# BST vs AVL vs RBT

Matteo Orlandi

February 22, 2026

# 1 Introduzione

## 1.1 Caso di studio

In questa relazione verranno analizzate e confrontate le prestazioni di tre diverse strutture dati utilizzate per la gestione di insiemi dinamici. Le strutture che verranno analizzate sono: gli **Alberi Binari di Ricerca (BST)**, gli alberi **AVL** e gli **Alberi Rosso-Neri (RBT)**.

Un obiettivo fondamentale di ogni struttura dati è la gestione efficiente dei dati che ospiterà. Sebbene i BST offrano una struttura semplice e intuitiva per l'inserimento e la ricerca, le prestazioni ottenute dipendono fortemente dall'ordine dei dati in ingresso.

L'esperimento mira a verificare sperimentalmente le complessità temporali delle operazioni di **inserimento** e di **ricerca**, mettendo a confronto le tre strutture dati negli scenari del caso peggiore e del caso medio.

## 1.2 Specifiche hardware

Per svolgere i test è stato utilizzato come ambiente di sviluppo **PyCharm 2025.3.2.1**, per realizzare la relazione è stato utilizzato l'editor online di **Overleaf**. Le caratteristiche hardware della macchina utilizzata per svolgere i test sono:

- CPU: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx (2.30 GHz)
- RAM: 8GB DDR4 2400 MHz
- SSD: SSD NVMe 512GB
- SO: Windows 11 Home

# 2 Teoria

Un albero è una struttura dati composta da un insieme di **nodi** e **archi**. Ogni nodo rappresenta un elemento dell'albero ed è composto da una chiave che lo identifica univocamente e da più puntatori, detti archi, che offrono un collegamento verso altri nodi, detti **figli**.

Il nodo principale, da cui ha origine l'intera struttura, prende il nome di **radice**. Ogni nodo che non ha figli viene detto **foglia**.

Si dice *cammino* un insieme di nodi e archi che devono essere attraversati per collegare un nodo a un altro. La lunghezza del cammino più lungo rappresenta l'**altezza** dell'albero e viene misurata contando il numero di nodi che si incontrano lungo quel cammino (parte dalla radice e giunge a una foglia)

## 2.1 Binary Search Tree (BST)

Un Albero Binario di Ricerca è un tipo particolare di albero specializzato per la ricerca di valori.

Ogni nodo avrà un puntatore sinistro che permette il collegamento a un sottoalbero contenente solo valori minori e uno destro che permette il collegamento a un sottoalbero contenente solo valori maggiori.

Le operazioni di base richiedono un tempo proporzionale all'altezza $h$ dell'albero, ovvero $O(h)$.

- **Caso Medio:** Se le chiavi vengono inserite in ordine casuale, l'altezza attesa dell'albero è $O(\log n)$.

- **Caso Peggiore:** Se le chiavi vengono inserite in ordine crescente o decrescente, l'albero degenera in una lista concatenata di altezza $h = n - 1$. In questo scenario, le operazioni raggiungono una complessità $O(n)$, rendendo la struttura inefficiente.

## 2.2 Alberi AVL

Gli alberi AVL sono alberi binari di ricerca **strettamente bilanciati in altezza**. Un albero AVL è bilanciato se, per ogni nodo $x$, le altezze dei sottoalberi sinistro e destro differiscono al massimo di 1. Per fare ciò, viene introdotto il *Fattore di Bilanciamento* $BF(x)$, definito come:

$$BF(x) = h(x.left) - h(x.right) \tag{1}$$

Perché un albero sia AVL, deve valere che $BF(x) \in \{-1, 0, 1\}$ per ogni nodo. A causa di questo vincolo molto stringente, gli alberi AVL sfruttano al massimo l'ampiezza, ottenendo un'altezza massima limitata a circa $1.44 \lg n$. Nonostante abbiano un costo di inserimento non molto ottimizzato a causa dei frequenti controlli e delle rotazioni, il costo per la ricerca è ottimale.

## 2.3 Red-Black Tree (RBT)

Gli alberi Rosso-Neri hanno un funzionamento simile a quello degli AVL, ma il bilanciamento viene effettuato secondo limiti meno restrittivi. Ogni nodo contiene un bit supplementare di informazione che indica il colore (rosso o nero). Un albero Rosso-Nero, oltre a soddisfare le caratteristiche di un BST, deve soddisfare le seguenti proprietà:

1. La radice è nera.
2. Ogni nodo è rosso o nero.
3. Ogni foglia (NIL) è nera.
4. Se un nodo è rosso, entrambi i suoi figli sono neri (non possono esserci due nodi rossi consecutivi).
5. Per ogni nodo, tutti i cammini semplici che scendono verso le foglie discendenti contengono lo stesso numero di nodi neri.

In questo modo abbiamo la garanzia che il cammino più lungo non sia mai maggiore del doppio del cammino più breve. L'altezza dell'albero è perciò limitata a $2 \lg(n + 1)$, assicurando che il limite asintotico delle operazioni di inserimento e ricerca sia $O(\log n)$ anche nel caso peggiore.

# 3 Documentazione del codice

## 3.1 Schema delle classi

Le uniche classi create sono per l'implementazione delle strutture dati. I test e la generazione dei grafici sono stati effettuati tramite metodi *standalone* al fine di semplificare il progetto.

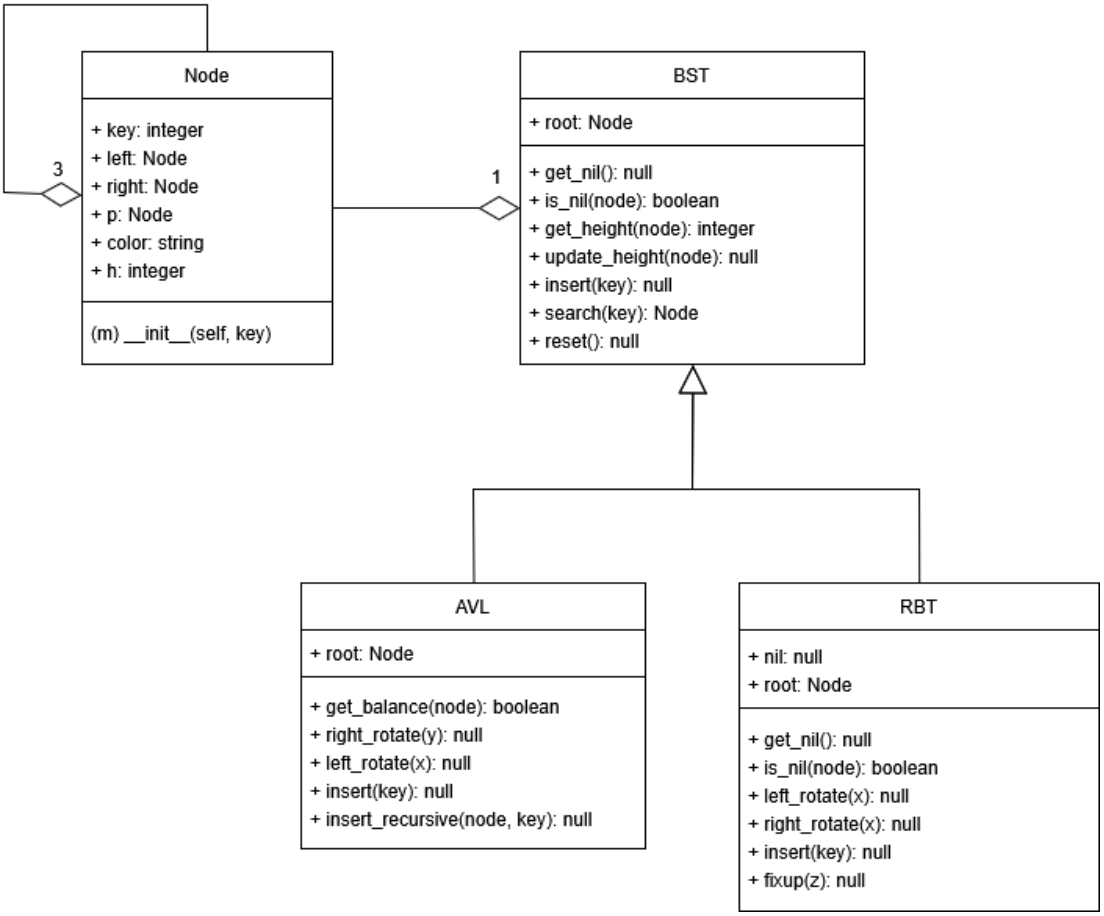


Figure 1: Diagramma UML delle classi utilizzate per i test

## 3.2 Metodi implementativi

- **Node**: è la classe che definisce tutte le informazioni che ogni nodo generico può contenere. Si è preferito definire solo una classe Nodo comune a tutte le tipologie di alberi per motivi di semplicità.
- **BST**: Binary Search Tree. E' la struttura dati più semplice.
  - get_nil: metodo ausiliario che, in base al tipo di albero che lo chiama, restituisce il valore nullo appropriato;
  - is_nil: metodo ausiliario booleano che indica se un nodo è nullo oppure no;
  - get_height: restituisce l'altezza di un nodo;
  - update_height: aggiorna l'altezza di un nodo sulla base di quella dei suoi figli;
  - insert: inserisce un nodo nell'albero;
  - search: ritorna, se presente, il nodo che si cerca;
  - reset: svuota un albero annullando la radice;
- **AVL**: estende il BST, ridefinendo i metodi che gestiscono l'inserimento e le rotazioni, così da soddisfare le sue proprietà.
  - get_balance: calcola il fattore di bilanciamento definito nella sezione teorica;

  - right_rotate: gestisce la rotazione in senso orario;
  - left_rotate: gestisce la rotazione in senso antiorario;
  - insert: metodo pubblico che gestisce l'inserimento di un nuovo nodo;
  - insert_recursive: è chiamato dal metodo *insert()* e si occupa di inserire effettivamente il nodo nell'albero

- **RBT**: estende il BST, ridefinendo i metodi che gestiscono l'inserimento e le rotazioni, così da soddisfare le sue proprietà.
  - get_nil: ridefinisce il metodo in base alle caratteristiche dell'albero;
  - is_nil: ridefinisce il metodo in base alle caratteristiche dell'albero;
  - right_rotate: gestisce la rotazione in senso orario;
  - left_rotate: gestisce la rotazione in senso antiorario;
  - insert: metodo pubblico che gestisce l'inserimento di un nuovo nodo;

# 4 Esperimento e risultati attesi

Tramite questo esperimento si vogliono verificare le complessità asintotiche esposte nella sezione teorica, misurando il tempo di esecuzione $T(n)$ al variare del numero di nodi $n$. L'esperimento è strutturato come segue.

Inserimento e ricerca sono indipendenti. Vengono effettuati quindi in tempi diversi e con sequenze numeriche in ingresso diverse (tranne per il caso peggiore, poiché questo è comune a entrambi).

Ciascun test viene effettuato aumentando, ad ogni iterazione, il numero di elementi da inserire/analizzare. Il numero totale delle iterazioni indica il numero di punti che comporrà ciascun grafico, e questo viene stabilito a priori impostando il valore dei parametri dedicati.

Ogni iterazione restituirà il tempo impiegato sia per il caso peggiore che per quello medio.

Il tempo che viene restituito deve dipendere il meno possibile dalle condizioni circostanziali che influenzano la macchina. Ogni iterazione verrà quindi ripetuta più volte, senza mai cambiare i dati in ingresso, replicando così le condizioni iniziali.

Ciascun gruppo di tempi ottenuti tramite lo stesso input viene rielaborato, sfruttando l'operatore della mediana, e ottenendo quindi un unico valore che sia il meno possibile influenzato da condizioni esterne.

Una volta che ho finito le misurazioni temporali per ogni iterazione di ogni operazione, i risultati vengono salvati in un apposito foglio di calcolo, in modo da poter essere utilizzati per realizzare i grafici.

## 4.1 Scenari di Test e Complessità Attesa

Vengono definiti due scenari per confrontare le caratteristiche delle diverse strutture dati:

#### Scenario A: Sequenza Ordinata (Caso Peggiore)

Vengono inseriti elementi in ordine strettamente crescente $(0, 1, ..., n)$.

- **BST:** Data l'assenza di controlli, l'albero dovrebbe degenerare, raggiungendo un'altezza pari al numero di nodi inseriti $h = n$. Il risultato asintotico, sia per la costruzione che per la ricerca, sarà $O(n)$, mentre l'altezza sarà $\theta(n)$
- **RBT e AVL:** Le strutture dovrebbero bilanciarsi grazie alle rotazioni, mantenendo un'altezza logaritmica e quindi prestazioni asintotiche pari a $O(\log n)$. Questo permetterà di mantenere un tempo logaritmico sia per l'inserimento che per la ricerca. I tempi per le due operazioni saranno estremamente inferiori rispetto ad un comune BST.

#### Scenario B: Sequenza Casuale (Caso Medio)

Vengono inseriti elementi casuali. Il BST dovrebbe quindi auto bilanciarsi nel lungo periodo. Il risultato atteso è che tutte le strutture (BST, AVL e RBT) mantengano un'altezza logaritmica, con prestazioni comparabili $O(\log n)$.

Per quanto riguarda l'inserimento, i tempi per BST potrebbero essere anche più bassi, in quanto l'assenza di controlli e rotazioni dovrebbe compensare il fatto che la struttura non sia perfettamente bilanciata.

Nel caso della ricerca, i tempi saranno a favore dell'albero che è bilanciato meglio. Da AVL mi aspetto quindi i tempi più bassi, seguito da RBT e, più distaccato, BST.

## 4.2 Tabella riassuntiva

La Tabella che segue riassume i risultati asintotici attesi e che dovranno essere verificati tramite i grafici derivanti dai test.

In generale, gli AVL hanno una struttura rigida da realizzare, ma ottimizzano lo spazio. Gli RBT, invece, sono più flessibili per quanto riguarda i controlli, ma l'organizzazione dei dati in memoria sarà peggiore. Nonostante i tempi effettivi siano molto simili, mi aspetto che gli AVL siano migliori per la ricerca, mentre gli RBT per l'inserimento.

| **Struttura** | **Operazione** | **Caso Peggiore (Ordinato)** | **Caso Medio (Casuale)** |
|---|---|---|---|
| BST | Inserimento | $\mathcal{O}(n)$ | $\mathcal{O}(\log n)$ |
| | Ricerca | $\mathcal{O}(n)$ | $\mathcal{O}(\log n)$ |
| | Altezza | $\mathcal{O}(n)$ | $\mathcal{O}(\log n)$ |
| AVL | Inserimento | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| | Ricerca | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| | Altezza | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| RBT | Inserimento | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| | Ricerca | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |
| | Altezza | $\mathcal{O}(\log n)$ | $\mathcal{O}(\log n)$ |

# 5 Risultati esperimento

## 5.1 Inserimento

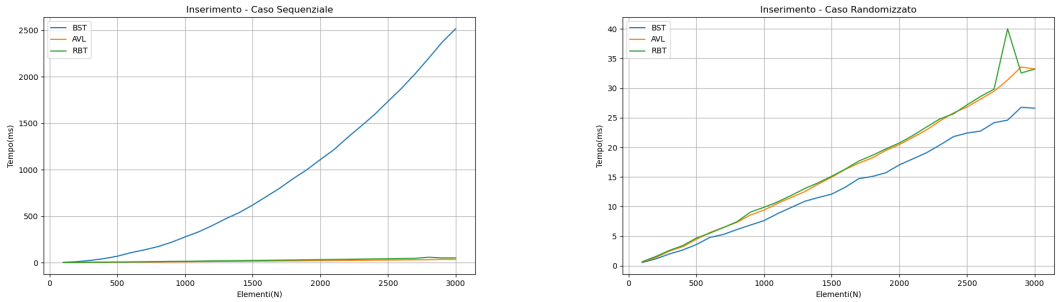


Figure 2: Tempi di inserimento di N elementi. Nel caso a gli elementi sono ordinati, nel caso b gli elementi sono casuali.

Come ipotizzato in partenza, il caso peggiore ha fatto degenerare il BST, portando a tempi sempre peggiori con l'aumentare degli elementi ospitati nella struttura.

Nel caso generale, invece, l'assenza di controlli ha giocato a suo favore, portando a tempi di inserimento più bassi rispetto alle altre due tipologie.

## 5.2 Altezza

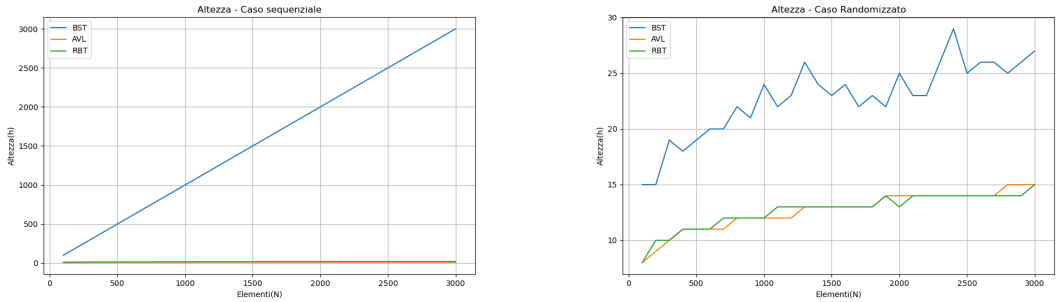


Figure 3: Confronto altezze per gli inserimenti. Nel caso a gli elementi sono ordinati, nel caso b gli elementi sono casuali.

La differenza di organizzazione dei dati in memoria è molto evidente. Si notano le carenze di BST, specialmente nel caso peggiore, da cui si evince che la crescita dell'altezza è lineare rispetto al numero dei dati in ingresso.

Nonostante le diverse implementazioni di AVL e RBT, la loro efficienza nel gestire al meglio lo spazio richiesto è molto simile. Questo dipende probabilmente dalla mole non troppo elevata dei dati da inserire.

Il grafico generato nel caso peggiore permette di visualizzare meglio le qualità della proprietà di bilanciamento implementata in AVL.

### 5.3 Ricerca

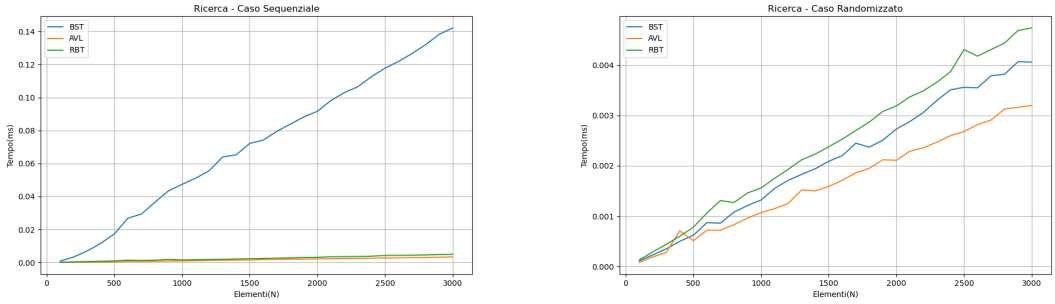


Figure 4: Tempi di ricerca di N elementi. Nel caso a gli elementi sono ordinati, nel caso b gli elementi sono casuali.

Per quanto riguarda la ricerca, la differenza nei tempi impiegati da ciascuna struttura dati per portare a termine i test è più evidente.

Nel caso peggiore, si evince in maniera abbastanza accurata la linearità del tempo impiegato da BST. Nel caso medio, invece, l'efficienza è assottigliata dai dati che costituiscono le strutture, anche se AVL rimane il migliore per questa operazione.

## 6 Conclusioni

Osservando i risultati dei test effettuati, possiamo notare che, per quanto riguarda il caso peggiore, le prestazioni sia per l'inserimento che per la ricerca calano drasticamente se implementate utilizzando un albero binario di ricerca. Lo stesso vale per l'altezza dell'albero, in quanto l'input particolare azzera tutti i vantaggi che un BST offrirebbe, sfociando in una lista concatenata.

Analizzando invece il caso medio, l'albero binario di ricerca tende a replicare i vantaggi delle altre due tipologie di alberi, talvolta anche con miglioramenti.

Questo lo rende quindi una struttura dati molto valida ed efficiente, specialmente se si punta alla semplicità, purché l'input che gli si fornisce sia il più possibile casuale.

Se il motivo per cui si deve utilizzare un albero come struttura dati è la ricerca, un AVL sarà la scelta migliore. Anche perché, per notare gli svantaggi nell'inserimento, è richiesta una mole di dati notevole.

Nel complesso, RBT rimane comunque la scelta migliore tra le 3, riuscendo a mitigare le caratteristiche di velocità di un BST, noncurante del bilanciamento della struttura complessiva, con quelle di organizzazione in memoria di un AVL, con controlli e rotazioni mirate a ottimizzare il più possibile la ricerca.