ordine di inserimento non può essere controllato
- Applicazioni che richiedono prestazioni consistenti e prevedibili

5.2.2 Quando considerare ABR

Gli ABR, a differenza dei B-alberi, risultano appropriati solo in contesti *limitati*:

- Dataset di piccole dimensioni residenti completamente in memoria primaria
- Casi dove si può garantire un inserimento bilanciato

5.3 Conclusioni finali

Dai risultati ottenuti, possiamo concludere che i B-alberi sono migliori rispetto agli ABR, soprattutto in casi che richiedono l'accesso a memoria secondaria. Infatti i B-alberi offrono:

1. **Efficienza**: Riduzione significativa degli accessi a disco
2. **Consistenza**: Prestazioni consistenti indipendentemente dal tipo di inserimento
3. **Scalabilità**: Vantaggio crescente al crescere della dimensione dei dati e del grado *t*
4. **Prevedibilità**: Complessità garantita O($\log_t n$) in tutti i casi

In conclusione, per applicazioni che gestiscono grandi volumi di dati su memoria secondaria, i B-alberi costituiscono la scelta migliore rispetto agli ABR.

References

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