UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laboratorio di Algoritmi e Strutture Dati

Autore:
Federico Marra

N° Matricola:
7025997

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

Elenco delle figure

1 Introduzione generale

1.1 Assegnazione del progetto

Qui di seguito verrà esplicato il progetto che mi è stato assegnato per il superamento della parte di Laboratorio di Algoritmi e Strutture Dati.

- Alberi Binari di Ricerca vs B-Alberi

1.2 Breve descrizione dello svolgimento degli esercizi

Per ogni esercizio suddivideremo la sua descrizione in 4 parti fondamentali:

- **Spiegazione teorica del problema** : qui è dove si descrive il problema che andremo ad affrontare in modo teorico partendo dagli assunti del libro di Algoritmi e Strutture Dati e da altre fonti.
- **Documentazione del codice** : in questa parte spieghiamo come il codice dell'esercizio viene implementato
- **Descrizione degli esperimenti condotti** : partendo dal codice ed effettuando misurazioni varie cerchiamo di verificare le ipotesi teoriche
- **Analisi dei risultati sperimentali** : dopo aver svolto i vari esperimenti riflettiamo sui vari risultati ed esponiamo una tesi

1.3 Specifiche della piattaforma di test

La piattaforma di test sarà la stessa per ogni esercizio che vedremo. Partiamo dall'hardware del computer fondamentale da conoscere per questo esercizio:

- **Modello** : Apple MacBookAir 8,1
- **CPU** : Intel 1,6 GHz Intel Core i5 dual-core
- **RAM** : 8 GB 2133 MHz LPDDR3
- **SSD** (interno) : APPLE SSD AP0256M 250,69GB

Il linguaggio di programmazione utilizzato sarà Python e la piattaforma in cui il codice è stato scritto e 'girato' è l'IDE **PyCharm Professional 2023.3.2**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

Parte I
B-Alberi vs Alberi Binari di Ricerca - Esercizio 1

Esercizio 1

- Vogliamo analizzare le differenze tra B-Alberi e Alberi Binari di Ricerca
- I B-alberi (capitolo 18 libro di testo) 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
  - 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)
- Per fare questo dovremo:
  - Scrivere i programmi Python (no notebook) che:
    - implementino quanto richiesto
    - eseguono un insieme di test che ci permettano di comprendere vantaggi e svantaggi delle diverse implementazioni
  - Svolgere ed analizzare opportuni esperimenti
  - Scrivere una relazione (in LATEX) che descriva quanto fatto

2 Spiegazione teorica del problema - Esercizio 1

2.1 Introduzione

In questo parte del progetto si descrive l'implementazione degli alberi binari di ricerca e dei b-alberi e li mettiamo a confronto valutando la complessità computazionale di alcuni dei metodi che hanno logicamente in comune. Ci concentreremo, in particolare, sui metodi di inserimento e ricerca perché sono fondamentalmente i metodi più importanti e con ogni probabilità quelli maggiormente utilizzati quando si fa riferimento a degli alberi. Non confronteremo invece il metodo di cancellazione il quale è difficilmente implementabile nei b-alberi.

2.2 Aspetti fondamentali

Nell'esperimento considereremo caso quello medio, ovvero una serie di numeri non ordinati, dunque in ordine casuale che vanno da n=step con $step = 50$ a "'step*nTests"', con $nTests = 20$, arriverà fino a $50 * 20 = 1000$. Dunque prenderemo la costruzione in modo casuale dell'albero (essenzialmente vuol dire che il numero che andrò ad inserire di volta in volta è casuale). Da qui in poi ci riferiremo agli Alberi Binari di Ricerca come ABR e ai B-Alberi come BA. Useremo anche alcune parti fondamentali della nomenclatura della teoria della complessità computazionale come $O(\cdot)$ e $\Theta(\cdot)$. Ci riferiremo anche ad n come numero di nodi dell'albero ed ad $h = \log_t n$ come altezza dell'albero. Non utilizzeremo altre strutture dati particolari all'infuori degli alberi in questione. Descriviamo brevemente i metodi che andremo ad analizzare in particolare:

- Inserimento
  - ABR:
    1. Inizio:
       L'inserimento inizia dalla radice dell'albero.
    2. Ricerca della Posizione:
       Si procede verso il basso confrontando la chiave da inserire con le chiavi nei nodi dell'albero.
       Se la chiave da inserire è minore della chiave del nodo corrente, si continua la ricerca

nel sottoalbero sinistro.
Se invece la chiave è maggiore, si continua la ricerca nel sottoalbero destro.

3. Inserimento:
La chiave da inserire viene posizionata in un nuovo nodo nel punto in cui termina la ricerca, in un nodo vuoto o sostituendo un nodo esistente.
Se il nodo di destinazione ha già un figlio a sinistra o a destra, la chiave viene inserita come figlio sinistro o destro, mantenendo l'ordinamento del ABR.
4. Fine: L'inserimento è completato e non c'è nessun ribilanciamento dell'albero.

– BA:

1. Inizio:
L'inserimento inizia nella radice dell'albero.
2. Ricerca del Nodo Adeguato:
Inizia la ricerca del nodo appropriato per l'inserimento. Si discende nell'albero seguendo i rami in base ai valori chiave fino a raggiungere una foglia o un nodo interno.
3. Inserimento nel Nodo:
Se il nodo in cui ci troviamo non è completamente pieno, possiamo semplicemente inserire la nuova chiave nel nodo mantenendo l'ordinamento.
Se il nodo è pieno, viene diviso in due nodi più piccoli. La chiave mediana viene promossa al genitore e i due nuovi nodi contengono le chiavi rimanenti divise in modo equo.
4. Propagazione delle Promozioni:
Se la promozione raggiunge la radice, la radice viene divisa generando un nuovo livello nell'albero.
5. Aggiornamento dei Puntatori:
Durante l'inserimento e la divisione, i puntatori dei nodi vengono aggiornati per riflettere la nuova struttura dell'albero.
6. Fine:
L'inserimento è completato e c'è ribilanciamento dell'albero ad ogni elemento inserito.

• Ricerca

– ABR:

1. Inizio:
La ricerca inizia dalla radice dell'albero.
2. Confronto con il Nodo Corrente:
Si confronta la chiave da cercare con la chiave del nodo corrente.
Se la chiave corrisponde, la ricerca è terminata e il nodo è stato trovato.
Se la chiave è minore della chiave del nodo corrente, si sposta nel sottoalbero sinistro (poiché tutte le chiavi nel sottoalbero sinistro sono minori di quella del nodo corrente).
Se la chiave è maggiore, si sposta nel sottoalbero destro (poiché tutte le chiavi nel sottoalbero destro sono maggiori di quella del nodo corrente).
3. Ricerca Ricorsiva:
Il passo 2 è ripetuto in modo ricorsivo nel sottoalbero scelto.
Questo processo continua fino a quando viene trovato il nodo con la chiave cercata o si raggiunge una foglia dell'albero senza trovare la chiave.
4. Fine:
La ricerca termina quando viene trovato il nodo con la chiave cercata o quando si raggiunge una foglia senza trovare la chiave cercata.

– BA:

1. Inizio:
La ricerca inizia dalla radice dell'albero.
2. Confronto con le Chiavi:
Si confronta la chiave da cercare con le chiavi presenti nel nodo corrente.
Se la chiave è presente nel nodo, la ricerca è terminata, e viene restituito il valore associato alla chiave.

Se la chiave è minore della chiave minima nel nodo, si scende nel sottoalbero sinistro.
Se la chiave è maggiore della chiave massima nel nodo, si scende nel sottoalbero destro.
Se la chiave è compresa tra le chiavi minima e massima, si scende nel sottoalbero corrispondente.

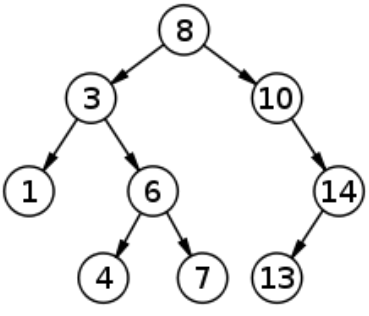
3. Ricerca Ricorsiva: Il passo 2 viene ripetuto in modo ricorsivo nel sottoalbero selezionato.
   Questo processo continua fino a quando viene trovata la chiave nel nodo o si raggiunge una foglia dell'albero senza trovare la chiave cercata.
4. Fine:
   La ricerca termina quando viene trovato il nodo con la chiave cercata o quando si raggiunge una foglia senza trovare la chiave cercata.

Ora vorrei descrivere le particolarità di queste due tipologia di alberi partendo da ABR. Le proprietà di un ABR (esempio in figura 1a) sono:

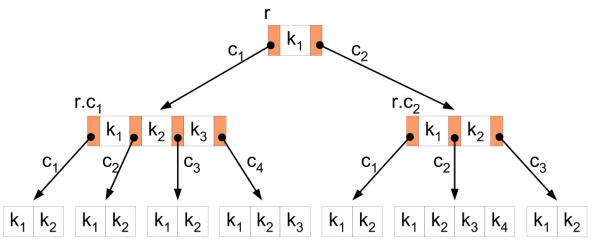
1. Il sottoalbero sinistro di un nodo x contiene soltanto i nodi con chiavi minori della chiave del nodo x.
2. Il sottoalbero destro di un nodo x contiene soltanto i nodi con chiavi maggiori della chiave del nodo x.
3. Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due ABR.

Le proprietà di un BA (esempio in figura 1b) sono:

1. Ogni nodo x ha i seguenti attributi:
   - $x.n$ è il numero di chiavi correttamente memorizzate nel nodo x.
   - Le $x.n$ chiavi stesse, $x.key_1, ..., x.key_{x.n}$ sono memorizzate in ordine non decrescente, in modo che: $x.key_1 \leq x.key_2 \leq ... \leq x.key_{x.n}$.
   - $x.leaf$ è un valore booleano che è TRUE se x è una foglia, e invece è FALSE se x è un nodo interno.
2. Ogni nodo interno x contiene anche $x.n + 1$ puntatori $x.c_1, ..., x.c_{x.n+1}$ ai suoi figli. I nodi foglia non hanno figli, quindi i loro attributi c_i non sono definiti.
3. Le chiavi $x.key_i$ separano gli intervalli delle chiavi memorizzate in ciascun sottoalbero: se k_i è una chiave qualsiasi memorizzata nel sottoalbero con radice $c_i[x]$ allora:
   $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq ... \leq x.key_{x.n} \leq x.key_{x.n+1}$
4. Tutte le foglie hanno la stessa profondità, che è l'altezza h dell'albero.
5. Ci sono limiti superiori e inferiori al numero di chiavi che un nodo può contenere. Questi limiti possono essere espressi in termini di un intero $t \geq 2$ chiamato **grado minimo** del BA:
   - Ogni nodo, tranne la radice deve avere almeno $t - 1$ chiavi- Ogni nodo interno, tranne la radice, quindi almeno t figli. Se l'albero non è vuoto la radice deve avere almeno una chiave.
   - Ogni nodo può contenere al massimo $2t - 1$ chiavi. Quindi un nodo interno può avere al massimo $2t$ figli. Diciamo che un nodo è **pieno** se contiene esattamente $2t - 1$ chiavi.



(a) Albero binario di ricerca



(b) B-Albero

Figura 1: Tipologie di alberi

2.3 Assunti ed ipotesi

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 $O(h)$ quindi le operazioni elementari svolte su questo tipo di albero richiedono in media $\Theta(h)$. Per vedere la complessità degli algoritmi più importanti di ABR basati sul caso peggiore e sul caso medio si richiama alla figura 2 facendo particolare attenzione ai metodi in rosso che sono quelli su cui andremo a svolgere gli esperimenti.

	Complessità al caso peggiore	Complessità al caso medio
Spazio	$\Theta(n)$	$\Theta(n)$
Inserimento	$O(n)$	$O(h)$
Ricerca	$O(n)$	$O(h)$
Cancellazione	$O(n)$	$O(h)$

Figura 2: Complessità degli algoritmi di ABR

Per un B-albero, le operazioni di base richiedono un tempo proporzionale all'altezza dell'albero. La struttura di un B-albero permette di mantenere un bilanciamento dell'altezza e di ridurre la complessità delle operazioni rispetto a un albero binario di ricerca. Le operazioni di base su un B-albero richiedono in media $\Theta(\log_t n)$, dove $log_t n = h$ è l'altezza dell'albero e n è il numero di chiavi nell'albero. Qui di seguito sono presentate le complessità degli algoritmi più importanti di un B-albero, sia nel caso peggiore che nel caso medio.

	Complessità al caso peggiore	Complessità al caso medio
Spazio	$\Theta(n)$	$\Theta(n)$
Inserimento	$O(\log n)$	$O(\log n)$
Ricerca	$O(\log n)$	$O(\log n)$
Cancellazione	$O(\log n)$	$O(\log n)$

Figura 3: Complessità degli algoritmi di B-albero

Il nostro obiettivo in questo test è verificare sperimentalmente la veridicità delle varie complessità descritte nelle figure 2 e 3 e capire sotto quali condizioni un albero è più conveniente di un altro confrontandoli, a parità di spazio, in base al tempo reale che ci mettono a completarsi.

3 Documentazione del codice - Esercizio 1

3.1 Scelte implementative

Con la seguente descrizione elencheremo i vari attributi delle classi che permettono l'implementazione degli alberi binari di ricerca e dei b-alberi.

- nessun padre
- colore nero
- chiave -1

BinaryTreeNode

- **key** : contiene il valore del nodo.
- **left** : è un puntatore al figlio sinistro.
- **right** : è un puntatore al figlio destro.
- **p** : è un puntatore al padre sinistro.

BinaryTree

- **node** : contiene nodo.
- **root** : è un puntatore alla radice.
- **node_read** : contarore che tiene nota di quante letture vengono fatte sull'albero.
- **node_written** : contarore che tiene nota di quante scritture vengono fatte sull'albero.

BTreeNode

- **leaf** : booleano vero se il nodo è una foglia.
- **keys** : lista con valori nel nodo.
- **child** : lista con puntatori ai figli.
- **n** : numero chiavi memorizzate nel nodo.

BTree

- **root** : è un puntatore alla radice*p*.
- **t** : numero minimo chiavi memorizzate in ogni nodo *p*.
- **node_read** : contarore che tiene nota di quante letture vengono fatte sull'albero *p*.
- **node_written** : contarore che tiene nota di quante scritture vengono fatte sull'albero*p*.

3.2 Descrizione dei metodi implementati

In questa parte descriverò le funzionalità di ogni metodo delle classi degli alberi.

- **BinaryTree**
  - **get_node_read()** : restituisce il numero di letture eseguite sull'albero.
  - **get_node_written()** : restituisce il numero di scritture eseguite sull'albero.
  - **search(x, key)** : in modo ricorsivo visita e quando trova una chiave ==key, ritorna quel nodo.
  - **insert(key)** : in modo iterativo va avanti nell'albero fino a ricercare la foglia più vicina a key e se è maggiore inserisce a destra, e se minore a sinistra
- **BinaryTree**
  - **get_node_read()** : restituisce il numero di letture eseguite sull'albero.

  - **get_node_written()** : restituisce il numero di scritture eseguite sull'albero.
  - **search(key)** : in modo ricorsivo visita e quando trova una chiave == key, ritorna quel nodo e la sua profondità.
  - **split_child(x, i)** : divide un nodo figlio troppo grande in due nodi più piccoli durante un'operazione di inserimento.
  - **insert(key)** : inserisce la chiave e usando **split child(x, i)** e **insert nonfull(key)** mantiene l'albero bilanciato.
  - **insert_nonfull(key)** : viene chiamata quando si inserisce una chiave in un nodo non completamente pieno.

- **Generazione dei grafici**
  - **random_array(n)** : restituisce un'array ordinato casualmente con n valori da 1 a n.
  - **test_insert(BinaryTree, BTree, array)** : restituisce più valori contenenti i tempi di esecuzione in serie del metodo **insert** per ABR e BA, e delle rispettive occorrenze di lettura e scrittura.
  - **test_search(BinaryTree, BTree, array)** : restituisce più valori contenenti i tempi di esecuzione in serie del metodo **search** per ABR e BA, e delle rispettive occorrenze di lettura e scrittura.
  - **draw_table(data, title, colorHead="orange", colorCell="yellow", filename="table")** : genera una tabella con la tupla data, e come stile della tabella prende i parametri in ingresso, e salva il tutto in memoria con il nome filename.
  - **draw_side_graphs(left data, right data, plot title, filename)** : genera due grafici l'uno accanto all'altro, e salva il tutto in memoria con il nome filename.
  - **draw_comparison_graphs(data1, data2, title, filename)** : genera grafico con entrambi i data nello stesso subplot, e salva il tutto in memoria con il nome filename.

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali - Esercizio 1

4.1 Dati utilizzati

L'esperimento che ho svolto si divide prima di tutto per il numero di nodi che sono andato ad inserire nei due alberi. Quindi divideremo l'esperimento in 4 parti:

- Alberi con $t =$ **100**
- Alberi con $t =$ **250**
- Alberi con $t =$ **1000**

Prendendo il caso della randomizzazione ho deciso di fare un test per verificare se il range in cui prendo i numeri influenzi il tempo che ci mette per lo svolgimento dell'algoritmo. Questo caso è stato testato con gli alberi da 1000 elementi per avere una miglior idea delle tempistiche. I range di valori che ho usato sono i seguenti:

50-1000, con uno step di 50 Quindi ho creato 20 array su cui testare il tutto, al variare della **t**

4.2 Misurazioni

I metodi di calcolo dei tempi nei metodi **test_insert** e **test_search** sono di due tipi. Innanzitutto dobbiamo prendere il tempo prima dell'esecuzione del metodo e quello dopo la sua esecuzione:

```
def test_search(BinaryTree, BTree, array):
    print(f"\nRicerca di {len(array)} elementi:")

    timesBin = 0
    timesBT = 0
    for i in range(len(array)):
        start = timer()
        BinaryTree.search(BinaryTree.root, array[i])
        end = timer()
        timesBin += (end - start) * 1000

        start = timer()
        BTree.search(array[i])
        end = timer()
        timesBT += (end - start) * 1000

    readBin = BinaryTree.get_node_read()
    writtenBin = BinaryTree.get_node_written()
    readBT = BTree.get_node_read()
    writtenBT = BTree.get_node_written()

    print(f"Albero binario di ricerca: {timesBin} ms")
    print(f"B-albero: {timesBT} ms")
    print(f"Albero binario di ricerca: {readBin} letture, {writtenBin} scritture")
    print(f"B-albero: {readBT} letture, {writtenBT} scritture")

    return timesBin, timesBT, readBin, readBT, writtenBin, writtenBT

def test_insert(BinaryTree, BTree, array):
    print(f"\nInserimento di {len(array)} elementi:")
    timesBin = 0
    timesBT = 0
    for i in range(len(array)):
        start = timer()
        BinaryTree.insert(BinaryTreeNode(array[i]))
        end = timer()
        timesBin += (end - start) * 1000
```

```
        start = timer()
        BTree.insert(array[i])
        end = timer()
        timesBT += (end - start) * 1000

    readBin = BinaryTree.get_node_read()
    writtenBin = BinaryTree.get_node_written()
    readBT = BTree.get_node_read()
    writtenBT = BTree.get_node_written()

    print(f"Albero binario di ricerca: {timesBin} ms")
    print(f"B-albero: {timesBT} ms")
    print(f"Albero binario di ricerca: {readBin} letture, {writtenBin} scritture")
    print(f"B-albero: {readBT} letture, {writtenBT} scritture")

    return timesBin, timesBT, readBin, readBT, writtenBin, writtenBT
```

Per calcolare i tempi salvo in **start** il tempo prima dell'operazione, e in **end**. La loro sottrazione mi darà il tempo trascorso in nanosecondi, che moltiplicherò per 1000 per ottenerlo in millisecondi.

4.3 Risultati sperimentali e commenti analitici

Guardo prima i tempi di esecuzione:

4.3.1 Alberi a confronto nei tempi con $t = 100$

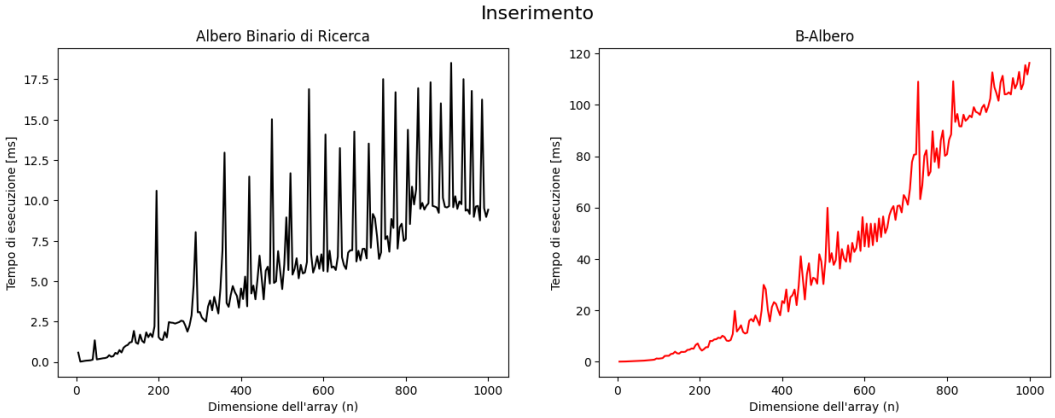


Figura 4: Grafico tempi dell'Inserimento con $t = 100$

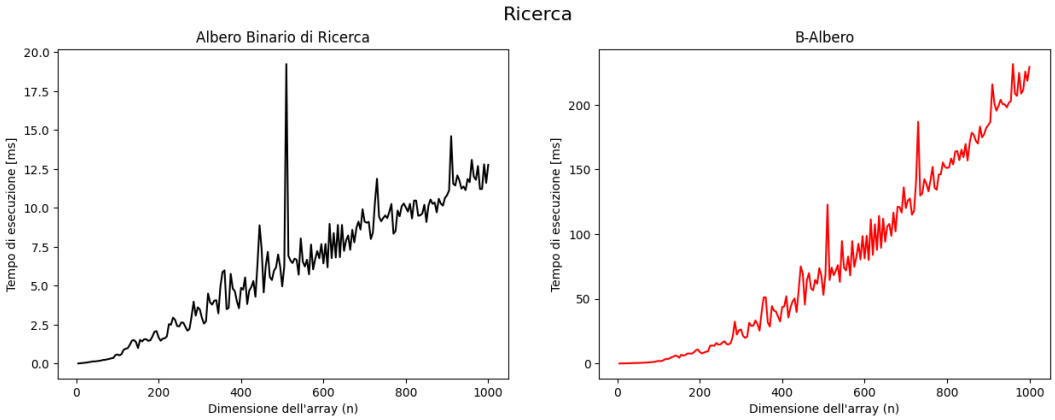
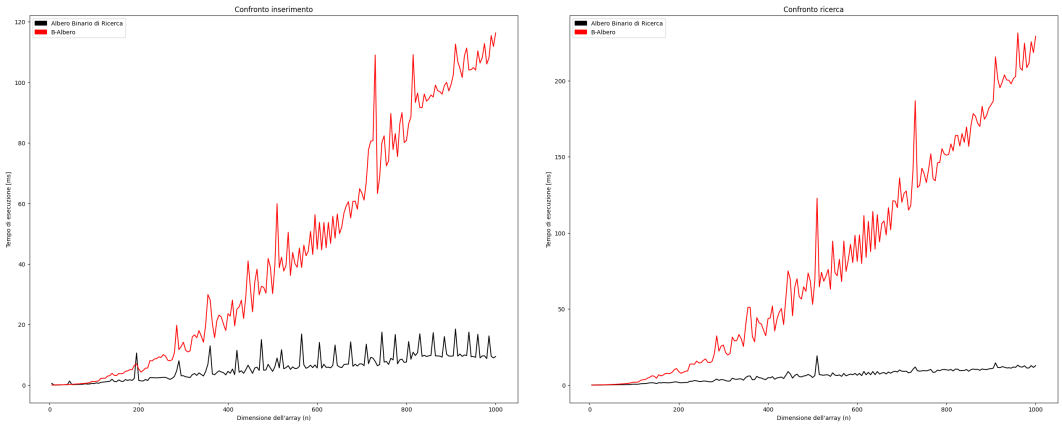


Figura 5: Grafico tempi della Ricerca con $t = 100$



(a) Inserimento (b) Ricerca

Figura 6: Grafici di confronto dei tempi con $t = 100$

4.3.2 Alberi a confronto nei tempi con $t = 250$

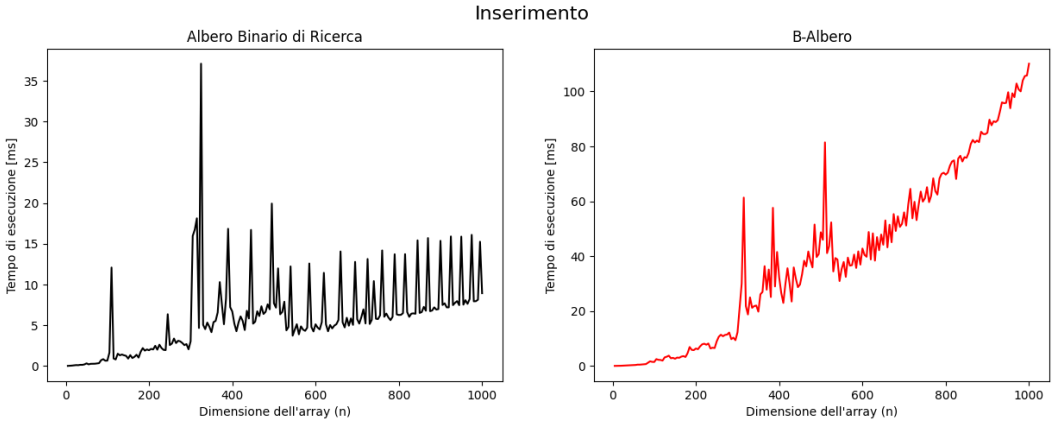


Figura 7: Grafico tempi dell'Inserimento con $t = 250$

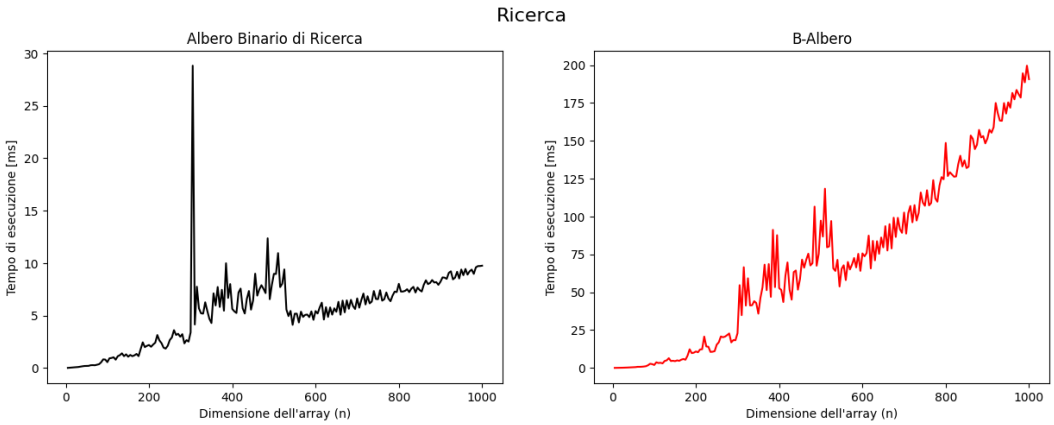
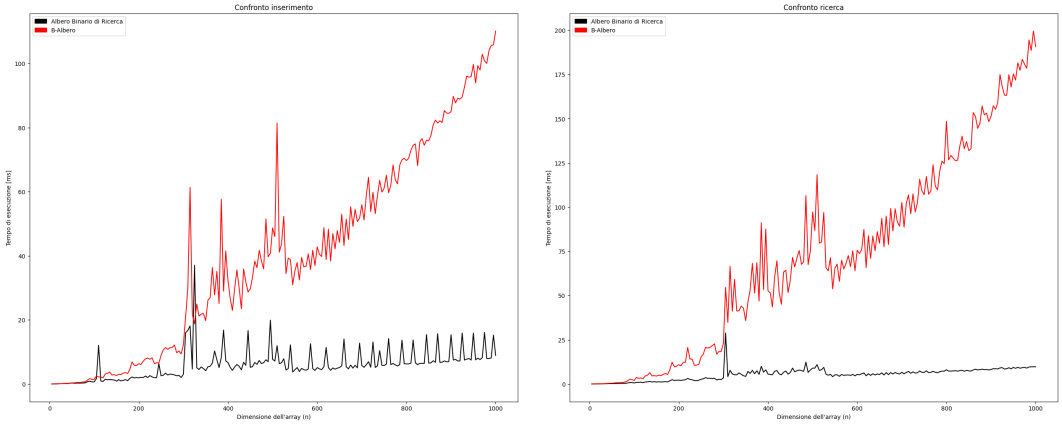


Figura 8: Grafico tempi della Ricerca con $t = 250$



(a) Inserimento (b) Ricerca

Figura 9: Grafici di confronto dei tempi con $t = 250$

4.3.3 Alberi a confronto nei tempi con $t = 1000$

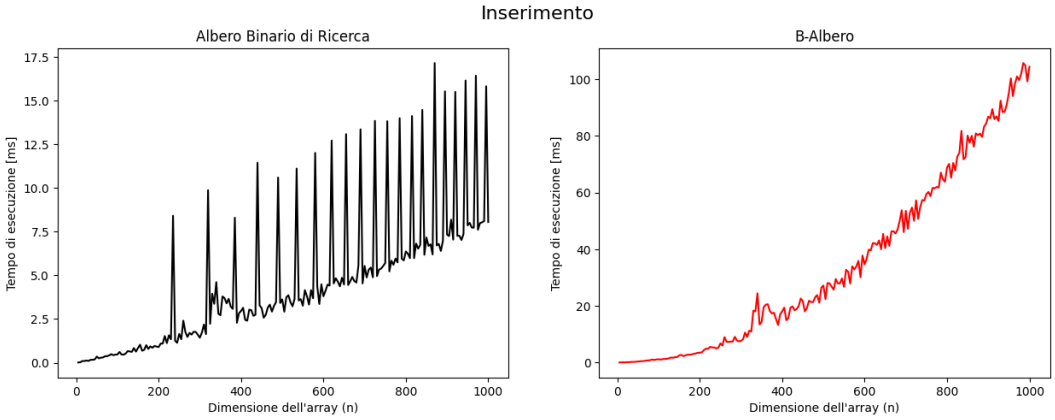


Figura 10: Grafico tempi dell'Inserimento con $t = 1000$

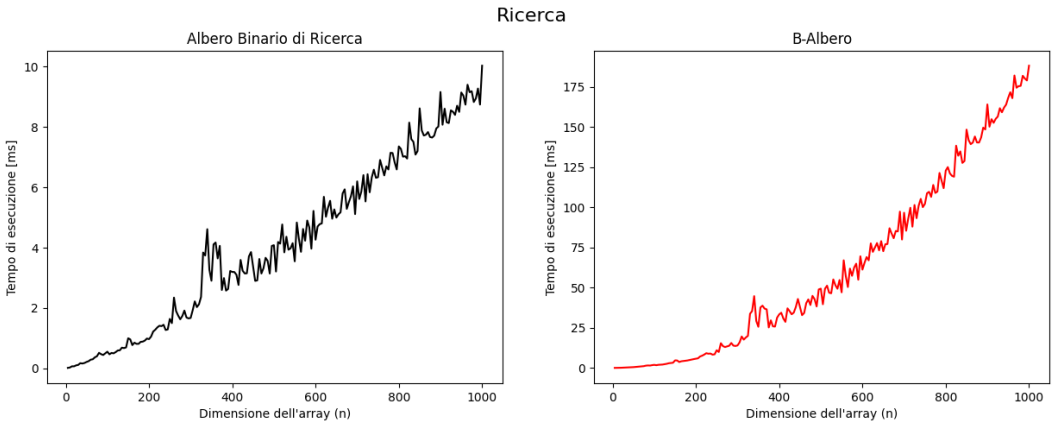
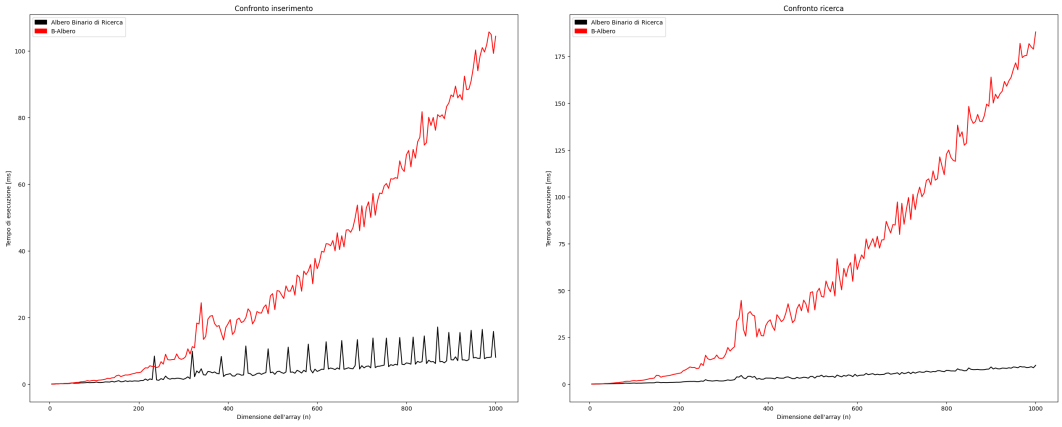


Figura 11: Grafico tempi della Ricerca con $t = 1000$



(a) Inserimento (b) Ricerca

Figura 12: Grafici di confronto dei tempi con $t = 1000$

Ora invece guardo il numero di letture e scritture:

4.3.4 Alberi a confronto nel numero di letture e scritture con $t = 100$

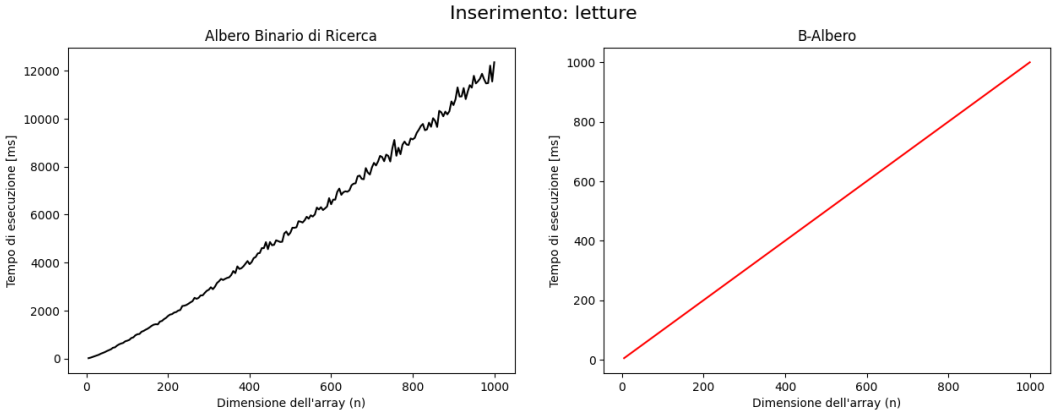


Figura 13: Grafico delle letture dell'Inserimento con $t = 100$

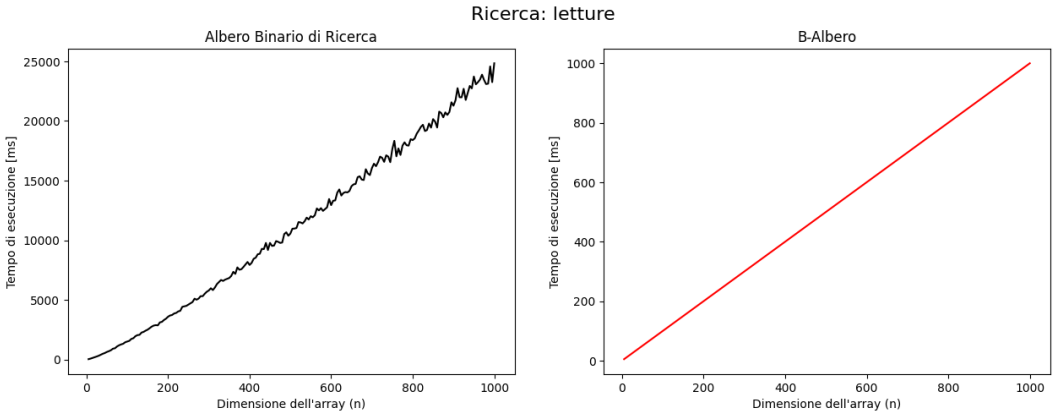


Figura 14: Grafico delle letture della Ricerca con $t = 100$

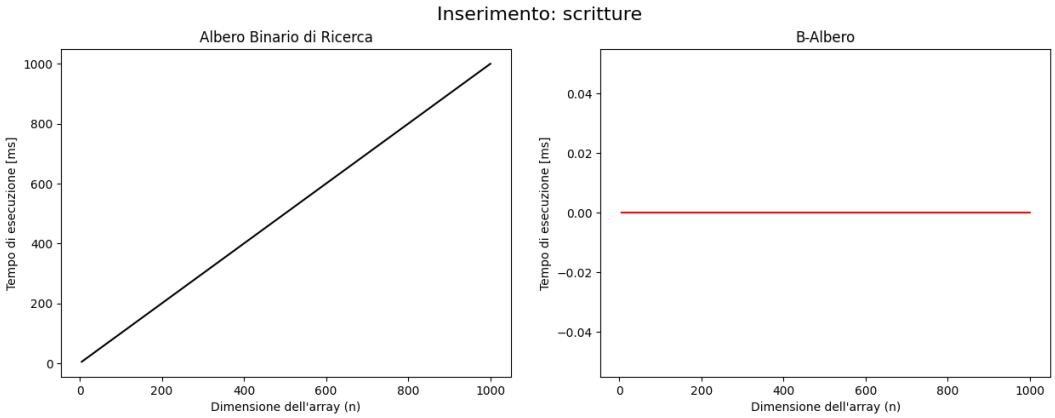


Figura 15: Grafico delle scritture dell'Inserimento con $t = 100$

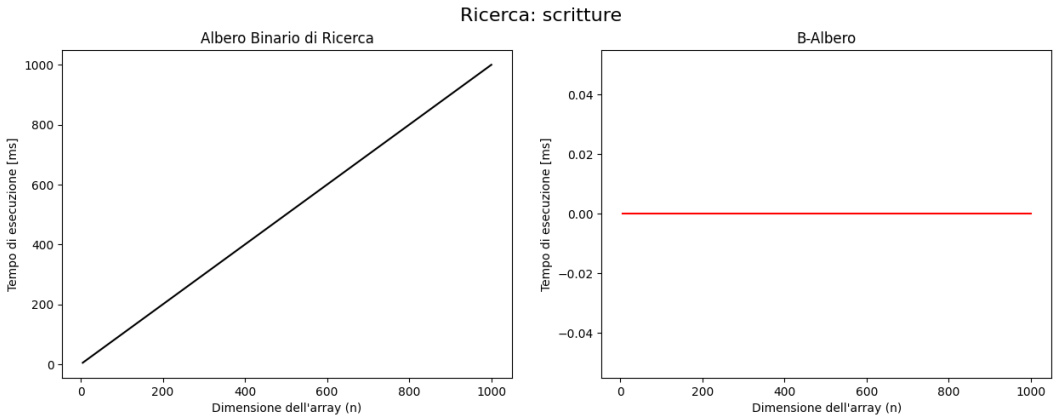
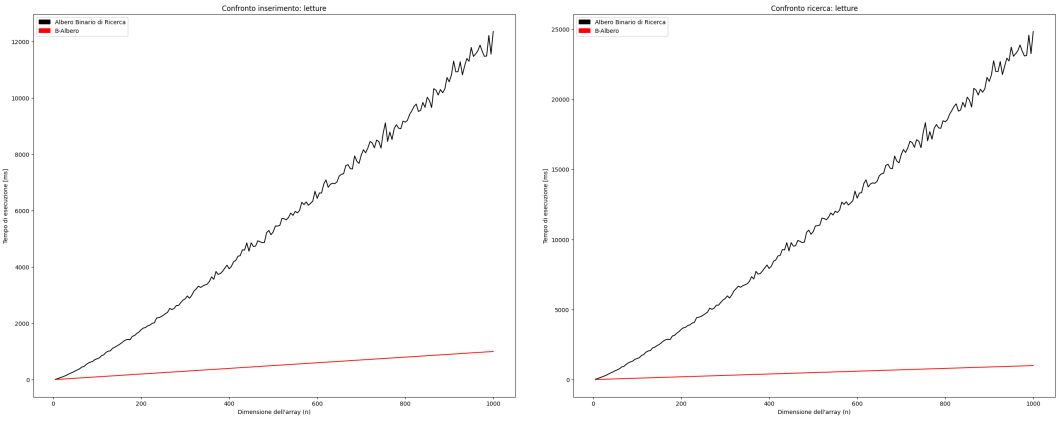
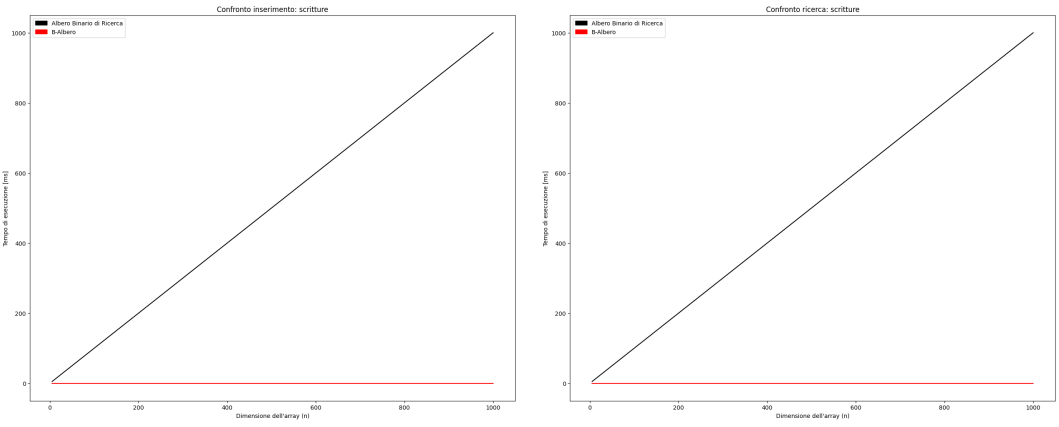


Figura 16: Grafico delle scritture della Ricerca con $t = 100$



(a) Inserimento (b) Ricerca

Figura 17: Grafici di confronto delle letture con $t = 100$



(a) Inserimento (b) Ricerca

Figura 18: Grafici di confronto delle scritture con $t = 100$

4.3.5 Alberi a confronto nel numero di letture e scritture con $t = 250$

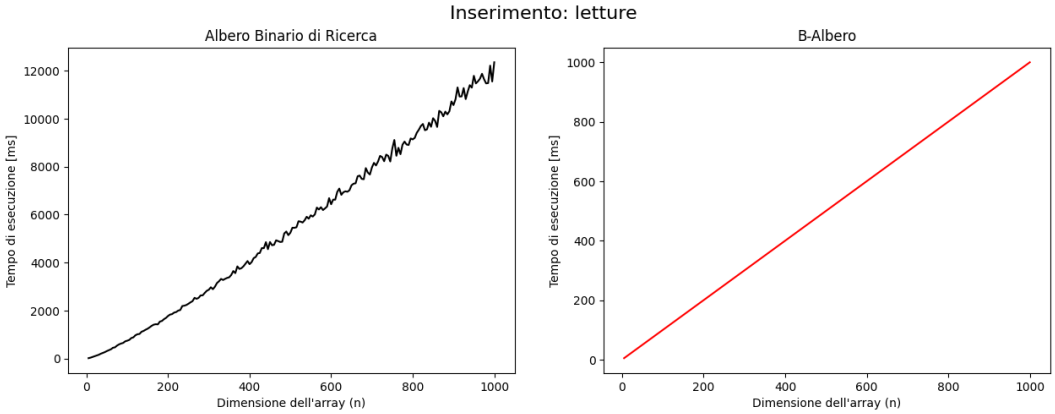


Figura 19: Grafico delle letture dell'Inserimento con $t = 250$

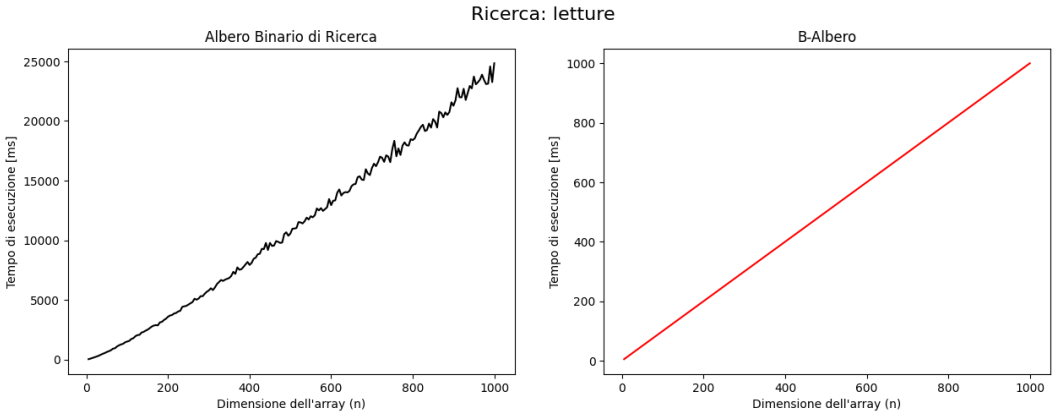


Figura 20: Grafico delle letture della Ricerca con $t = 250$

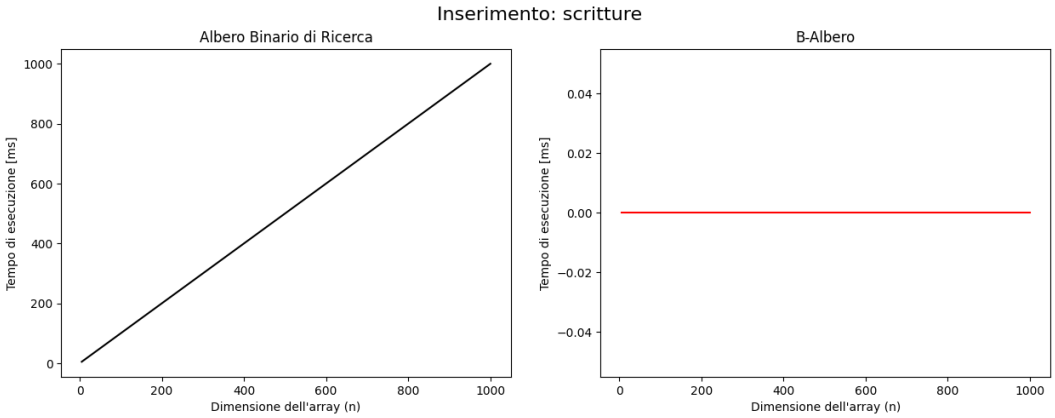


Figura 21: Grafico delle scritture dell'Inserimento con $t = 250$

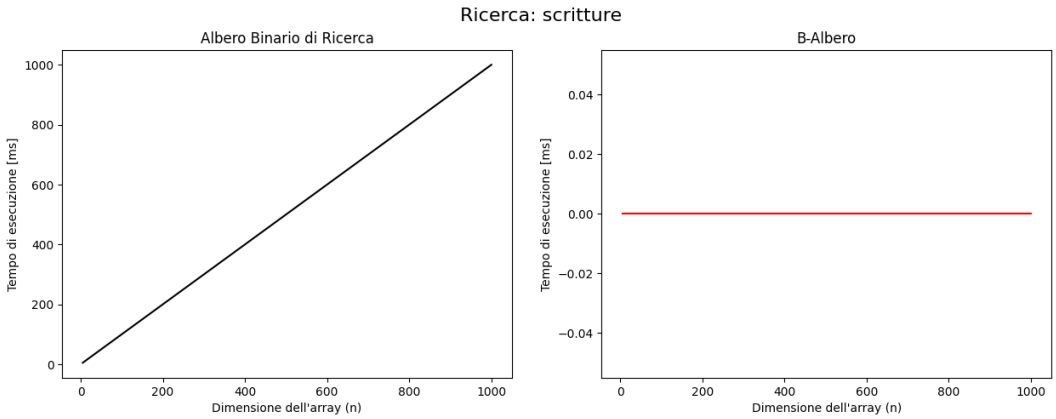
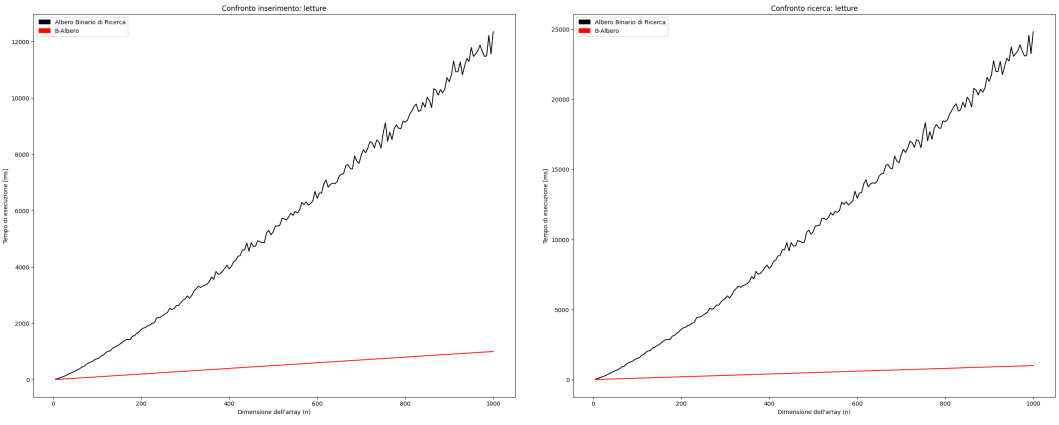
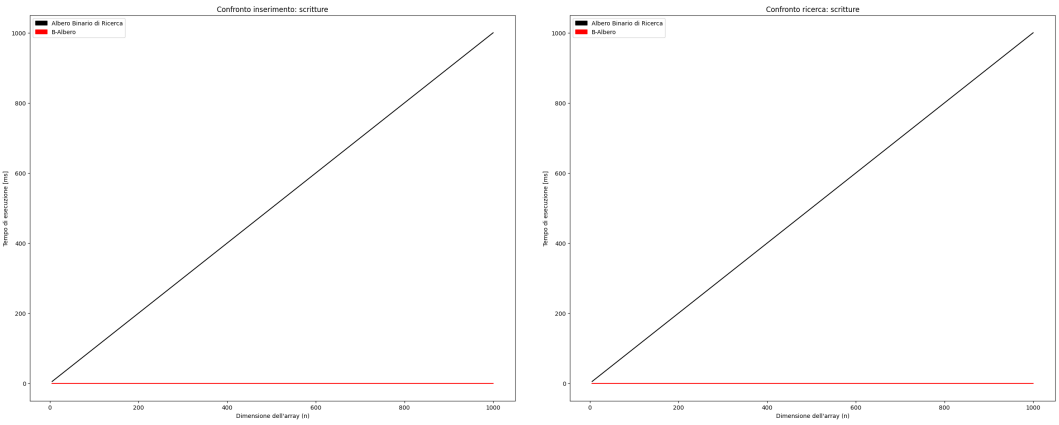


Figura 22: Grafico delle scritture della Ricerca con $t = 250$



(a) Inserimento (b) Ricerca

Figura 23: Grafici di confronto delle letture con $t = 250$



(a) Inserimento (b) Ricerca

Figura 24: Grafici di confronto delle scritture con $t = 250$

4.3.6 Alberi a confronto nel numero di letture e scritture con $t = 1000$

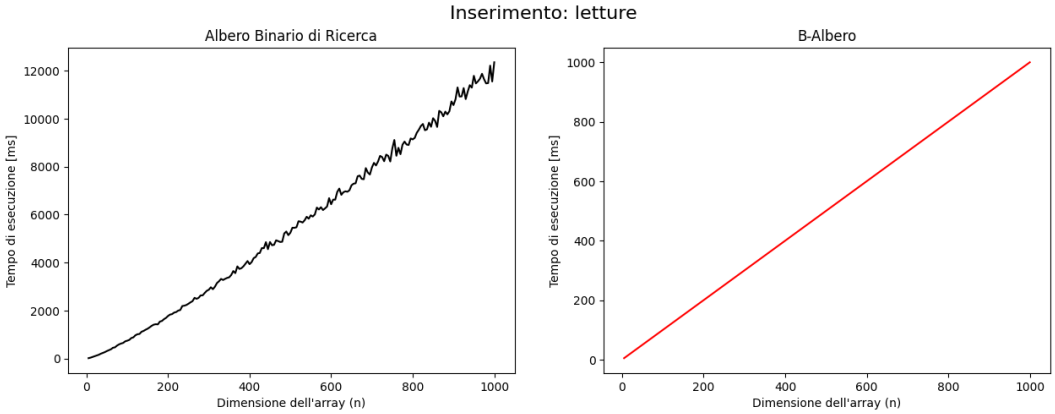


Figura 25: Grafico delle letture dell'Inserimento con $t = 1000$

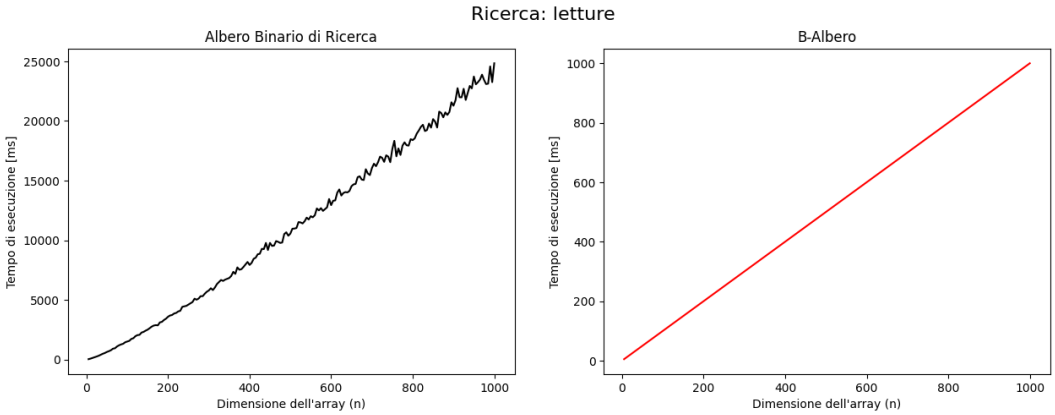


Figura 26: Grafico delle letture della Ricerca con $t = 1000$

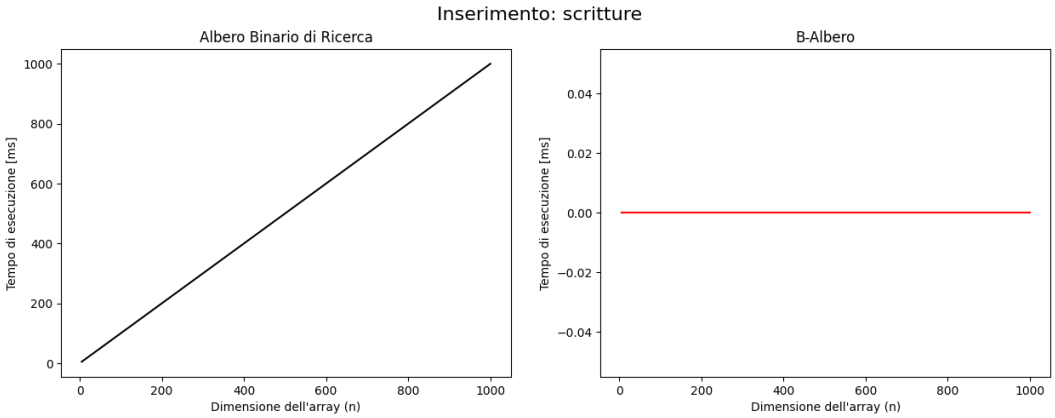


Figura 27: Grafico delle scritture dell'Inserimento con $t = 1000$

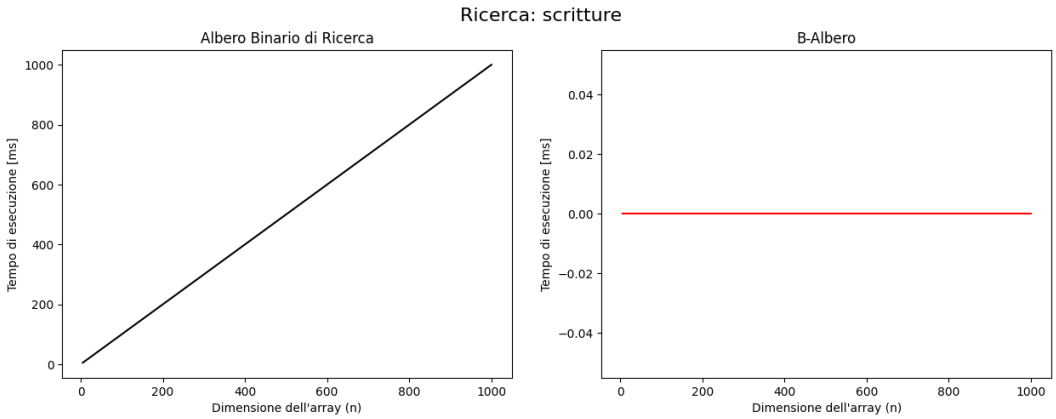
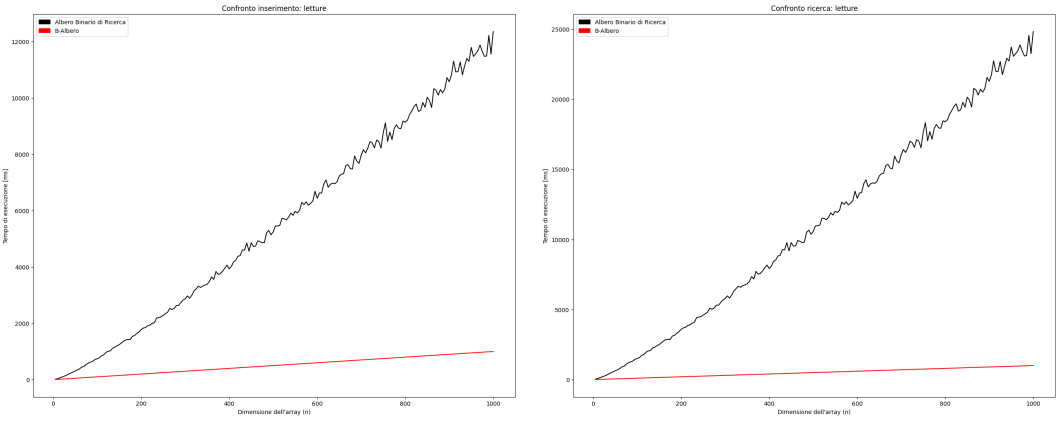
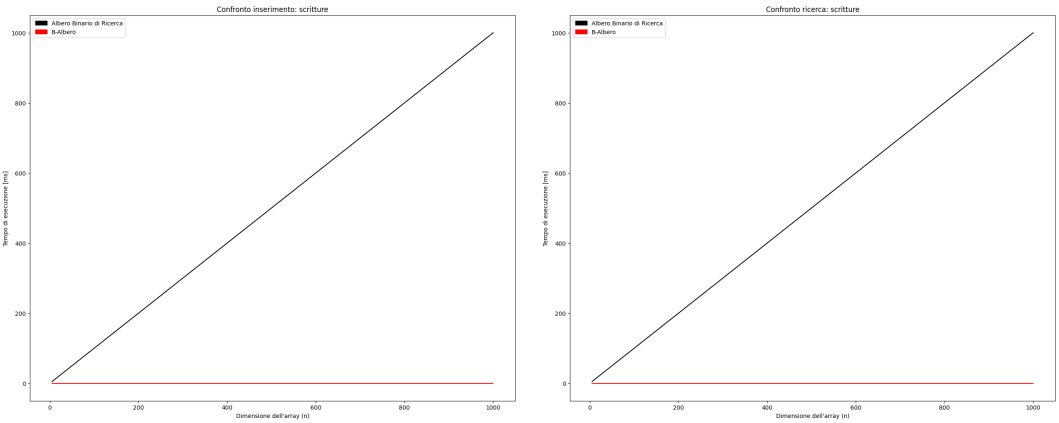


Figura 28: Grafico delle scritture della Ricerca con $t = 1000$



(a) Inserimento (b) Ricerca

Figura 29: Grafici di confronto delle letture con $t = 1000$



(a) Inserimento (b) Ricerca

Figura 30: Grafici di confronto delle scritture con $t = 1000$

4.4 Tesi e sintesi finale

Possiamo dare le seguenti conclusioni:

- Complessità degli algoritmi:
  I metodi di inserimento e ricerca nei B-alberi hanno una complessità asintotica ottimale di $O(h) = O(\log_t n)$, dove n è il numero di nodi nell'albero. Questa complessità è mantenuta anche nel caso peggiore, garantendo prestazioni efficienti.

- Tempo di esecuzione:
  Nel caso di un inserimento e la ricerca in catena nei B-alberi mantengono una complessità temporale ottimale di $O(log_t n * nTests)$, simile agli alberi ABR. Tuttavia, nei casi più complessi, la complessità è notevolmente migliorata rispetto agli ABR, mantenendo tempi di esecuzione più bassi.

- Influenza del riequilibrio:
  Nei B-alberi, il riequilibrio è intrinseco al processo di inserimento e non richiede passi aggiuntivi come negli ABR. Questo contribuisce a tempi di esecuzione più consistenti e prevedibili per l'inserimento e la ricerca, indipendentemente dal tipo di input.

- Variazione dei tempi con range di chiavi:
  Nel caso di operazioni casuali, i B-alberi possono mantenere tempi di esecuzione stabili anche con variazioni nel range delle chiavi. La struttura bilanciata dei B-alberi riduce l'impatto della distribuzione delle chiavi sull'efficienza delle operazioni.

- Conclusioni generali:
  I B-alberi dimostrano una robusta efficienza nelle operazioni di inserimento e ricerca, con una complessità che scala in modo ottimale con l'aumentare della dimensione dell'albero. Il riequilibrio automatico contribuisce a mantenere prestazioni stabili e prevedibili in una varietà di scenari, rendendo i B-alberi una scelta affidabile per applicazioni che richiedono una gestione efficiente di grandi quantità di dati.

Riferimenti bibliografici

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