Università degli Studi di Udine

Scienze Informatiche

Laboratorio di Algoritmi e Strutture Dati
Anno: 2019/2020

Riccardo Ferrarese

mat. 140491

ferrarese.riccardo@spes.uniud.it

Contents

1 Introduzione

Il seguente elaborato propone tre diverse implementazioni per gli alberi.

Le implementazioni sono state realizzate in linguaggio *C++* e grafici utilizzando *R*.
Nelle seguenti sezioni verranno presentate ed analizzate le soluzioni.
L'analisi dei tempi cerca di ridurre al minimo le variazioni nei tempi calcolati facendo eseguire più volte l'algoritmo su un input fissato per raggiungere una determinata precisione nella misurazione. Inoltre avendo implementato tutte le operazioni di ricerca con l'ipotesi che l'elemento da cerca sia presente, per l'analisi dei tempi ammortizzati, che richiede di inserire un elemento se non presente, è stata leggermenete modifica la procedura di *find* con un valore di ritorno *true* se l'elemento è presente, *false* altrimenti.
Si rimanda all'ultima sezione per la spiegazione di come si sono generati i campioni per l'analisi dei tempi.

2 Binary Search Tree - BST

Descrizione

Un albero binario di ricerca è un particolare tipo di albero binario. Ogni nodo è costituito dai seguenti campi:

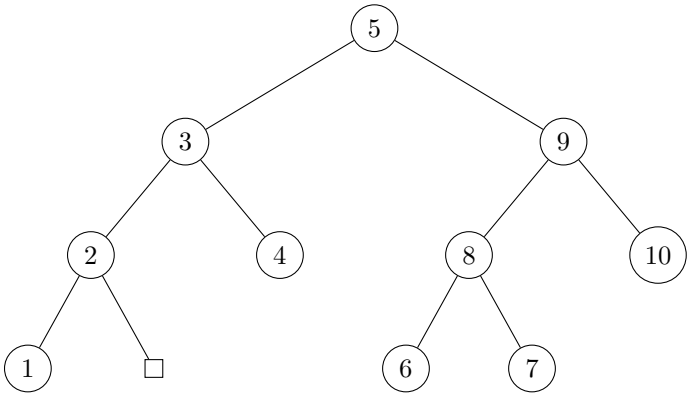
- key → valore della chiave (di tipo intero)
- value → valore alfanumerico (di tipo stringa)
- left right → puntatori ai rispettivi figli sinistro e destro.

Non abbiamo la necessità di mantenere in memoria un puntatore per il nodo "genitore", risparmiando spazio in memoria.

Le chiavi vengono memorizzate in modo tale da garantire le seguenti proprietà:
Sia x un nodo in un albero binario di ricerca.

- se y è un nodo nel sottoalbero sinistro di x, allora $key[y] \leq key[x]$.
- se y è un nodo nel sottoalbero destro di x, allora $key[x] \geq key[y]$.
- sottoalbero sinistro e destro di x sono a loro volta dei binary search tree.

Queste proprietà forniscono un ordine delle chiavi che permettono a operazioni come la ricerca e trovare minimo o massimo degli elementi, di esser fatte più velocemente rispetto la ricerca su un vettore.
Si fornisce di seguito un esempio di BST.



Operazioni

Ricerca

La ricerca di una chiave in un BST risulta molto simile a una ricerca binaria. Quello che si deve fare è, partendo dalla radice, confrontare il valore da cercare con il valore del nodo in cui si è arrivati, se è uguale abbiamo finito la ricerca, se è minore si deve scendere nel sottoalbero sinistro, viceversa se è maggiore si deve scendere nel sottoalbero destro. La decisione su che ramo prendere è fatta essendo garantita la proprietà che tutti gli elementi più piccoli della chiave che stiamo confrontando si trovano nel sottoalbero sinistro e rispettivamente quelli più grandi nel sottoalbero destro.
Se l'albero risulta bilanciato, si inizia la ricerca in un spazio di n nodi e quando si sceglie uno dei sottoalberi escludiamo $\frac{n}{2}$ nodi, riducendo lo spazio di ricerca a $\frac{n}{2}$. Nello step successivo riduciamo ancora lo spazio di ricerca a $\frac{n}{4}$ e così facendo fino a trovare la chiave cercato o a ridurre lo spazio di ricerca a un nodo solo. Questo permette di eseguire una ricerca in tempo $\mathcal{O}(h)$, con h altezza dell'albero.
Nel caso in cui si ha un albero coompleto con n nodi, la ricerca richiede al massimo un tempo $\theta(log(n))$ se la chiave da cercare si trova su una foglia.
Nel caso pessimo, se l'albero risulta sbilanciato su un ramo solo (catena lineare), la ricerca ha complessità $\theta(n)$.

Inserimento

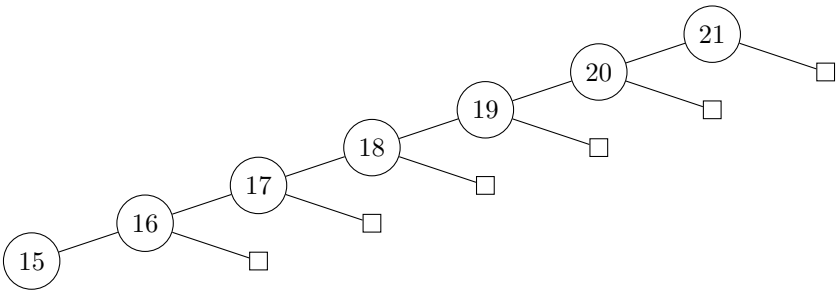
Nell'operazione di inserimento un nuovo nodo viene sempre inserito come foglia. Viene eseguita una ricerca dalla radice confrontando la chiave per trovare il ramo corretto in cui cercare il nodo foglia a cui verrà legato il nuovo nodo da inserire. Similmente all'operazione di ricerca, anche l'inserimento richiede un tempo di eseguzione proporzionale all'altezza h dell'albero ($\mathcal{O}(h)$), dovendo eseguire una ricerca fino ai nodi foglia.
Nel caso di un albero completo con n nodi l'operazione ha complessità $\theta(log(n))$, viceversa nel caso di un albero sbilanciato $\mathcal{O}(n)$.

Analisi

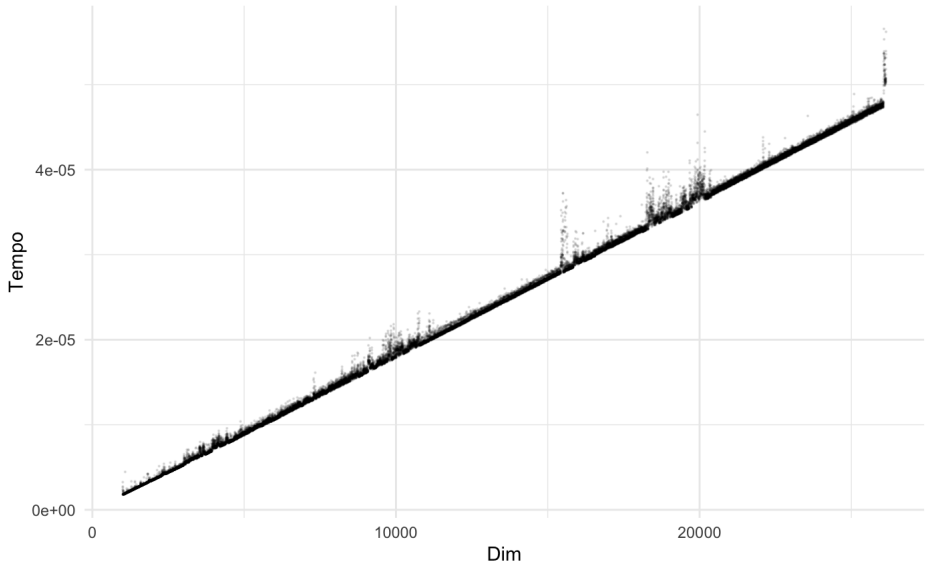
Per analizzare i tempi di inserimento in un BST si è deciso di studiare per prima cosa il caso peggiore, in cui vengono inserire chiavi tali da costruire una catena lineare. Per rappresentare un nodo si utilizza una rappresentazione in cui la coppia $x : y$, rappresenta la coppia $key : value$ del nodo; per semplicità viene inserito lo stesso valore. Si dia un esempio della successione di operazioni di inserimento:

```
21:21 NULL NULL
21:21 20:20 NULL NULL NULL
21:21 20:20 19:19 NULL NULL NULL NULL
21:21 20:20 19:19 18:18 NULL NULL NULL NULL NULL
21:21 20:20 19:19 18:18 17:17 NULL NULL NULL NULL NULL NULL
21:21 20:20 19:19 18:18 17:17 16:16 NULL NULL NULL NULL NULL NULL NULL
21:21 20:20 19:19 18:18 17:17 16:16 15:15 NULL NULL NULL NULL NULL NULL NULL NULL
```

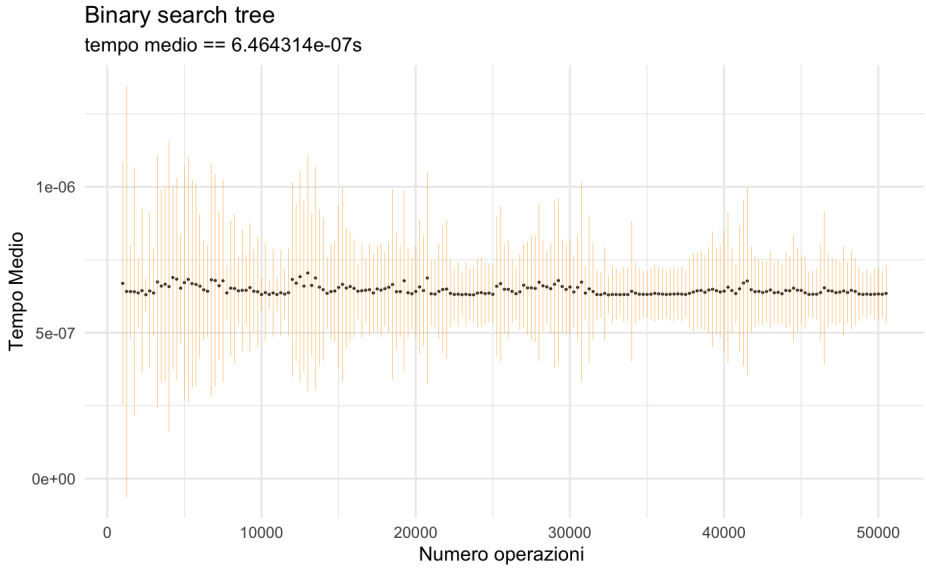
Graficamente la catena lineare costruita risulta:



Già graficamente si può notare come l'inserimento di un nuovo nodo più piccolo di 15 avverrà in tempo $\theta(n)$, con n numero di nodi presenti. Per confermare l'ipotesi si fa un'analisi del tempo ammortizzato.
Per riuscire ad analizzare il caso pessimo, si è deciso di fissare una k (key del nodo) con la quale fare inserimenti su un numero di nodi crescente. Avendo fissato un solo valore per la chiave, si procede con solo operazioni di inserimento, ed ogni iterazione di n (numero di nodi) verrà inserito un nuovo nodo come figlio destro, venendosi a formare una catena lineare. Si analizza il tempo di esecuzione ammortizzato ($\frac{tempototalediesecuzioneinserimento}{numerodiinserimenti} = \frac{t\_tot}{m}$) su un numero di nodi crescente.
In questo caso l'intervallo in cui generare k è ininfluente, essendo che per ogni albero creato verrà usata una sola chiave. Il grafico risultante rispecchia l'analisi teorica sul caso pessimo.



Per ogni valore N, numero dei nodi, si sono generati 100 campioni diversi su cui misurare i tempi ammortizzati. Per ogni N viene quindi misurato il tempo medio e la standard deviation sui campioni misurati.



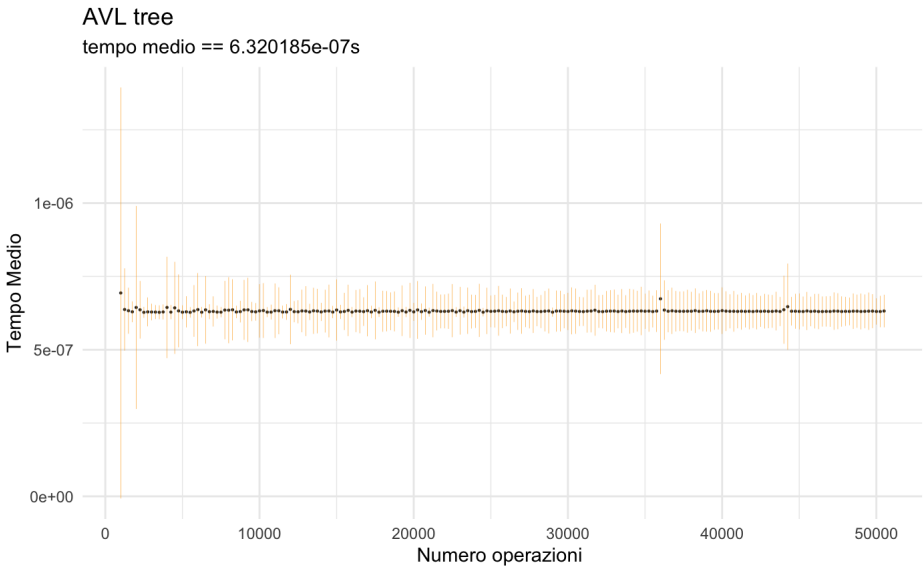
Come si può notare dal grafico, i tempi non sono uniformi al crescere delle operazioni non essendo garantito il bilanciamento dell'albero, indipendentemente dalla sequenza con cui i valori vengono inseriti.

3 AVL Tree

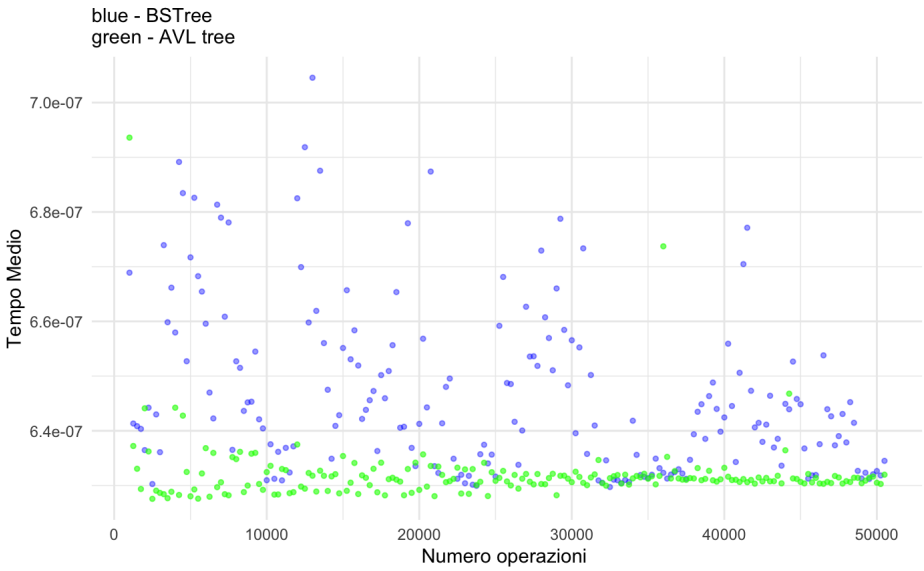
Un albero AVL è un binary search tree nel quale la differenza delle altezze del sottoalbero sinistro e destro per ogni nodo differiscono di al più un unità.
Ogni nodo contiene informazione sull'*altezza* per non aumentare la complessità calcolandola ogni volta. In questo modo si può calcolare il fattore di bilanciamento ($\rightarrow height(left) - height(right)$) che può prendere valore 1, -1 o 0, a seconda se l'altezza del sottoalbero di sinistra è maggiore, minore o uguale all'altezza del sottoalbero di destra.
A seguito delle operazioni di ribilanciamento, un albero di ricerca di tipo AVL mantiene un'altezza proporzionale al logaritmo del numero dei suoi nodi. In questo modo tutte le operazioni supportate avranno complessità logaritmica rispetto al numero di nodi.

Analisi

Un albero AVL consumerà più memoria rispetto a un BST, essendo che ogni nodo ricorderà la sua altezza. Inoltre le operazioni saranno più lente dovendo calcolare il *balance factor* ad ogni inserimento e facendo operazioni di rotazione. È preferibile a un *binary search tree* garantendo il costo delle operazioni di ricerca e inserimento in $\mathcal{O}(logn)$ in qualsiasi caso. Si mostra il grafico dei tempi ammortizzati.



Confrontando i risultati con quelli del BST si può notare la differenza di avere un albero sempre bilanciato.

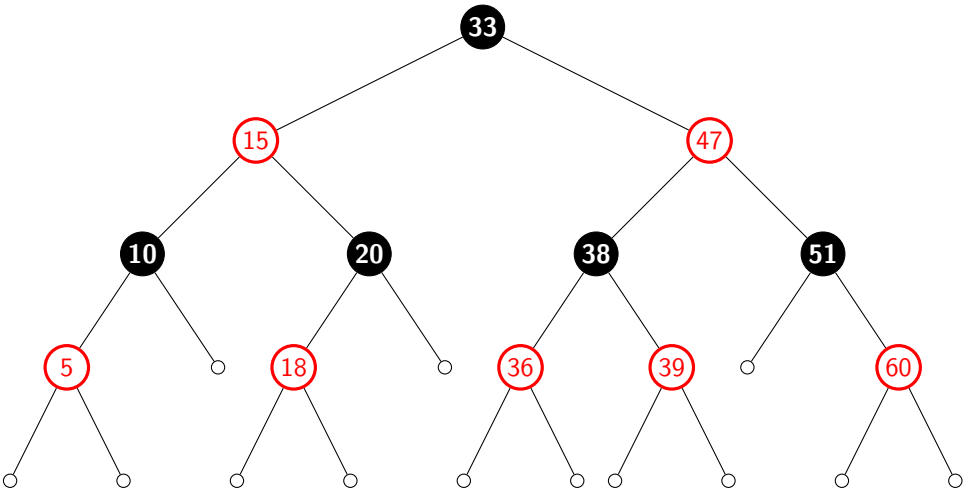


4 Red-Black Tree

Un albero red-black è un binary search tree con un campo aggiuntivo per ogni nodo che ne specifica il *colore*, che può essere impostato a rosso oppure nero.
Se manca un figlio o il padre di un nodo il corrispondente campo puntatore del nodo contiene un nodo di valore NIL di colore nero.
Un albero binario di ricerca è un red-black tree se soddisfa le seguenti proprietà:

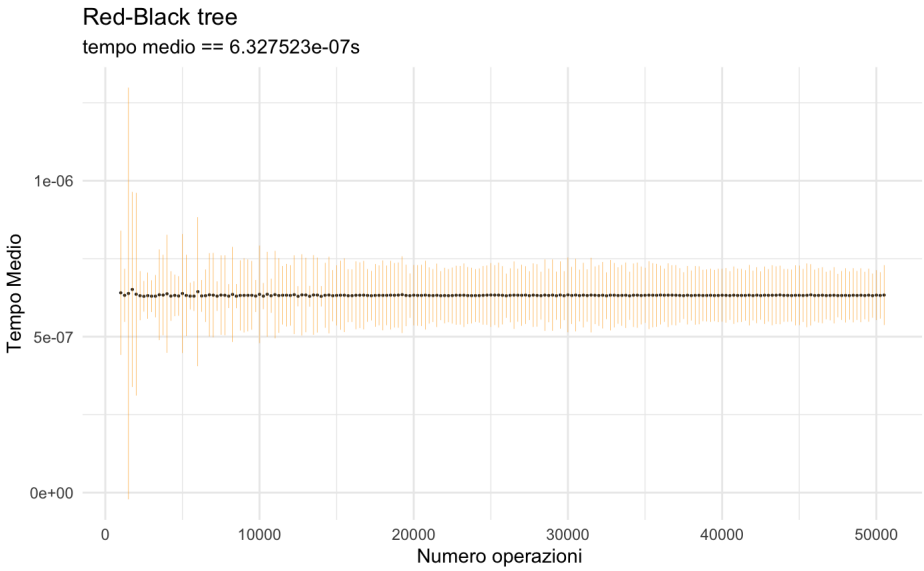
- Ogni nodo è rosso o nero.
- La radice è nera.
- Le foglie sono NIL e sono nere (non hanno chiave).
- Se un nodo è rosso allora entrambi i figli sono neri e ogni nodo interno ha due figli.
- Per ogni nodo x, i cammini semplici che vanno da x alle foglie del suo sottoalbero contengono lo stesso numero di nodi neri.

Queste proprietà permettono di mantenere l'albero bilanciato, in questo modo l'altezza di un albero di n nodi è pari a $\mathcal{O}(log(n))$.
Di seguito un esempio di un red-black tree.

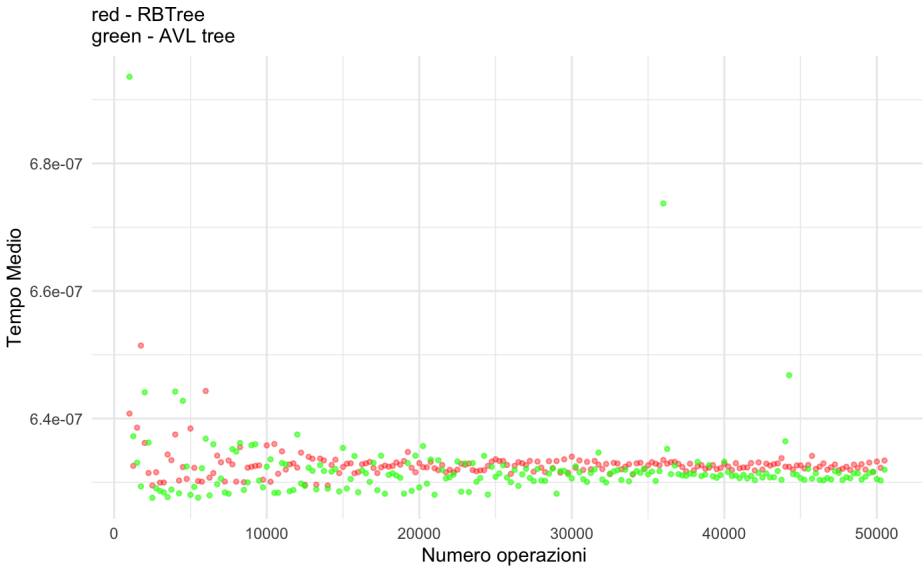


Analisi

Di seguito il grafico dei tempi ammortizzati per un RBTree.

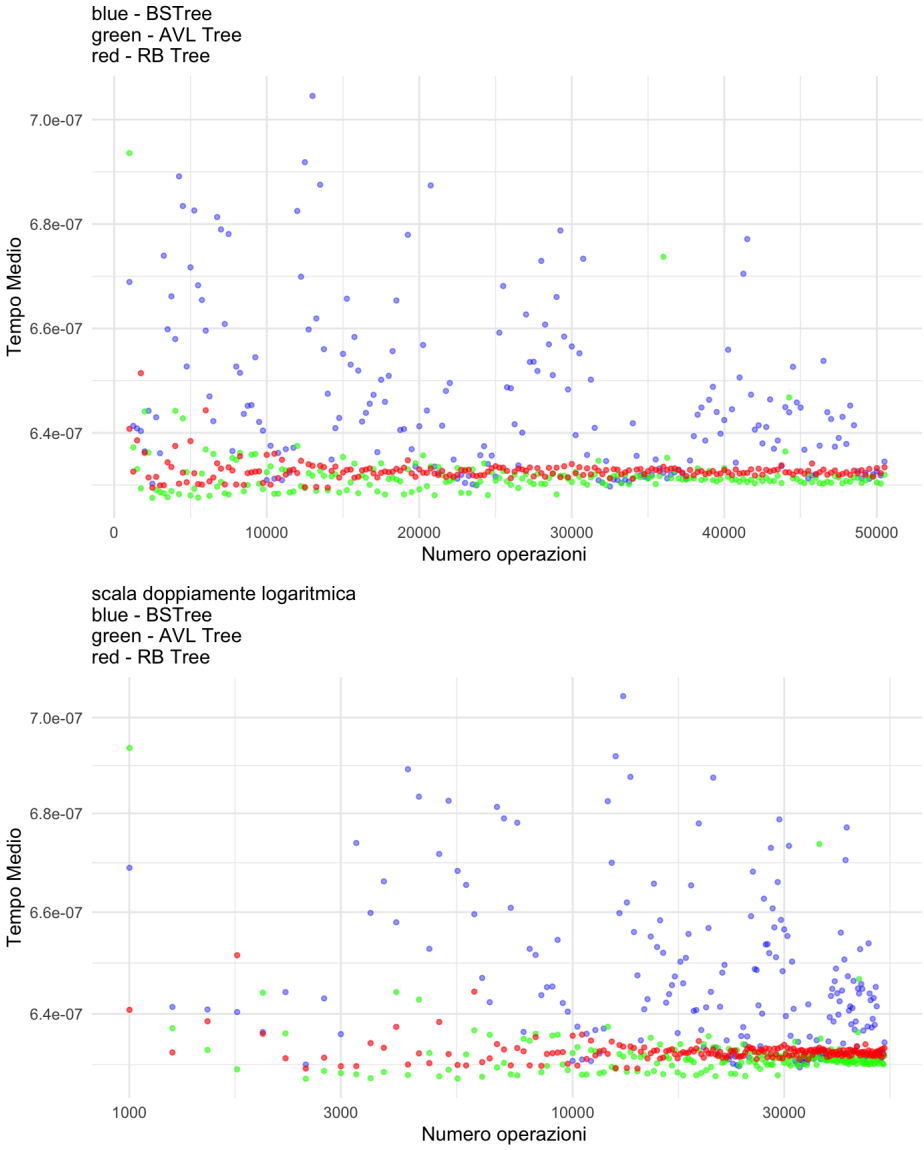


Confrontando con i tempi raccolti per gli alberi AVL, si può notare come i Red-Black Trees, a parità di operazioni svolte, non garantiscano un bilanciamento ottimale come gli AVL. Anche se, in alcuni casi dovuti all'ordine di inserimento, a causa delle rotazioni in più da fare, un albero AVL avrà costo di tempo maggiore ripsetto ai RB Trees.



5 Confronti e osservazioni

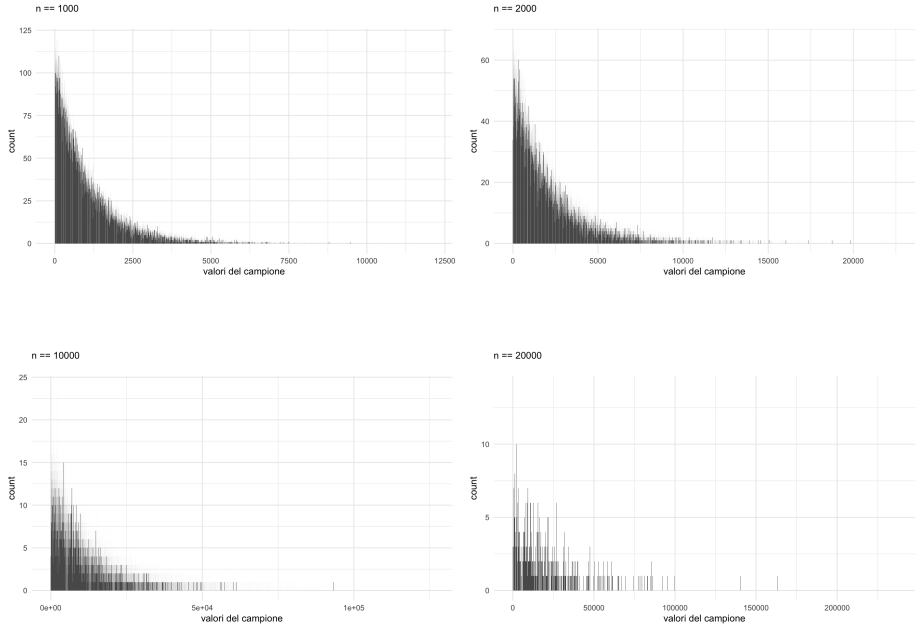
Gli alberi AVL sono più bilanciati rispetto ai Red-Black trees, ma causano più rotazioni durante gli inserimenti. Quindi se per l'applicazione è neccessario un numero frequente di inserimenti (e cancellazioni), i Red-Black trees sono da preferire. Al contrario se si hanno più operazioni di ricerca rispetto alle altre, sono da preferire gli AVL trees.
A livello di memoria i Red-Black trees hanno un bit extra per l'informazione del colore (0 o 1, rispettivamente per rosso o nero), gli alberi AVL devono invece mantenere un valore intero come altezza per ogni nodo.
Di seguito un grafico comparativo tra le tre implementazioni degli alberi.



Il grafico può confermare che un albero bilanciato ha tempi di esecuzione più efficienti e, facendo riferimento a quando detto sopra, la scelta tra AVL e RB dipende dal campo di utilizzo avendo tempi di esecuzione simili nella maggior parte dei casi.

6 Generazione dei campioni e modalità di analisi

Per l'analisi dei tempi, per ogni N (numero di ricerche da eseguire), si è scelto di generare 100 campioni con una distribuzione geometrica, essendo una distribuzione discreta e generando numeri nel campo dei Naturali. Come parametro p si è scelto di inserire $1/N$ in modo tale da avere una distribuzione più densa, per valori di N bassi. A scopo dimostrativo si mostrano i grafici della distribuzione scelta al variare del parametro, generando 100_000 campioni con $p = \frac{1}{n}$.



Per l'analisi dei tempi ammortizzati per ogni N che varia da 100 a 100000, si è misurato il tempo medio e il numero medio di operazioni per 200 campioni con valori presi randomicamente dalla distribuzione geometrica generata per ogni campione. È stato tenuto un contatore per le operazioni di ricerca e inserimento, in questo modo si è ottenuta una misurazione coerente con il numero di operazioni fatte, generando valori casuali all'interno della distribuzione.
In tutte e tre le implementazioni, per la misurazione, è stata usata una versione iterativa della ricerca per non aver differenze dovute alla gestione della ricorsione.