
UNIVERSITÀ DEGLI STUDI FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

# Laboratorio di Algoritmi e Strutture Dati

*Autore:*
Niccolò Benedetto

*N° Matricola:*
7024656

*Corso principale:*
Algoritmi e Strutture Dati

*Docente corso:*
Simone Marinai

# Indice



# Elenco delle figure

# 1 Introduzione generale

## 1.1 Scelta dei progetti

Specifica dei progetti assegnati per il superamento dell'esame:

- Btree vs Binary Search Tree in Disk Operations

## 1.2 Breve descrizione dello svolgimento degli esercizi

Per ogni esercizio si suddivide la descrizione in 4 parti fondamentali:

- **Spiegazione teorica del problema**: sezione in cui si affronta da un punto di vista teorico il problema, partendo dagli assunti del libro di testo associato al corso [**Introduzione agli algoritmi e strutture dati**, McGraw Hill - Terza edizione].
- **Documentazione del codice**: si tratta l'implementazione del codice necessario per la creazione delle strutture dati e la risoluzione dell'esercizio.
- **Descrizione degli esperimenti condotti**: si misurano le performances, in riferimento al codice implementato, cercando di verificare le ipotesi teoriche assunte.
- **Analisi dei risultati sperimentali**: sezione in cui vengono formulate le tesi in relazione ai risultati ottenuti.

## 1.3 Specifiche della piattaforma di test

La piattaforma di test sarà la stessa per ogni esercizio assegnato. Le principali caratteristiche hardware di questa sono di seguito riportate:

- **CPU** : Intel(R) Celeron(R) N4120 CPU @ 1.10GHz
- **RAM** : 4 DIMM slots, 8GB total
- **Disco di Memoria** : Teclast BD256GB SHCB-2280 256GB

Il linguaggio di programmazione utilizzato è il linguaggio Python. L'IDE di riferimento è **PyCharm Edu 2021.3.1**, mentre la stesura di questo testo è stata eseguita mediante l'editor online **Overleaf**.

# Parte I
# B Alberi vs Alberi Binari di Ricerca nelle Disk Operations

**Assignment**

- I B-alberi (capitolo 18 libro di testo, 3 ed.) sono strutture dati utilizzate in memoria secondaria in cui ogni nodo può avere molti figli.
  - 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 svolgere il precedente esercizio è necessario:
  - Scrivere i programmi Python (no notebook) che:
    - implementano 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 (come indicato nelle istruzioni nella pagina moodle del corso)

## 2 Spiegazione teorica del problema

### 2.1 Introduzione

Si deve mettere a confronto l'implementazione dei B alberi con quella degli alberi binari di ricerca, valutando la complessità computazionale delle operazioni che hanno in comune, quali ricerca, inserimento e cancellazione. Tale valutazione, come richiesto, viene eseguita in relazione al numero di nodi letti/scritti da parte della struttura dati in oggetto per ogni operazione specificata.

### 2.2 Aspetti fondamentali

Nell'esperimento considereremo come caso peggiore di ogni operazione la catena lineare (set di chiavi ordinate linearmente), mentre come caso medio un set randomico di chiavi. Da questo punto in poi ci riferiremo agli alberi binari di ricerca con la notazione "BST" e ai B alberi con "BT". Nella stesura della relazione ci affideremo svariate volte alla nomenclatura della complessità computazionale, attraverso i simboli $O(\cdot)$ e $\Theta(\cdot)$. Inoltre associeremo le variabili $n$ al numero di nodi dell'albero, $h$ all'altezza dell'albero e $t$ al grado minimo del BT. Non si utilizzano altre strutture dati particolari eccetto quelle appena specificate. Descriviamo brevemente le operazioni che analizzeremo:

- **Inserimento**
  - BST: l'operazione ha inizio dalla radice dell'albero, dunque in maniera ricorsiva viene individuato ogni volta su quale dei due rami spostarsi in relazione al confronto della chiave da inserire e la chiave del nodo corrente. La ricorsione continua finchè non si giunge a un nodo foglia, quindi si crea un nuovo nodo che contenga la chiave da inserire.
  - BT: è un'operazione più laboriosa in quanto si deve stare attenti ogni volta a preservare le proprietà di bilanciamento e altezza logaritmica dell'albero. In particolare si segue la seguente procedura:
    1. Individua il nodo di inserimento:
       - Si parte dalla radice e si scende attraverso l'albero per trovare il nodo foglia in cui la nuova chiave deve essere inserita, confrontando la chiave da inserire con quelle presenti nei nodi per capire in quale sottoalbero scendere.
    2. Inserimento diretto se il nodo non è pieno:

    * Se il nodo foglia trovato ha meno di $(2t-1)$ chiavi, si inserisce la chiave in ordine crescente tra le altre chiavi del nodo. Non è necessario alcun bilanciamento in questo caso.
  3. Suddivisione del nodo se è pieno:
    * Se il nodo foglia in cui si vuole inserire la chiave è già pieno (contiene già $(2t-1)$ chiavi), è necessario suddividere il nodo.
    * Suddivisione del nodo:
      · Si trova la chiave mediana del nodo.
      · Si divide il nodo in due nuovi nodi: uno con le chiavi minori della mediana e l'altro con le chiavi maggiori.
      · La chiave mediana viene promossa al nodo padre.
      · Se il nodo padre è pieno, può rendersi necessaria una suddivisione ricorsiva verso l'alto, fino alla radice.
  4. Creazione di una nuova radice:
    * Se la radice è piena e viene suddivisa, la chiave mediana diventa la nuova radice, e l'albero cresce in altezza di un livello. Questo è l'unico caso in cui l'altezza di un B-albero aumenta.

- **Ricerca**
  - BST: si scende ogni volta nei sottoalberi in base al risultato del confronto tra la chiave del nodo corrente e la chiave da cercare finchè non si raggiuge il nodo che contiene la chiave da cercare. La ricerca parte dalla radice dell'albero.
  - BT: segue un processo ben definito che prevede
    1. Inizia dalla radice: La ricerca inizia sempre dal nodo radice.
    2. Confronta la chiave cercata con le chiavi nel nodo corrente:
      * Se la chiave cercata è presente nel nodo corrente, la ricerca termina con successo.
      * Se la chiave non è presente nel nodo corrente, si identifica il sottoalbero in cui continuare la ricerca. Questo viene fatto in base all'ordinamento delle chiavi nel nodo corrente.
    3. Seleziona il sottoalbero:
      * Ogni nodo in un B-albero contiene $k$ chiavi ordinate, e ha $k+1$ puntatori ai figli. Per trovare il sottoalbero corretto in cui continuare la ricerca, si trova l'intervallo di chiavi in cui ricade la chiave cercata. Per esempio, se nel nodo ci sono le chiavi $[k_1, k_2, \ldots, k_n]$, e la chiave cercata $k$ soddisfa $k_i < k < k_{i+1}$, si scende al $(i+1)$-esimo figlio.
    4. Ricorsione nel sottoalbero:
      * Selezionato il sottoalbero corretto, si ripete il processo (ovvero, si confronta $k$ con le chiavi del nodo e si individua il successivo sottoalbero), procedendo così in modo ricorsivo.
    5. Fallimento della ricerca:
      * La ricerca termina senza successo se si raggiunge un nodo foglia e la chiave non è presente. In questo caso, si restituisce un valore che indica l'assenza della chiave nell'albero.

- **Cancellazione**
  - BST: segue la seguente procedura
    * Caso 1: Nodo foglia
      · Se il nodo da cancellare è una foglia (non ha figli), basta rimuoverlo semplicemente, cioè si imposta il puntatore del nodo genitore a `None`.
    * Caso 2: nodo con un solo figlio
      · Se il nodo da cancellare ha solo un figlio, il nodo viene rimosso e il suo genitore viene aggiornato per puntare al suo unico figlio; in altre parole si sostituisce il nodo da cancellare con il suo unico figlio.
    * Caso 3: nodo con due figli

· Se il nodo da cancellare ha due figli, non è possibile rimuoverlo direttamente. In questo caso, ci sono due strategie comuni:

1. Sostituzione con il successore inorder: si trova il nodo con il valore più piccolo nel sottoalbero destro (successore) e si sostituisce il valore del nodo da cancellare con il valore del successore. In seguito, si rimuove il successore. *Questa sarà la strategia implementata nel codice.*
2. Sostituzione con il predecessore inorder: si trova il nodo con il valore più grande nel sottoalbero sinistro (predecessore) e si sostituisce il valore del nodo da cancellare con il valore del predecessore. Quindi si rimuove il predecessore.

– BT: segue la seguente procedura

1. Trova il nodo contenente la chiave:
   * Si scorrono le chiavi nel nodo per trovare la posizione della chiave da eliminare, o si individua il figlio appropriato per continuare la ricerca se la chiave non è presente nel nodo attuale. Se si raggiunge un nodo foglia e non si trova la chiave, questa allora non è presente nel BT.
2. Caso 1: la chiave si trova in un nodo foglia
   * Si elimina la chiave direttamente dal nodo foglia.
   * Se, dopo la rimozione, il nodo foglia ha almeno $(t-1)$ chiavi (dove $t$ è il grado minimo del BT), la struttura dell'albero è ancora valida.
   * Se il nodo foglia ha meno di $(t-1)$ chiavi, è necessario eseguire delle operazioni di ribilanciamento.
3. Caso 2: la chiave si trova in un nodo interno
   (a) Il nodo figlio sinistro della chiave ha almeno $t$ chiavi:
      * Si trova il predecessore della chiave (la chiave massima nel sottoalbero sinistro).
      * Si sostituisce la chiave da eliminare con il suo predecessore.
      * Si esegui ricorsivamente la cancellazione del predecessore nel sottoalbero sinistro.
   (b) Il nodo figlio destro della chiave ha almeno $t$ chiavi:
      * Si trova il successore della chiave (la chiave minima nel sottoalbero destro).
      * Si sostituisce la chiave da eliminare con il suo successore.
      * Si esegue ricorsivamente la cancellazione del successore nel sottoalbero destro.
   (c) Entrambi i figli della chiave hanno $(t-1)$ chiavi:
      * Si unisce il nodo sinistro, la chiave da eliminare, e il nodo destro in un unico nodo. Questo nuovo nodo avrà $(2t-1)$ chiavi.
      * Si esegue ricorsivamente la cancellazione della chiave nel nuovo nodo unito.
4. Caso 3: la chiave non è nel nodo attuale si deve scendere in un nodo figlio
   * Prima di scendere, è necessario assicurarsi che il figlio abbia almeno $t$ chiavi.
   * Se il figlio ha solo $(t-1)$ chiavi, si eseguono operazioni di ribilanciamento. In particolare si potrebbe attuare la strategia del prestito da un fratello: se infatto il fratello sinistro o destro del figlio ha almeno $t$ chiavi, si sposta una chiave dal nodo padre al nodo figlio e una chiave dal fratello al nodo padre. Oppure la strategia di merge con un fratello, che prevede, se i fratelli hanno $(t-1)$, di unire il figlio con uno di questi e spostare una chiave dal nodo padre al nuovo nodo unito.
   * Si scende nel nodo figlio e si continua la ricerca/cancellazione.

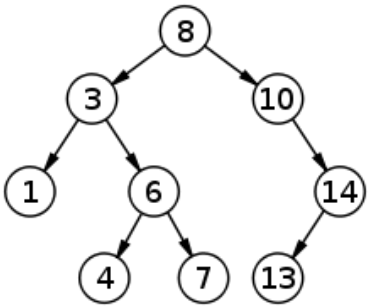
Le operazioni descritte sopra sono implementate in maniera tale da rispettare le proprietà di queste due strutture dati. Nel dettaglio le proprietà di un BST (esempio in figura 1a) possono essere riassunte con le seguenti affermazioni:

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 $x$.

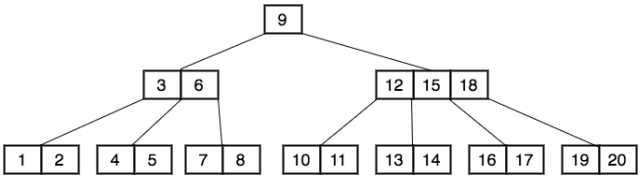
3. Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due BST.

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

1. Ordine dell'albero (o grado minimo):
   - Ogni nodo dell'albero contiene un numero di chiavi che varia tra $(t-1)$ e $(2t-1)$, dove $t$ è il grado minimo dell'albero.
   - Ogni nodo (eccetto la radice) deve avere almeno $(t-1)$ chiavi e al massimo $(2t-1)$ chiavi.
2. Numero di figli per nodo:
   - Ogni nodo interno può avere tra $t$ e $2t$ figli.
   - Se un nodo ha $k$ chiavi, esso ha $(k+1)$ figli, mantenendo coerenza tra il numero di chiavi e di figli.
3. Nodi foglia uniformi:
   - Tutti i nodi foglia si trovano alla stessa profondità, garantendo che il B-albero sia bilanciato.
4. Nodo Radice:
   - La radice può avere meno di $(t-1)$ chiavi solo se è l'unico nodo dell'albero; altrimenti deve avere almeno $(t-1)$ chiavi.
5. Ordinamento delle chiavi:
   - Le chiavi all'interno di ogni nodo sono ordinate in modo crescente.
6. Chiavi non duplicate:
   - Ogni chiave è unica e identifica una posizione o un valore specifico.
7. Altezza dell'Albero:
   - L'altezza massima di un B-albero con $n$ chiavi e ordine $t$ è $O(\log_t(n))$.




(a) Albero binario di ricerca




(b) B albero

Figura 1: Tipologie di alberi

## 2.3 Assunti ed ipotesi

In relazione al caso medio e al caso peggiore sopra definiti e in riferimento allo studio della complessità degli algoritmi delle tre operazioni fondamentali di un albero riportati sul libro di testo, viene stesa la seguente tabella.

| Operazione | Struttura | Caso Medio | Caso Peggiore |
|---|---|---|---|
| **Inserimento** | BST | **Nodi letti:** $O(\log n)$<br>**Nodi scritti:** $O(1)$<br>**Complessità:** $O(\log n)$ | **Nodi letti:** $\Theta(n)$<br>**Nodi scritti:** $O(1)$<br>**Complessità:** $\Theta(n)$ |
| | BT | **Nodi letti:** $O(\log_t n)$<br>**Nodi scritti:** $O(\log n)$<br>**Complessità:** $O(\log_t n)$ | **Nodi letti:** $O(\log_t n)$<br>**Nodi scritti:** $O(\log n)$<br>**Complessità:** $O(\log_t n)$ |
| **Ricerca** | BST | **Nodi letti:** $O(\log n)$<br>**Nodi scritti:** $O(0)$<br>**Complessità:** $O(\log n)$ | **Nodi letti:** $\Theta(n)$<br>**Nodi scritti:** $O(0)$<br>**Complessità:** $\Theta(n)$ |
| | BT | **Nodi letti:** $O(\log_t n)$<br>**Nodi scritti:** $O(0)$<br>**Complessità:** $O(\log_t n)$ | **Nodi letti:** $O(\log_t n)$<br>**Nodi scritti:** $O(0)$<br>**Complessità:** $O(\log_t n)$ |
| **Cancellazione** | BST | **Nodi letti:** $O(\log n)$<br>**Nodi scritti:** $O(\log n)$<br>**Complessità:** $O(\log n)$ | **Nodi letti:** $\Theta(n)$<br>**Nodi scritti:** $O(1)$<br>**Complessità:** $\Theta(n)$ |
| | BT | **Nodi letti:** $O(\log_t n)$<br>**Nodi scritti:** $O(\log_t n)$<br>**Complessità:** $O(\log_t n)$ | **Nodi letti:** $O(\log_t n)$<br>**Nodi scritti:** $O(\log_t n)$<br>**Complessità:** $O(\log_t n)$ |

Tabella 1: Confronto tra la complessità di BST e BT

# 3 Documentazione del codice

## 3.1 Schema del contenuto e interazione tra i moduli

Di seguito è riportato il diagramma UML che integra le relazioni tra i diversi moduli del programma. I nomi delle classi sono autoidentificativi. Particolare precisazione meritano invece le relazioni tra queste. La classe $Node$ rappresenta un nodo di un BST, dunque la relazione ricorsiva con cardinalità 0...2 definisce il concetto che in un un albero binario, ogni nodo può avere al massimo due figli: uno a sinistra e uno a destra. Tuttavia, un nodo potrebbe anche non avere alcun figlio (se è una foglia) o avere solo uno dei due figli (sinistro o destro). Mentra la relazione di composizione tra le classi $BinarySearchTree$ e $Node$ con cardinalità 1...* sposta l'attenzione sul ciclo di vita degli oggetti: i nodi $Node$ sono parti fondamentali della struttura di un albero binario. Se l'istanza dell'albero viene distrutta, anche tutti i suoi nodi associati vengono distrutti automaticamente. Quindi i nodi non hanno una vita indipendente dall'albero (e il medesimo ragionamento può essere applicato alle classi $BTree$ e $NodeBT$). La classe $PlotGenerator$ invece definisce i metodi necessari per la creazione di un infografica necessaria al risultato degli esperimenti. Ha dunque bisogno di utilizzare istanze delle classi $BTree$ e $BinarySearchTree$, da cui le relazioni di dipendenza nel diagramma UML.

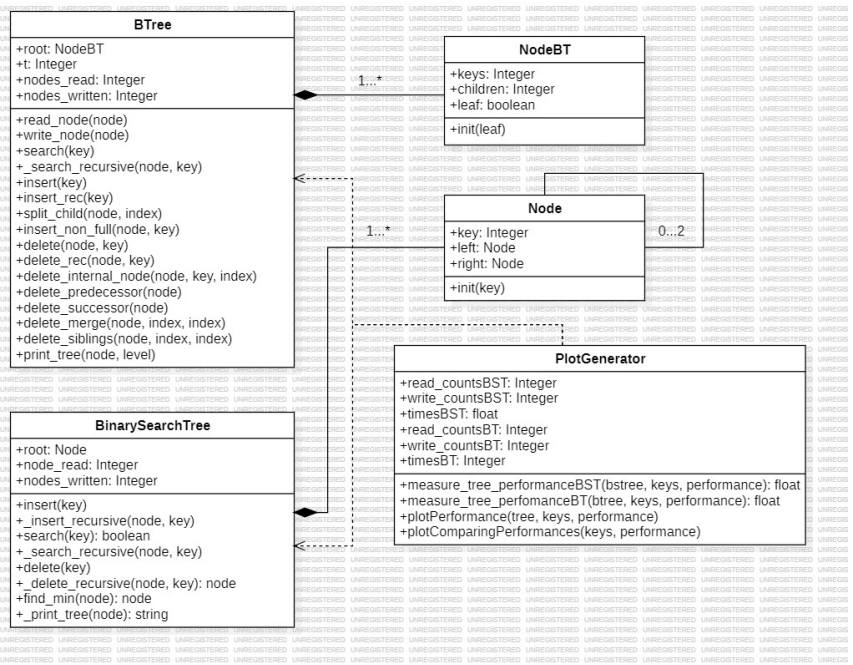



Figura 2: Diagramma delle classi

## 3.2 Descrizione dei metodi implementati

Viene di seguito riportato una breve descrizione letterale che mette luce sulle funzionalità offerte dai vari metodi delle classi, le quali signatures sono presenti nel diagramma UML della precedente sezione.

- **Node**
  - **init(key)**: imposta il valore della chiave del nodo corrente con il valore $key$, dunque inizializza a $None$ i puntatori ai nodi figli destro e sinistro. Quindi gestisce il nodo come se fosse una foglia.
- **BinarySearchTree**
  - **insert(key)**: riceve in ingresso una chiave $key$ da inserire, inizializza a zero i contatori dei nodi scritti e letti prima di ogni operazione di reale inserimento, ottenuta attraverso la chiamata del metodo ricorsivo seguente.
  - **_insert_recursive(node, key)**: metodo ricorsivo che implementa la logica dell'inserimento dell'attributo $key$ per un BST, a partire dal nodo $node$.
  - **search(key)**: riceve in ingresso una chiave $key$ da cercare nell'albero, quindi azzera prima di ogni ricerca il contatore dei nodi letti. Chiama il metodo ricorsivo seguente, restituendo un valore booleano che indica l'esito finale della ricerca.
  - **_search_recursive(node, key)**: implementa la logica di ricerca dell'attributo $key$ per un BST a partire dal nodo $node$.
  - **delete(key)**: riceve in ingresso come parametro la chiave $key$ da rimuovere, azzerea i contatori dei nodi letti e scritti, dunque effettua una chiamata al metodo ricorsivo che segue.
  - **_delete_recursive(node, key)**: implementa la logica della cancellazione per un BST a partire dal nodo $node$. Il nodo che viene restituito rappresenta il nodo corrente dell'albero aggiornato, dopo aver applicato la logica di cancellazione del nodo contenente la chiave $key$. Questo metodo viene chiamato ricorsivamente per risalire verso la radice e aggiornare i collegamenti tra i nodi, mantenendo intatta la struttura dell'albero binario di ricerca.
  - **find_min(node)**: cerca il nodo con il valore minimo in un albero binario di ricerca partendo dal nodo specificato come input $node$. Questo metodo è utilizzato in operazioni di cancellazione: quando si elimina un nodo con due figli, è prassi sostituirne il valore con il valore minimo del sottoalbero destro per mantenere le proprietà dell'albero binario di ricerca.

- **NodeBT**
  - **init(leaf)**: inizializza un vettore vuoto atto a contenere i valori della chiavi per quel nodo, dunque un array vuoto per i puntatori ai figli. Attraverso il valore booleano $leaf$ imposta si indica se il nodo corrente ha le proprietà di un nodo foglia o meno.
- **BTree**
  - **read_node(node)**: incrementa il contatore dei nodi letti, dunque restituisce il nodo la quale lettura è appena avvenuta.
  - **write_node(node)**: incrementa il contatore dei nodi letti.
  - **search(key)**: resetta a zero il contatori dei nodi letti prima di iniziare una nuova ricerca. Effettua dunque una chiamata al metodo ricorsivo seguente al fine di garantire l'effettiva ricerca del parametro $key$.
  - **_search_recursive(node, key)**: implementa la logica di ricerca di un BT della chiave $key$ a partire dal nodo $node$. Gestisce il conteggio dei nodi letti per ogni ricerca tenendo escluso da questo il nodo radice, in quanto si presupponde che si trovi sempre già in memoria.
  - **insert(key)**: resetta a zero i contatori dei nodi letti e scritti prima di ogni nuova operazione di inserimento. Chiama dunque il metodo ricorsivo che segue per inserire effettivamente la chiave $key$ nel BT.
  - **insert_rec(key)**: implementa la logica di inserimento del valore della chiave $key$ nel BT a partire sempre dal nodo radice dell'albero.
  - **split_child(node, index)**: effettua lo "splitting" di un nodo full, cioè un nodo che contiene il numero massimo di chiavi possibili. Il nodo su cui effettuare questa operazione è quel nodo che corrisponde all'$index$-esimo figlio di $node$.
  - **insert_non_full(node, key)**: questo metodo è progettato per inserire una chiave $key$ in un nodo $node$ di un BT che non è pieno (cioè, contiene meno del numero massimo di chiavi per ogni nodo). Distingue i casi in cui $node$ è un nodo foglia o un nodo interno.
  - **delete(node, key)**: resetta a zero i contatori dei nodi scritti e letti prima di ogni cancellazione nell'albero. Effettua poi la chiamata al metodo ricorsivo seguente.
  - **delete_rec(node, key)**: effettua la cancellazione della chiave $key$ a partire dal nodo $node$. Gestisce tutti i possibili casi che si possono verificare in un BT.
  - **delete_internal_node(node, key, index)**: gestisce la cancellazione della chiave $key$ che si trova alla posizione $index$ nel nodo interno del BT $node$.
  - **delete_predecessor(node)**: rimuove e ritorna il predecessore del nodo $node$, cioè la chiave più grande del sottoalbero sinistro con radice $node$.
  - **delete_successor(node)**: rimuove e ritorna il successore del nodo $node$, cioè la chiave minore del sottoalbero destro con radice $node$.
  - **delete_merge(node, index, index)**: questo metodo viene invocato quando almeno uno dei figli index-esimi del nodo $node$ contengono $(t-1)$ chiavi. Il suo scopo è fondere tali figli col fine di preservare le proprietà strutturali del BT.
  - **delete_siblings(node, index, index)**: viene invocato quando un nodo ha meno di $(t-1)$ chiavi con l'obbiettivo di ribilanciarlo dopo una cancellazione di una chiave del nodo stesso. In particolare questo metodo implementa la logica di prestare una chiave al nodo figlio $index$-esimo (secondo parametro $index$ della signature del metodo) presa in prestito dal nodo figlio $index$-esimo (terzo parametro della signature del metodo).

La classe $PlotGenerator$ implementa i metodi necessari alla misurazione delle operazioni sulla struttura dati, quindi genera un'infografica utile per le considerazioni effettuate nella **??**. L'infografica è definita attraverso un plot in cui si osservano tre grafici cartesiani dove sull'asse delle ascisse vengono posti i valori di ogni chiave, e sull'asse delle ordinate rispettivamente i tempi, i nodi letti e i nodi scritti.

# 4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali

## 4.1 Dati utilizzati

Gli esperimenti condotti discostano per il numero di nodi considerati della struttura dati, dunque per l'ordinamento del set di chiavi oggetto delle diverse operazioni e per il range in cui tali chiavi vengono considerate. In particolare verrà posta l'attenzione sulle seguenti casistiche:

- alberi con **30** elementi
  - Inserimento con set di chiavi lineare;
  - Inserimento con set di chiavi randomico;
- alberi con **100** elementi
  - Inserimento con set di chiavi lineare;
  - Inserimento/Cancellazione con set di chiavi randomico;
- alberi con **300** elementi
  - Inserimento con set di chiavi lineare;
  - Inserimento/Cancellazione con set di chiavi randomico;
- alberi con **900** elementi
  - Inserimento con set di chiavi lineare;
  - Inserimento/Cancellazione con set di chiavi randomico;

Gli obbiettivi sono osservare come si comportano le strutture in relazione al numero di nodi scritti/letti quando aumenta in progressione la loro cardinalità, Si ponga anche l'attenzione sulle tempistiche delle operazioni quando varia il range di randomicità nel caso di alberi con 300 elementi. Per i BT si osservi anche come queste caratteristiche evolvono quando si considerano alberi con gradi minimi diversi.

NOTA:

- durante gli esperimenti condotti si è cercato di non superare la soglia dei 1000 elementi. La ragione ultima risiede nel fatto che il compilatore dell'IDE riscontra l'errore '`MAXIMUN CALL RECURSION ACHIEVED`', se si usa un approccio ricorsivo.
- le chiavi considerate sono appartenenti a set interi di valori. Il range di randomicità di default utilizzato è [0-1000].
- le operazioni di cancellazione vengono effettuate su strutture dati il cui inserimento è stato effettuato con set di chiavi randomici.
- le operazioni di ricerca vengono effettuate su strutture dati il cui inserimento è stato effettuato con set di chiavi randomici. Risulterebbe infatti avere poco senso procedere con ricerche di set lineari in alberi riempiti con lo stesso criterio. **Non viene dunque posta l'attenzione sulla conduzione degli esperimenti nel caso dell'operazione di ricerca.**

## 4.2 Misurazioni temporali

La classe $PlotGenerator$ implementa il metodo $measure\_tree\_performanceBST()$ (e la sua duale $measure\_tree\_performanceBT()$) che valuta, per ogni chiave del set, il numero dei nodi scritti/letti e il tempo necessario al conseguimento dell'operazione in questione (inserimento, ricerca, cancellazione). La misurazione temporale viene eseguita attraverso una sottrazione di uno `end_time` da uno `start_time`. Sommando i risultati ottenuti per ogni chiave nel set considerato, si giunge al tempo totale necessario per il conseguitmento dell'operazione dell'intero set. Di particolare interesse dunque sarà la valutazione di tali misurazioni per alberi con lo stesso numero di nodi ma set di chiavi differenti.

```
        def measure_tree_performanceBST(self, bstree, keys,
        ↪  performance):
        self.read_countsBST = []
        self.write_countsBST = []
        self.timesBST = []

        for key in keys:
            start_time = time.perf_counter()  # time in sec

            if performance == "Insert":
                bstree.insert(key)
            elif performance == "Search":
                bstree.search(key)
            elif performance == "Delete":
                bstree.delete(key)

            end_time = time.perf_counter()
            self.timesBST.append(end_time - start_time)
            self.read_countsBST.append(bstree.nodes_read)
            self.write_countsBST.append(bstree.nodes_written)

        return sum(self.timesBST)
```

## 4.3 Risultati sperimentali

In questa sezione si presentano i risultati dei test effettuati sul programma realizzato. I risultati sono riportati attraverso l'infografica già precedentemente descritta, per ogni operazione considerata. In aggiunta viene anche stesa una tabella che dettaglia le misurazioni temporali associate

### 4.3.1 Alberi con 30 elementi

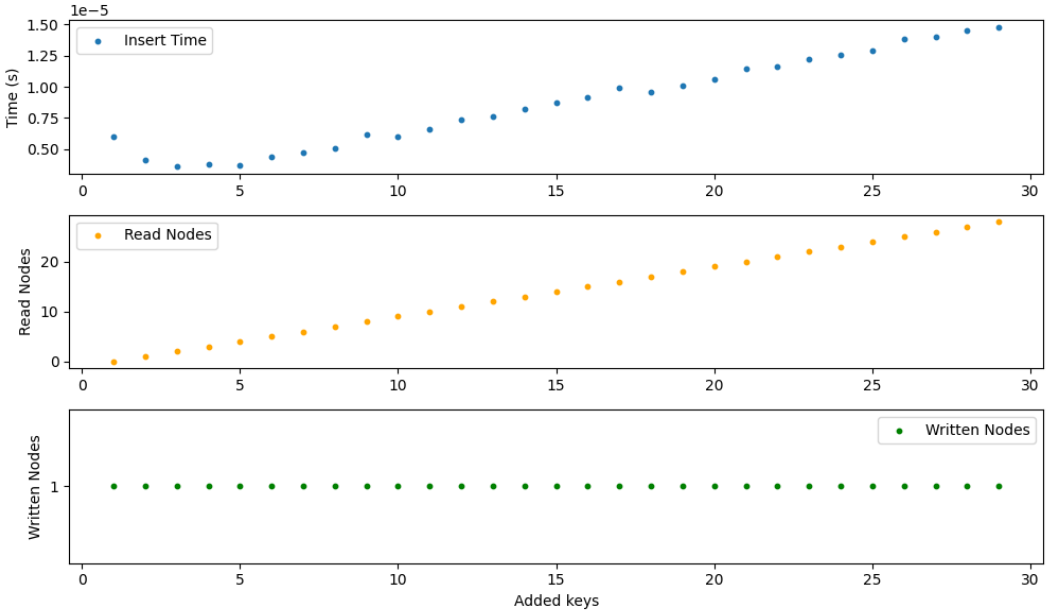



Figura 3: Inserimento BST - Set lineare con 30 chiavi

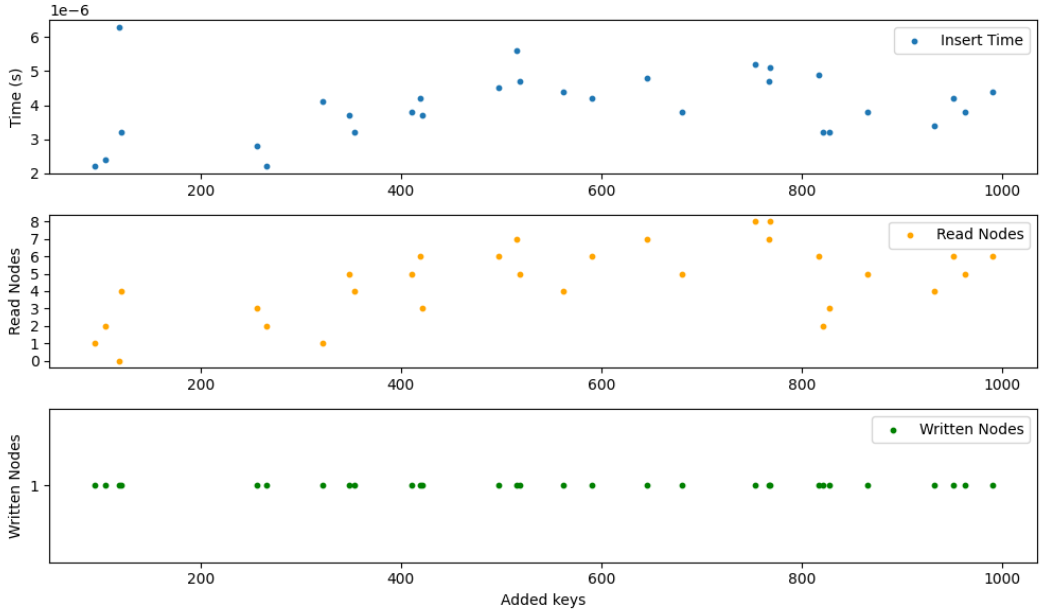



Figura 4: Inserimento BST - Set randomico con 30 chiavi

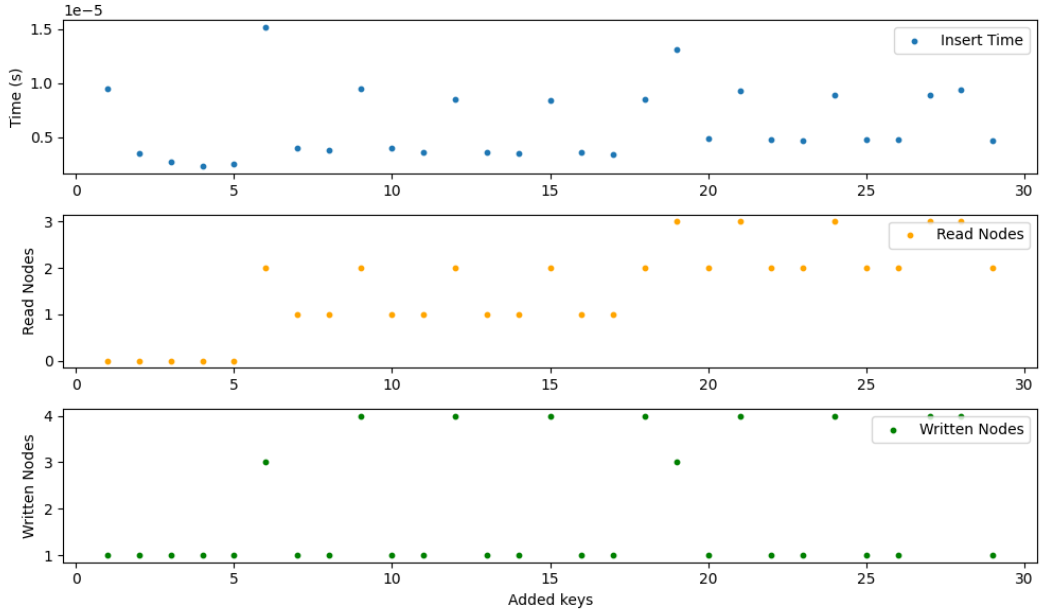



Figura 5: Inserimento BT - Set lineare con 30 chiavi e t=3

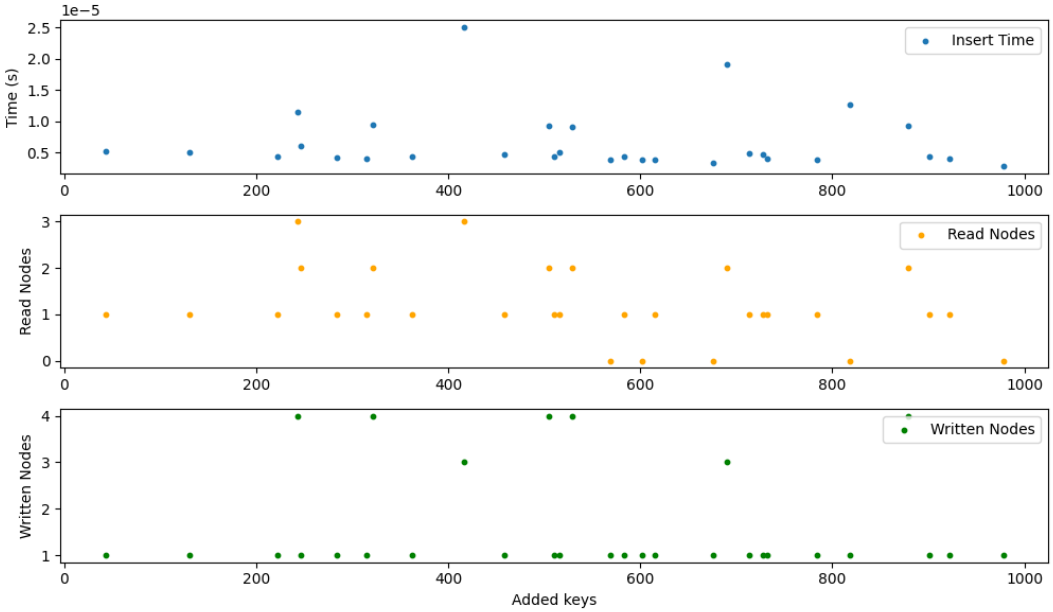



Figura 6: Inserimento BT - Set randomico con 30 chiavi e t=3

| Struttura dati | Inserimento | |
|---|---|---|
| | Lineare | Randomico |
| BST | 0.2511ms | 0.1147ms |
| BT | 0.1706ms | 0.2010ms |

Tabella 2: Tempistiche (in microsecondi) alberi con 30 elementi

### 4.3.2 Alberi con 100 elementi

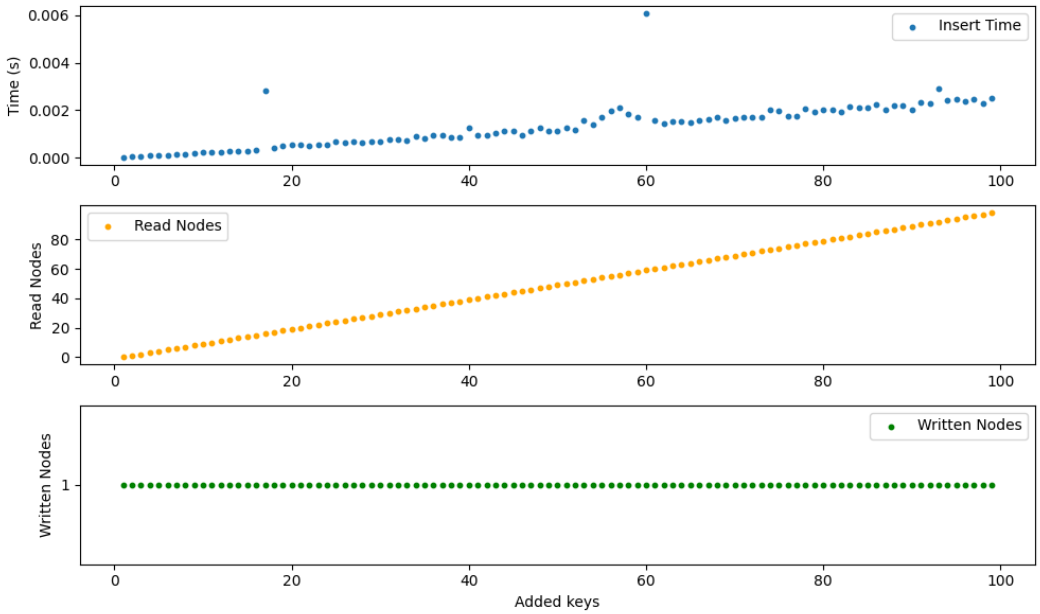



Figura 7: Inserimento BST - Set lineare con 100 chiavi

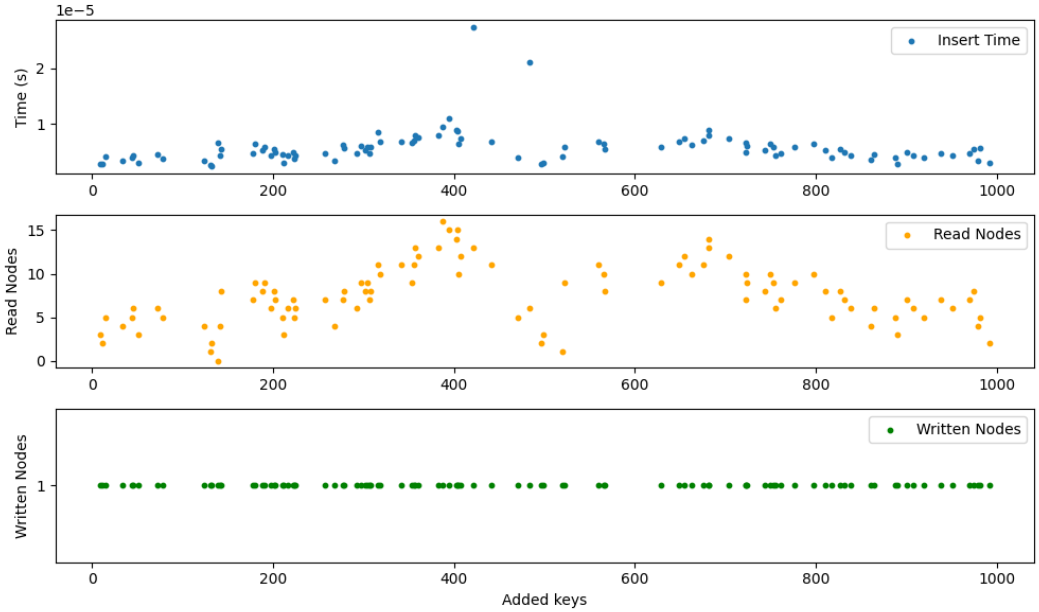



Figura 8: Inserimento BST - Set randomico con 100 chiavi

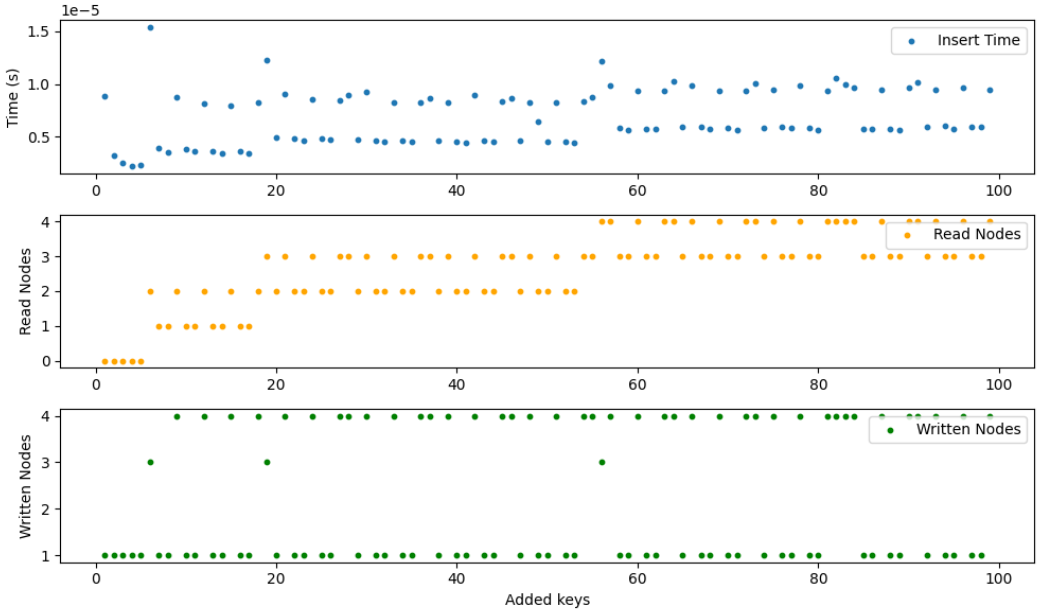



Figura 9: Inserimento BT - Set lineare con 100 chiavi e t=3

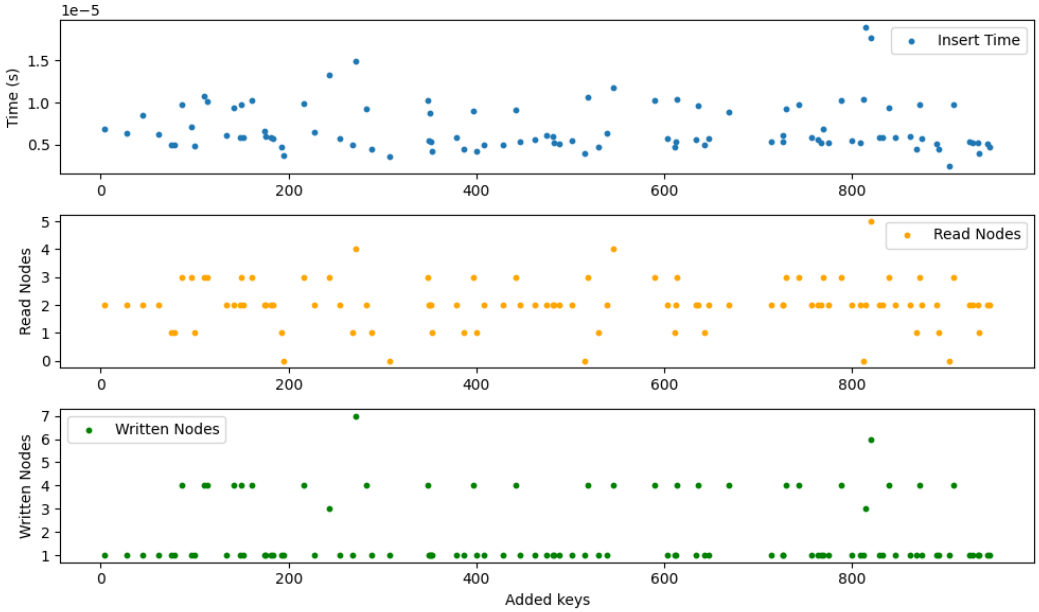



Figura 10: Inserimento BT - Set randomico con 100 chiavi e t=3

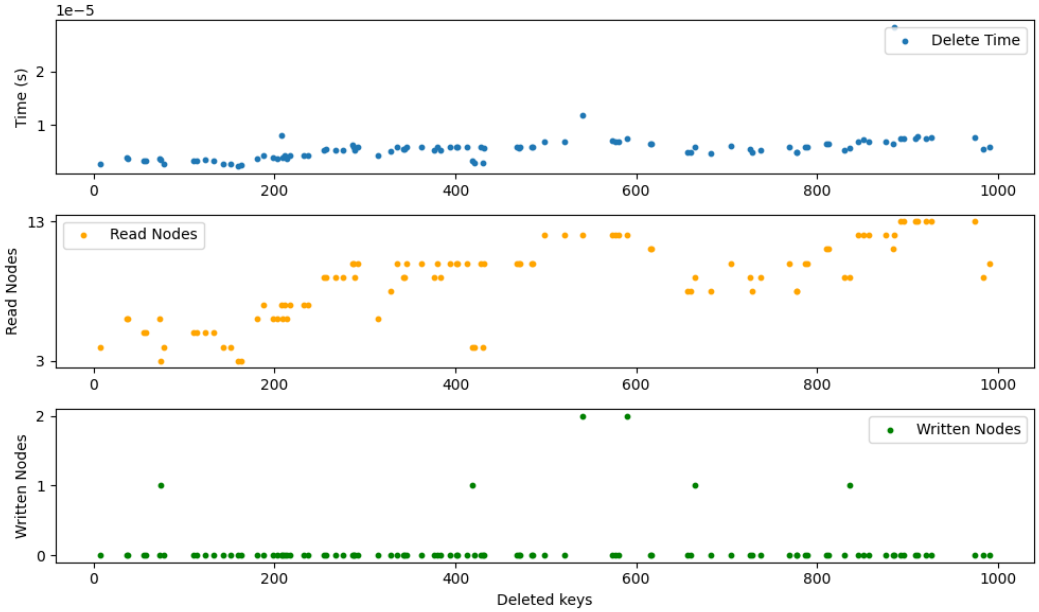



Figura 11: Cancellazione BST - Set randomico con 100 chiavi e t=3

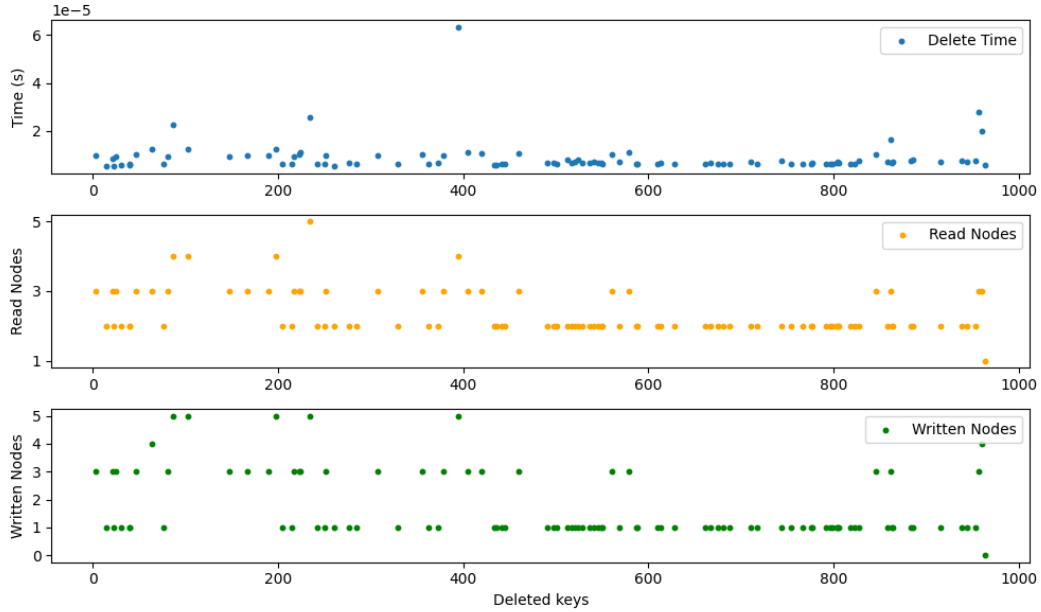



Figura 12: Cancellazione BT - Set randomico con 100 chiavi e t=3

| Struttura dati | Inserimento | | Cancellazione |
|---|---|---|---|
| | Lineare | Randomico | |
| BST | 3.0318ms | 0.5768ms | 0.5639ms |
| BT | 0.6801ms | 0.6931ms | 0.8693ms |

Tabella 3: Tempistiche (in microsecondi) alberi con 100 elementi

### 4.3.3 Alberi con 300 elementi

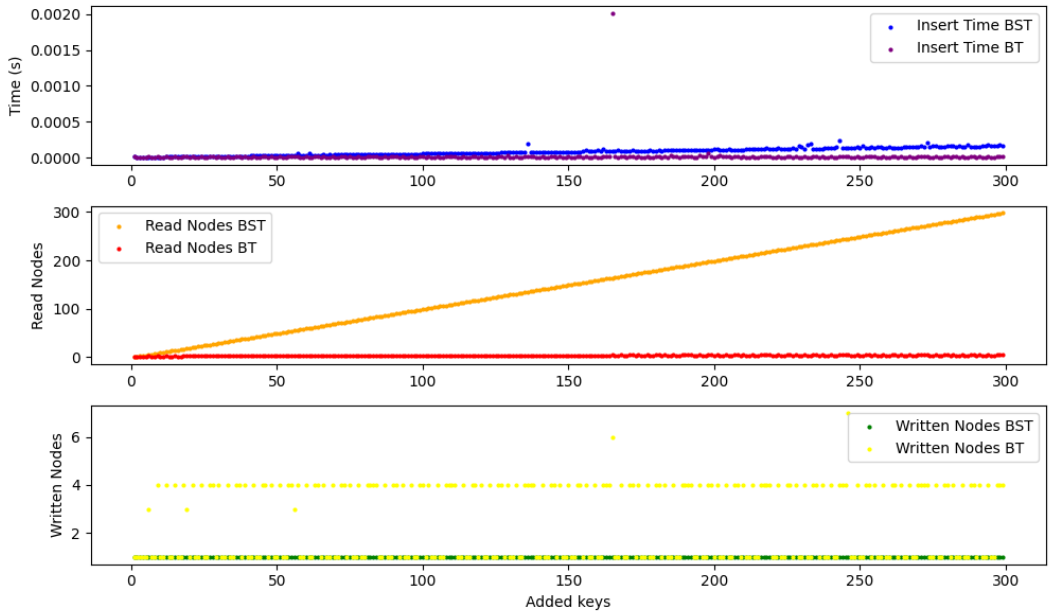



Figura 13: Inserimento BST vs BT - Set lineare con 300 chiavi e t=3


Figura 14: Inserimento BST vs BT - Set randomico con 300 chiavi e t=3

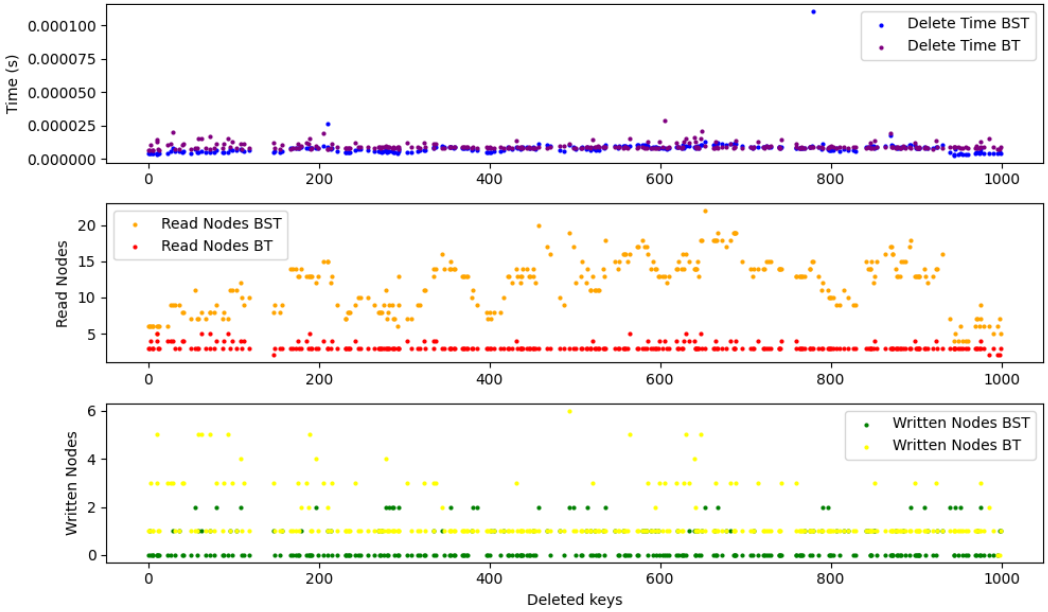



Figura 15: Cancellazione BST vs BT - Set randomico con 300 chiavi e t=3

| **Struttura dati** | **Inserimento** | | **Cancellazione** |
|---|---|---|---|
| | **Lineare** | **Randomico** | |
| BST | 24.5012ms | 2.1048ms | 2.4027ms |
| BT | 4.182ms | 2.5586ms | 2.8427ms |

Tabella 4: Tempistiche (in microsecondi) alberi con 300 elementi

### 4.3.4 Alberi con 900 elementi

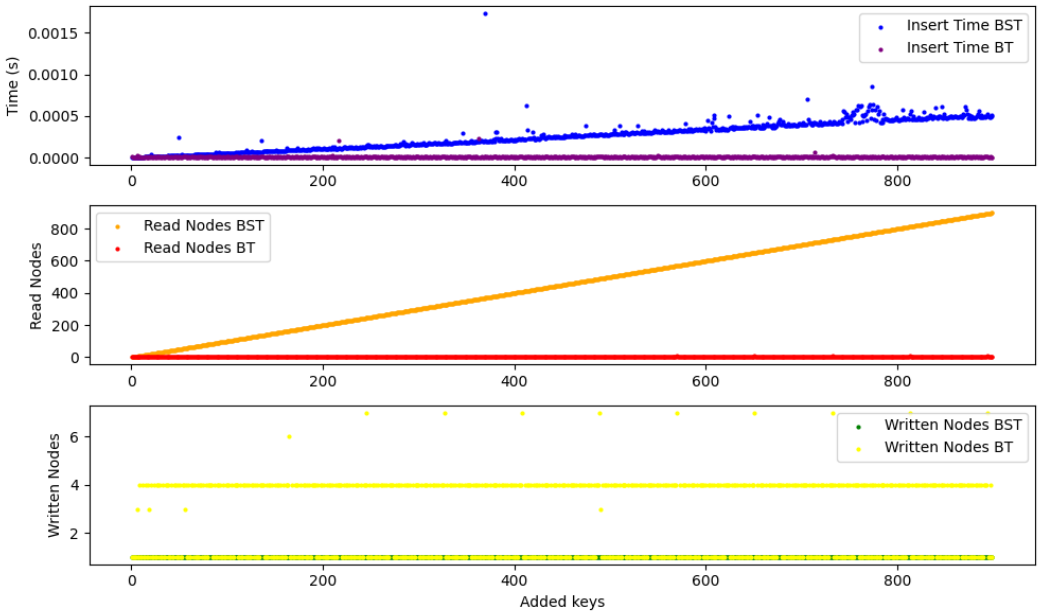



Figura 16: Inserimento BST vs BT - Set lineare con 900 chiavi e t=3


Figura 17: Inserimento BST vs BT - Set randomico con 900 chiavi e t=3

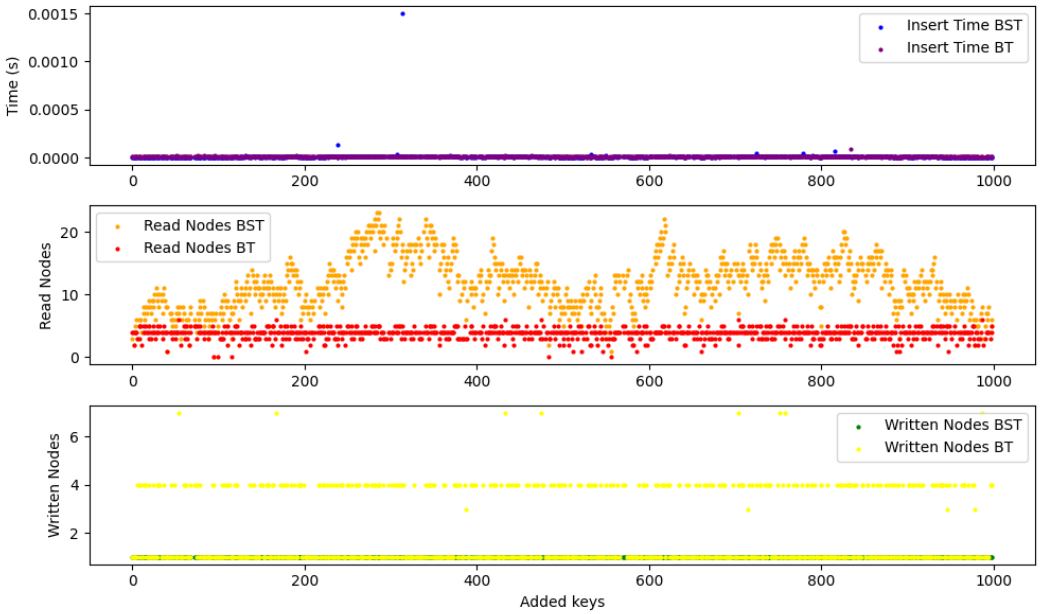



Figura 18: Inserimento BST vs BT - Set randomico con 900 chiavi e t=3

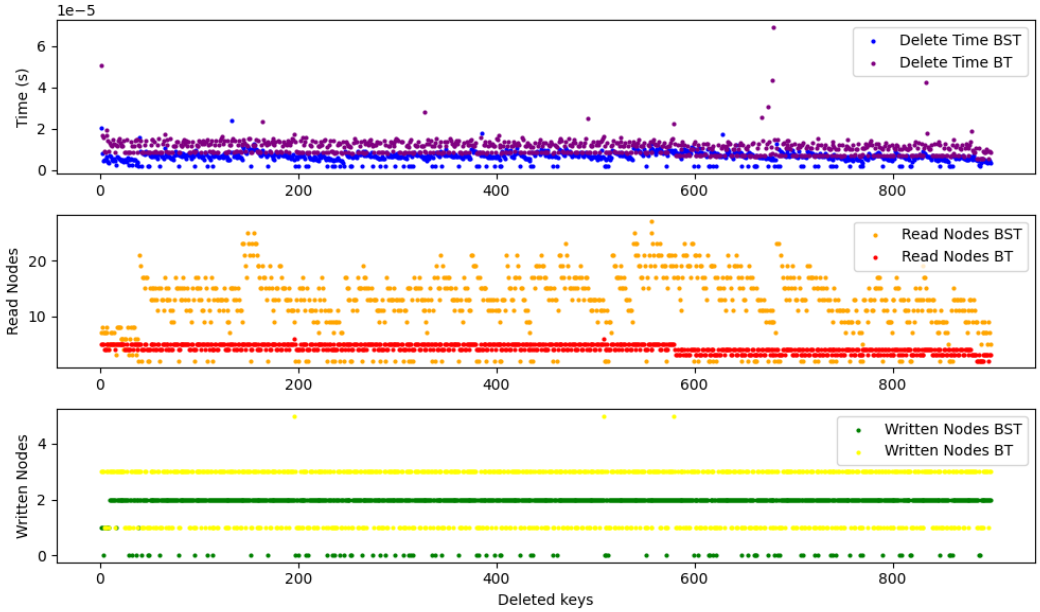



Figura 19: Cancellazione BST vs BT - Set lineare con 900 chiavi e t=3

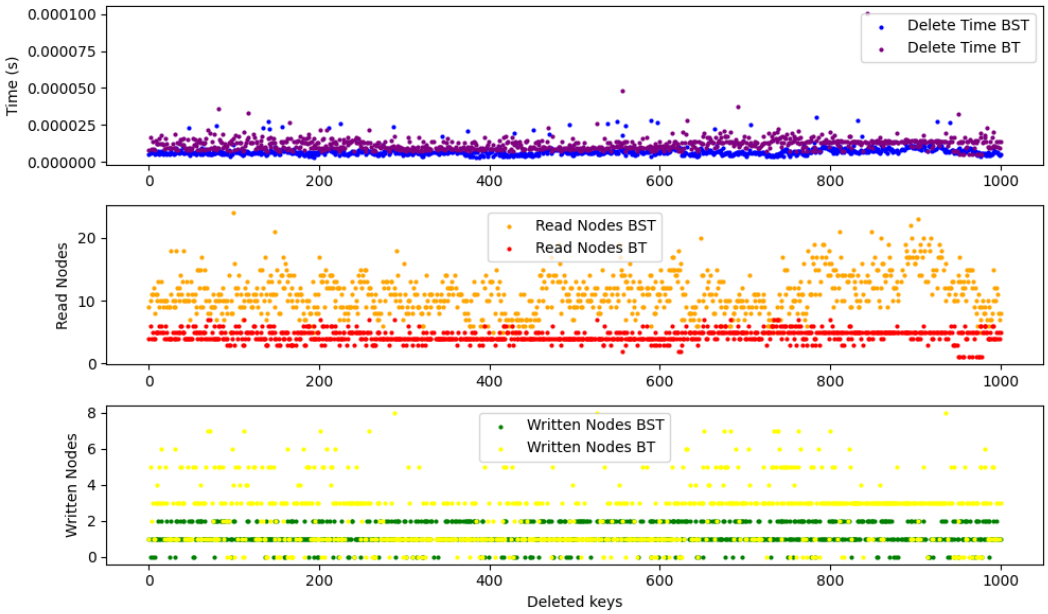



Figura 20: Cancellazione BST vs BT - Set randomico con 900 chiavi e t=3

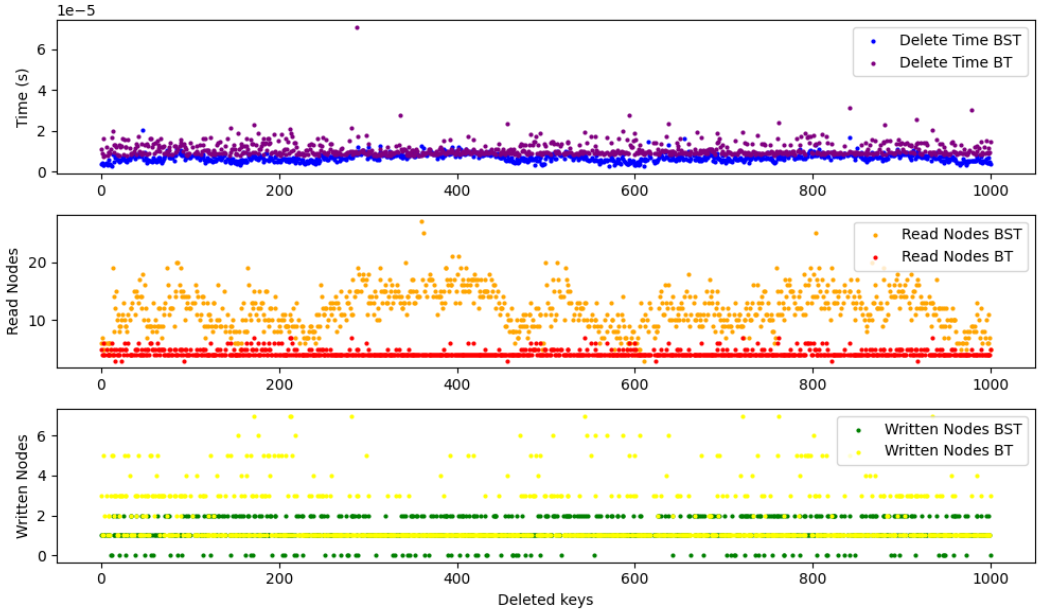



Figura 21: Cancellazione BST vs BT - Set randomico con 900 chiavi e t=3

| **Struttura dati** | **Inserimento** | | **Cancellazione** | |
|---|---|---|---|---|
| | **Lineare** | **Randomico** | **Lineare** | **Randomico** |
| BST | 235.7490ms | 8.3671ms | 5.9401ms | 9.1760ms |
| BT | 8.6561ms | 8.42759ms | 9.9464ms | 10.7027ms |

Tabella 5: Tempistiche (in microsecondi) alberi con 900 elementi

## 4.4 Commenti analitici

L'analisi dei risultati degli esperimenti condotti viene anche estesa alla relazione temporale, oltre che al confronto del numero dei nodi letti/scitti per l'operazione in oggetto.

- **Inserimento**
  - Nodi letti: se i casi lineari non necessitano di particolare attenzione, in quanto si possono estrarre le informazioni sufficienti dalle infografiche per confermare gli assunti, ci concentreremo più energicamente sulle casistiche randomiche. Si osservi che il numero dei nodi letti per un BST ricade in un intervallo che riflette l'altezza variabile dell'albero stesso. Questa infatti può divenire alta se la struttura non è bilanciata, con una complessità $O(n)$. In genere con set di chiavi randomici tende ad essere più vicina a $O(\log n)$, ma possono comunque verificarsi cammini più lunghi, ad esempio si confrontino le figure 17 e 18. Con un BT invece questo intervallo si riduce bruscamente in quanto, dividendo lo spazio delle chiavi tra i nodi interni, l'accesso per trovare quello corretto avviene in una profondità molto ridotta (appunto al massimo $O(\log_t n)$.
  - Nodi scritti: per il BST questo valore è sempre 1, in quanto l'inserimento non comporta la modifica di nodi già esistenti ma aggiunge semplicemente un nuovo nodo come foglia senza alterare la struttura. Per il BT invece l'inserimento richiede talvolta la divisione dei nodi quando questi eccedono il grado massimo (split). Questo spiega il numero di nodi scritti, che può aumentare in base alla necessità di aggiornare i nodi genitori e propagare la divisione verso l'alto.
- **Cancellazione**
  - Nodi letti: prendendo in esame le figure 20 e 21 il range di intervallo è circa [5-30] per un BST. Questo ampio intervallo è coerente con il comportamento dei BST non bilanciati, dove il numero di nodi letti dipende dall'altezza dell'albero. Un albero non bilanciato può degradare in altezza fino a $n$ (il numero dei nodi), aumentando significativamente il costo in termini di letture, soprattutto per chiavi situate verso la foglia più profonda. Invece per i BT il range si riduce a circa [1-6]. Questo comportamento riflette la struttura bilanciata dei BT. Il numero massimo di nodi letti è proporzionale alla profondità dell'albero $O(\log_t n)$ (t = 3 in queste infografiche). Si noti quindi che con un set di 900 chiavi e grado 3, la profondità è bassa, e questo quindi mantiene il numero di nodi letti entro un range ristretto.
  - Nodi scritti: dalle figure 20 e 21 si apprende che il range in cui questo contatore varia è [0-2] per i BST. Si tratta di un range atteso poiché la cancellazione comporta al massimo una riorganizzazione locale (ad esempio, la sostituzione di un nodo con il suo successore o predecessore in caso di nodi con figli). Non vi sono quindicosti di bilanciamento o modifiche strutturali globali. Nei BT invece, la cancellazione è più complessa rispetto ai BST. Potrebbe richiedere la fusione di nodi (merge) o la redistribuzione di chiavi tra nodi fratelli per mantenere le proprietà dell'albero. Questi aggiustamenti si propagano verso l'alto, influendo sul numero di nodi scritti. Questo spiega il range [0-8] dei nodi scritti per il BT.

L'analisi delle misurazioni temporali che segue si riferisce all'andamento dei valori che si osservano in tabella 5.

- **Inserimento (lineare e randomico)**: per il BST il tempo per un inserimento lineare è significativamente più alto (235.7490ms) rispetto a quello randomico (8.3671ms). Questo comportamento è coerente con le complessità teoriche in quanto un inserimento lineare in un BST non bilanciato degrada a $O(n)$, mentre un inserimento randomico può rimanere $O(\log n)$ in media, grazie a una struttura più bilanciata. Per il BT i tempi per gli inserimenti lineari e randomici sono molto simili (8.6561ms e 8.42759ms). Questo andamento è atteso, poiché un albero bilanciato mantiene sempre una complessità $O(\log n)$ (in particolare $O(\log_t n)$ se è un BT) per l'inserimento, indipendentemente dall'ordine dei dati.
- **Cancellazione (lineare e randomica)**: nel caso del BST, i tempi di cancellazione sono vicini ma leggermente migliori per la casistica lineare (2.4027ms contro 3.1452ms). Questo può essere dovuto al fatto che i nodi vicini nel caso lineare si trovano frequentemente in posizioni prevedibili, riducendo il tempo di navigazione. Tuttavia, la cancellazione nel caso randomico rispecchia il comportamento tipico $O(h)$. Per il BT, i tempi per la cancellazione lineare e randomica sono molto simili (2.8427ms e 3.0124ms), riflettendo una gestione uniforme grazie alla garanzia di bilanciamento dell'albero. Anche qui, la complessità $O(\log_t n)$ è rispettata.

# 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.