UNIVERSITÀ DEGLI STUDI DI UDINE

Corso di Informatica e Internet of Things, Big Data and Web

Cognome e Nome	Numero Matricola
Mattia Guiotto	147654 147654@spes.uniud.it
Nicola Zerajic De Giorgio	149299 149299@spes.uniud.it
Mattia Novo	109538 109538@spes.uniud.it
Andrea Comisso	147984 147984@spes.uniud.it

Oggetto della prova

L'obiettivo primario di questo progetto è quello di mettere a confronto tre strutture dati e di analizzarne i tempi medi di esecuzione di due operazioni, al fine di comprendere quale delle tre strutture dati risulti la più efficace.

In particolare, le strutture dati prese in esame sono tre alberi binari di ricerca - BST, RBT e AVL - e le operazioni da esaminare sono la ricerca e l'inserimento di una determinata chiave.

Descrizione della prova

Un albero binario di ricerca è un albero binario (avente cioè al più due figli per ogni nodo) in cui per ogni nodo x di chiave k vale la seguente proprietà: ogni nodo radicato nel sottoalbero sinistro di x ha chiave minore di k e ogni nodo radicato nel sottoalbero destro ha chiave maggiore di k.

Da questa proprietà deriva che un'operazione di ricerca/inserimento ha costo asintoticamente proporzionale all'altezza dell'albero in quanto l'algoritmo di ricerca o inserimento di una chiave compirà un cammino dalla radice a una foglia, o a un determinato nodo, esaminando uno e un solo nodo per livello. Tale cammino avrà quindi una lunghezza massima pari all'altezza dell'albero.

Si deduce quindi che per valutare l'efficienza dei tre alberi sia necessario esaminare le relative altezze in funzione del numero di nodi che costituiscono la struttura dati.

Un albero bilanciato, ovvero avente più o meno la stessa quantità di nodi a destra e a sinistra di ogni nodo, sarà più efficiente di uno sbilanciato, cioè tendente ad avere un solo figlio per nodo e quindi una distribuzione non uniforme.

Sono state effettuate quindi due prove: una per il caso pessimo (cercando di sbilanciare il più possibile l'albero) e una per il caso medio.

Descrizione dell'algoritmo per la misurazione dei tempi

Per ricavare i grafici dei tempi per i tre alberi, sono state generate *n* chiavi pseudocasuali con *n* che varia da 1000 a 100.000 in una distribuzione formata da 100 punti. Per ognuna delle 100 iterazioni (100 punti) l'albero che si era creato dall'iterazione precedente è stato cancellando attraverso una procedura che dealloca in maniera ricorsiva ogni nodo dell'albero.

L'algoritmo, partendo da un albero vuoto, per ogni *i* da 0 a *n,* controlla se la chiave generata non sia già presente nell'albero (quindi fa un'operazione di ricerca) ed eventualmente la inserisce. Dato che le chiavi verranno generate in un intervallo molto grande è molto probabile che verranno effettuate *n* ricerche e *n* inserimenti.

Il caso medio è stato realizzato generando chiavi casuali nell'intervallo [0, INT_MAX]. Con un range così ampio, infatti è molto probabile che l'albero che andrà a formarsi non sarà troppo sbilanciato. Dato che in questo caso i tempi di esecuzione sono molto bassi e un intervallo dei valori di *n* da 1000 a 100.000 non fornirebbe un campione rappresentativo per un confronto fra i tre alberi, è stato scelto di aumentare il valore massimo di *n* a 1.000.000 di operazioni.

Il caso pessimo invece si ottiene inizializzando in partenza la chiave e, ad ogni iterazione di *n* (numero di operazioni di ricerca/inserimento), sommando ad essa un certo valore, nel nostro caso un numero tra 0 e 9999. In questo modo si avrà, nel caso dei BST, un albero completamente sbilanciato, con la chiave minima nella radice e quella massima nell'unica foglia.

Alla fine di ogni set di *n* operazioni viene calcolato il relativo tempo ammortizzato dividendo il tempo di esecuzione totale di *n* operazioni per *n*.

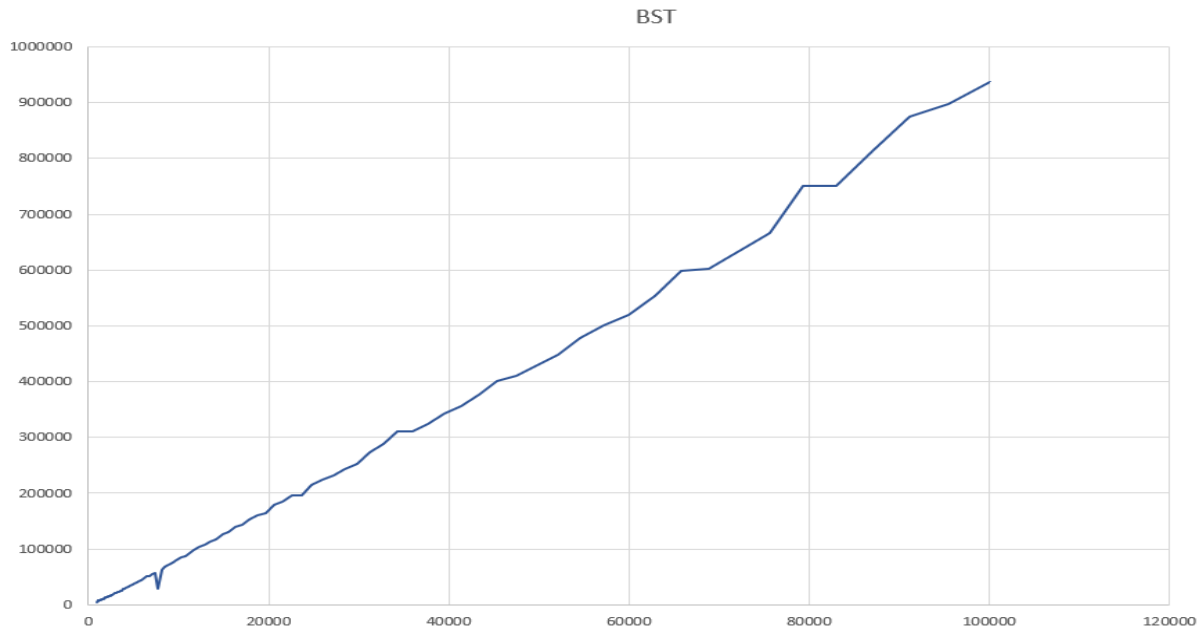
I tempi ammortizzati per le 100 iterazioni da *n* operazioni vengono scritti in un file di testo mediante l'uso delle funzioni di libreria del C.

- ***BST (semplice) – Caso pessimo***

  Il BST è un albero binario di ricerca semplice e la sua procedura di inserimento non esegue alcun controllo sul bilanciamento dell'albero. La sua altezza nel caso pessimo è quindi $theta(n)$, con *n* dimensione dell'albero. Di conseguenza le operazioni di ricerca e inserimento avranno costo $O(n)$.

  Il tempo ammortizzato registrato per l'esecuzione di *n* operazioni di ricerca/inserimento sarà dato da $\sum_{i=0}^{n-1} i = O(n^2)$ (tempo totale) diviso *n* => $O(n)$

  Il grafico ottenuto è il seguente:



Sulle ascisse è riportata la dimensione dei vari set di operazioni di inserimento/ricerca (da 1000 a 100.000) e le ordinate indicano il tempo di elaborazione in nanosecondi.

Si può notare che il grafico segue un andamento lineare, quindi al crescere del numero di operazioni sull'albero la durata della procedura cresce in modo costante.

- ***RBT – Caso pessimo***

  Un Red-Black Tree è un BST i cui nodi hanno un campo che definisce il loro colore che può essere RED o BLACK. Un RBT in aggiunta ha le seguenti proprietà:

  - la radice e le foglie NIL sono BLACK
  - ogni nodo RED ha due figli BLACK
  - per ogni nodo x, ogni cammino da x a una foglia contiene lo stesso numero di nodi BLACK

  Grazie a queste proprietà il RBT riesce ad essere sempre ben bilanciato. L'algoritmo di inserimento, infatti, dopo aver colorato di RED la chiave appena inserita, controlla se le tre proprietà sono verificate. Nel caso in cui non lo fossero effettua ricolorazioni o rotazioni dei nodi, a destra o a sinistra, in modo da spostare il "peso" da sinistra a destra e viceversa, ribilanciando quindi l'albero.

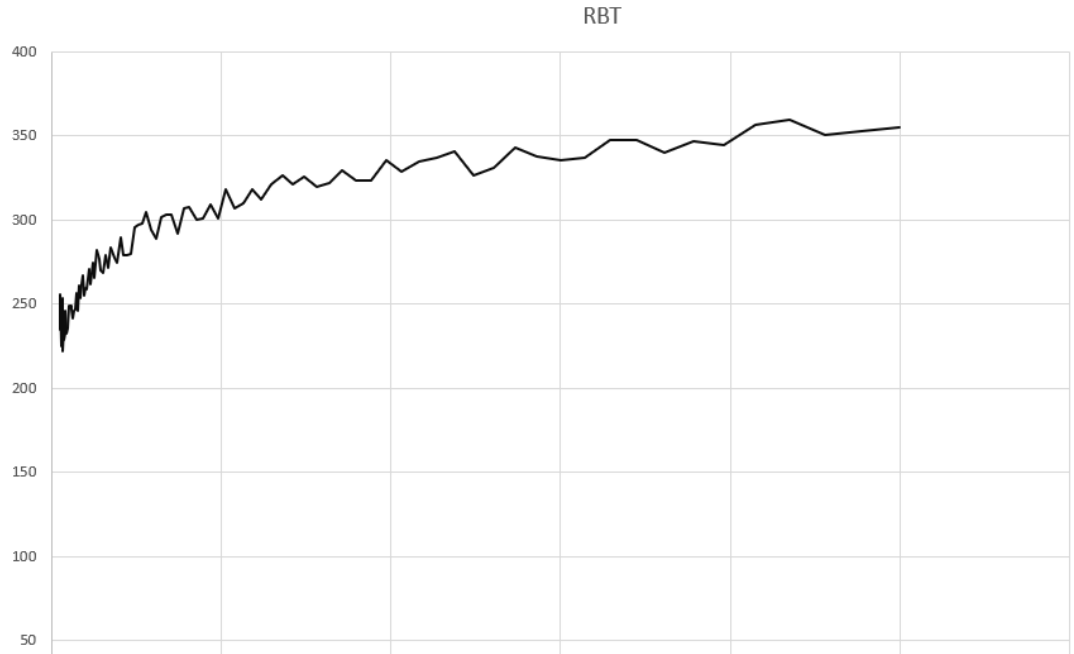
  Di conseguenza, se l'albero è bilanciato, in generale ogni nodo interno avrà due figli e quindi ad ogni livello dell'albero il numero di nodi raddoppierà, perciò un albero costituito da *n* nodi avrà $\log(n)$ livelli. Dato che il numero dei livelli di un albero rappresenta la sua altezza, quest'ultima avrà dimensione $O(\log(n))$.

  Ne deriva che il costo di ricerca/inserimento di una chiave sarà, anche nel caso pessimo, $O(\log(n))$.

  Le rotazioni consistono in semplici modifiche dei puntatori ai figli e al genitore del nodo perno e pertanto impiegano tempo costante $theta(1)$.

  Il tempo ammortizzato per l'esecuzione di *n* operazioni in un albero di dimensione $O(n)$ sarà quindi $\sum_{i=0}^{n-1} \log(i) = O\big(nlog(n)\big)$ diviso *n* => $O\big(log(n)\big)$

  Il grafico ottenuto dai risultati delle misurazioni dei tempi è il seguente:



Il grafico descrive una curva logaritmica, pertanto all'aumentare del numero di operazioni la durata della procedura aumenta molto lentamente.

- ***AVL – Caso pessimo***

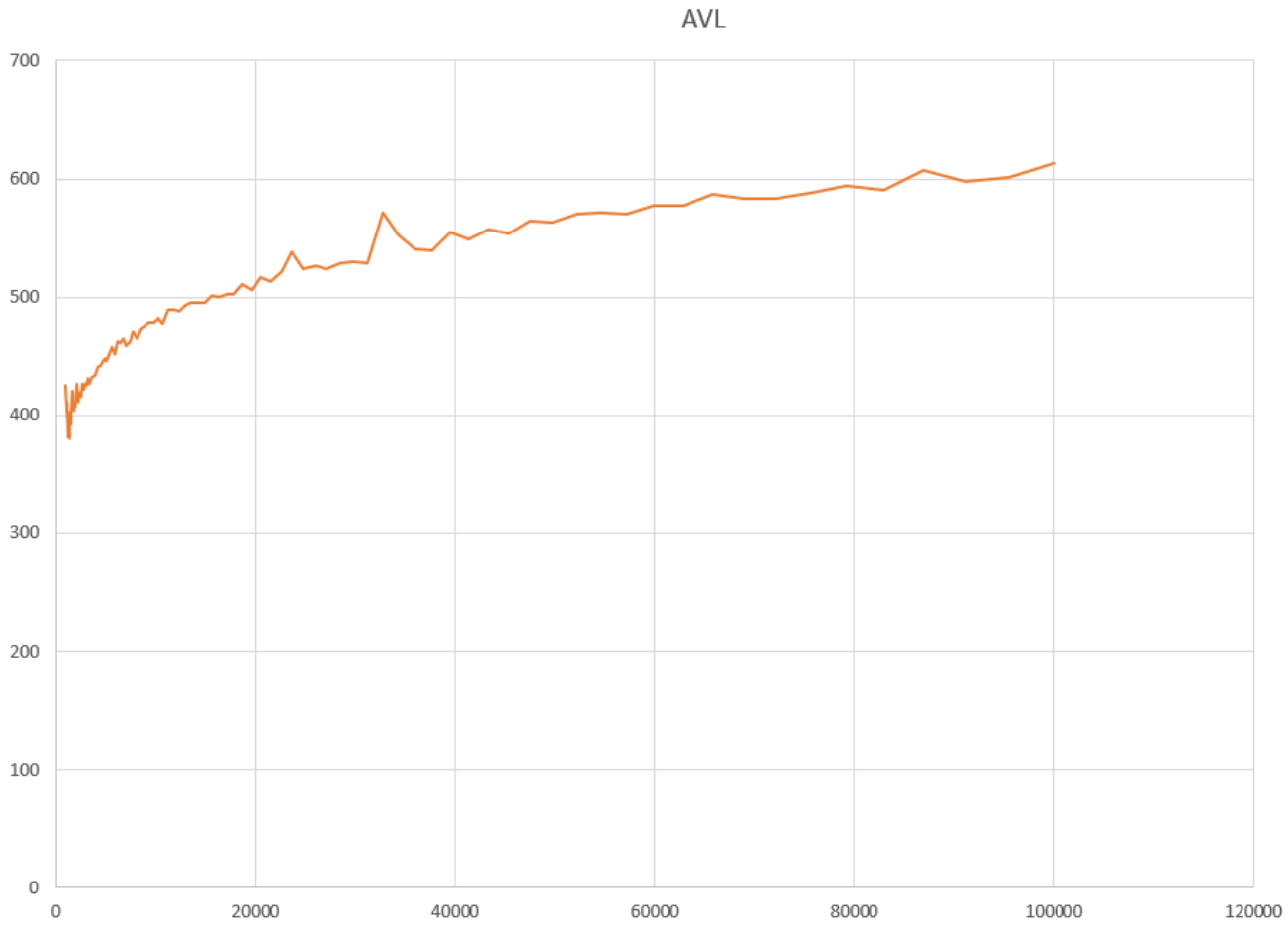
  Un albero AVL è un albero binario di ricerca in cui i nodi hanno un campo aggiuntivo per indicare la propria altezza, definita dal massimo tra l'altezza del sottoalbero sinistro e destro più 1. Un albero di un solo elemento ha altezza 1.

  Per mantenere un buon bilanciamento gli AVL possiedono la seguente proprietà: per ogni nodo x, la differenza in valore assoluto tra l'altezza del sottoalbero sinistro (Hsx) e quella del sottoalbero destro (Hdx) deve essere minore o uguale a 1.

  Come per gli RBT, gli AVL per mantenere questa proprietà effettuano rotazioni dei nodi. Se la differenza delle altezze Hsx – Hdx è minore di -1 allora l'albero radicato in x è sbilanciato a destra e quindi andrà effettuata una rotazione a sinistra con x nodo perno, viceversa sarà necessario ruotare l'albero a destra.

  È fondamentale che dopo ogni operazione di inserimento o rotazione la procedura risalga l'albero aggiornando tutte le altezze.

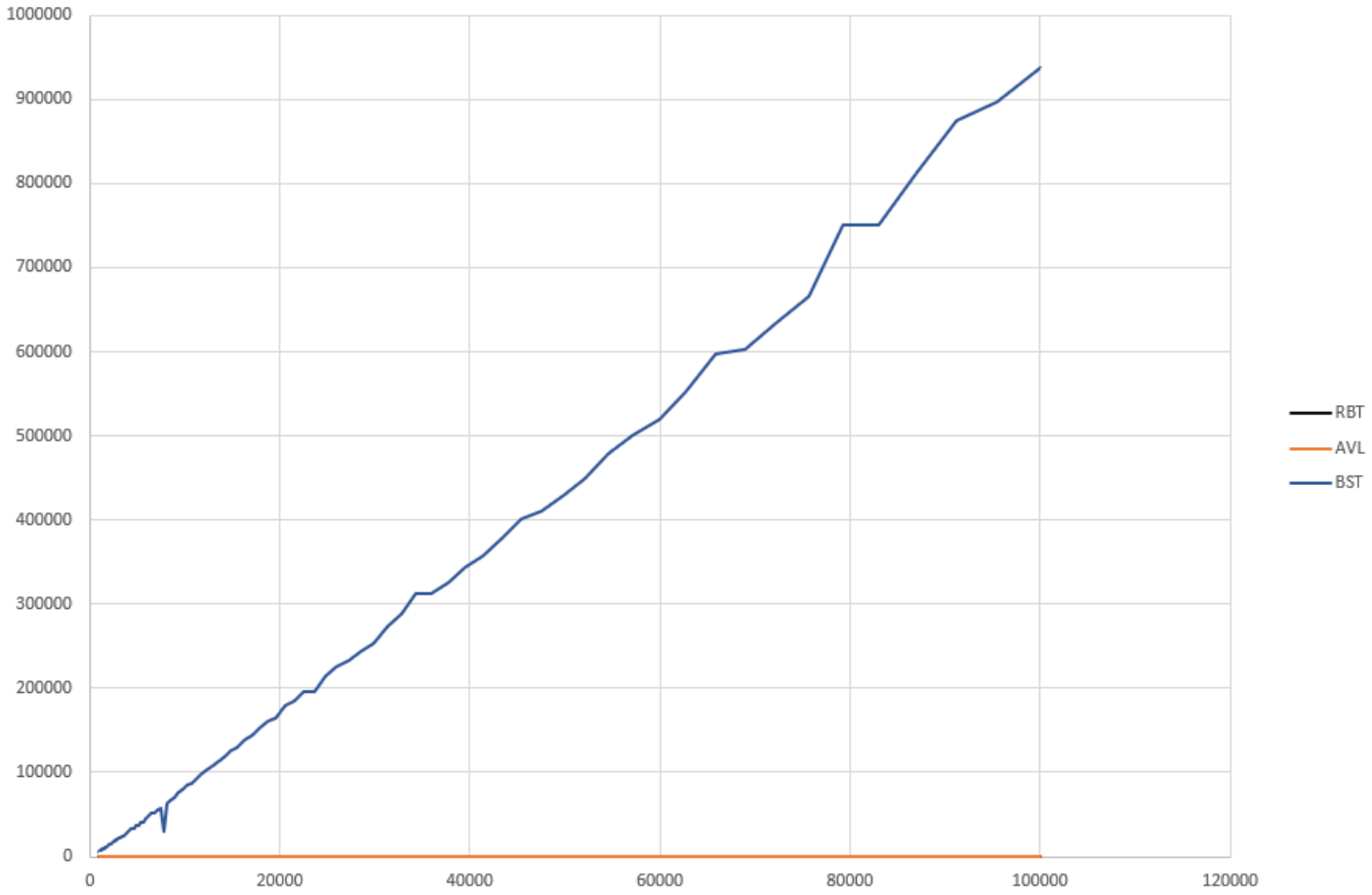
  Grazie a queste operazioni, l'albero AVL riesce, analogamente agli RBT, a restare bilanciato anche nel caso pessimo. Anche in questo caso avremo quindi un costo per le operazioni di ricerca e inserimento logaritmico rispetto alla dimensione dell'albero.



Come nel grafico degli RBT è evidente l'andamento logaritmico. Rispetto ai Red-Black Tree, però, i tempi di esecuzione sono più alti negli AVL. Questo probabilmente è conseguenza del fatto che per ogni inserimento la procedura degli AVL può risalire più volte l'albero per sistemare le altezze.

Confronto BST, RBT, AVL – Caso pessimo

Unendo i tre grafici si ottiene il seguente:

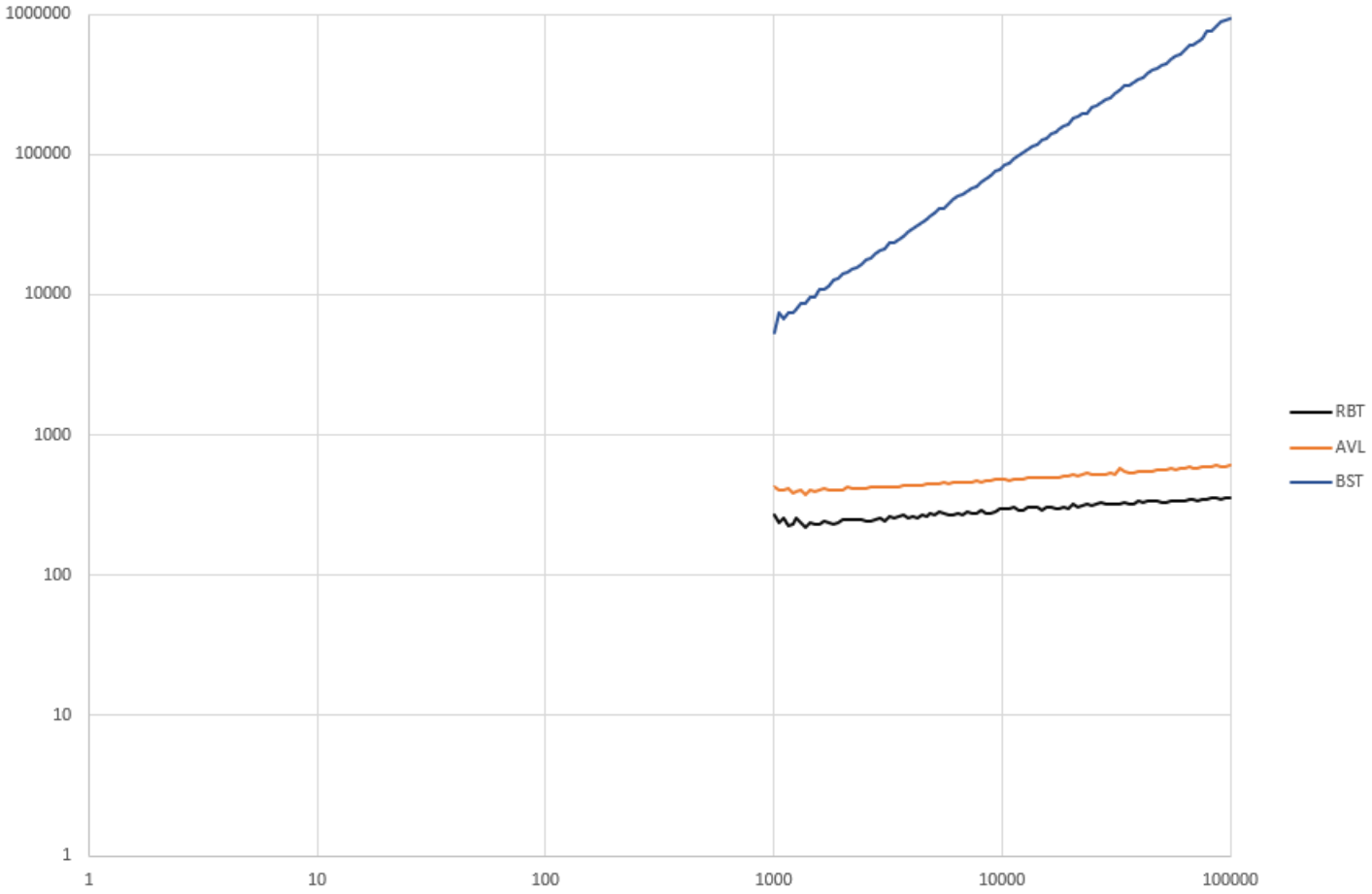


Da questo grafico si può notare che nel caso pessimo l'albero BST è estremamente inefficiente rispetto agli AVL e RBT.

Analizzando il rapporto tra i tempi di esecuzione degli RBT e quelli dei BST, notiamo che questi ultimi sono da 20 a 2600 volte più lenti:

num operazioni	BST	RBT	rapporto
1000	5377	272	19,77
1048	7455	235	31,72
1097	6697	256	26,16
1150	7418	225	32,97
1205	7414	229	32,38
1262	7893	254	31,07
1322	8658	234	37,00
1385	8756	222	39,44
68926	602029	348	1729,97
72208	633872	340	1864,33
75646	665653	347	1918,31
79248	751436	345	2178,08
83022	751007	357	2103,66
86975	812594	360	2257,21
91116	873967	351	2489,93

A supporto del risultato ottenuto dal precedente grafico di confronto risulta utile analizzare anche il grafico in scala doppiamente logaritmica che esprime i vari ordini di grandezza:



Si nota che le rette relative all'AVL e al RBT sono parallele, a conferma del fatto che i due costi sono entrambi logaritmici.

La retta relativa al BST, invece, ha una pendenza molto più marcata.

Questa caratteristica è dettata dal diverso valore assunto dai coefficienti angolari, che corrispondono alla velocità di risposta delle varie strutture dati al crescere del numero di operazioni da eseguire.

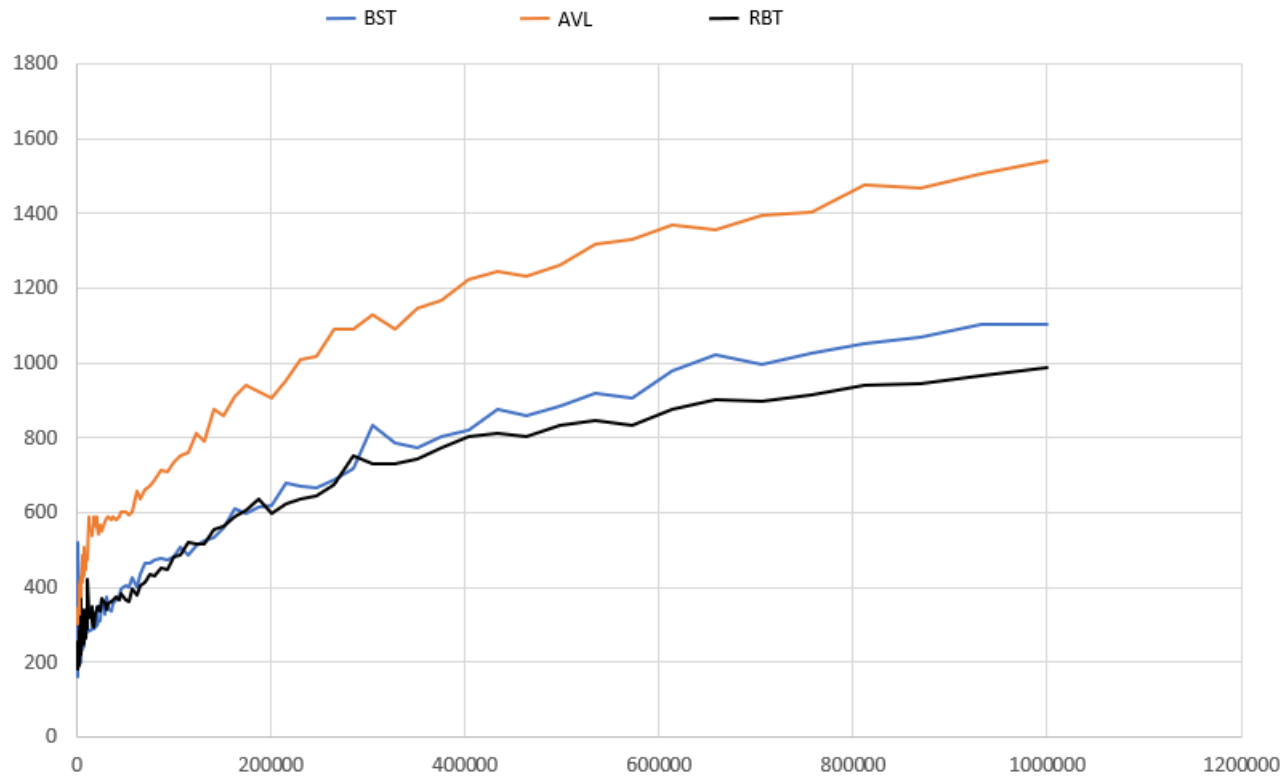
AVL e RBT risultano avere pendenza inferiore, quindi sono in grado di soddisfare le richieste con tempi di risposta inferiori rispetto al BST.

Confronto BST, RBT, AVL – Caso medio

Il caso medio si verifica quando non c'è un particolare schema nella sequenza degli inserimenti delle chiavi, le quali sono appunto generate casualmente formando alberi che saranno in generale bilanciati.

Tutte e tre le strutture dati avranno quindi costo di ricerca/inserimento logaritmico $O(\log(n))$ con n dimensione del relativo albero.

A sostegno di questa tesi è utile analizzare il grafico delle tre curve:



È interessante notare come perfino l'albero BST sia più efficiente di quello AVL nel caso medio. Come visto in precedenza questo è conseguenza del fatto che l'algoritmo di inserimento dell'AVL può effettuare più cammini fino alla radice, mentre la procedura di inserimento di una chiave nel BST non fa altro che inserire una nuova foglia in un albero semi-bilanciato.

Descrizione dell'algoritmo per il calcolo della varianza

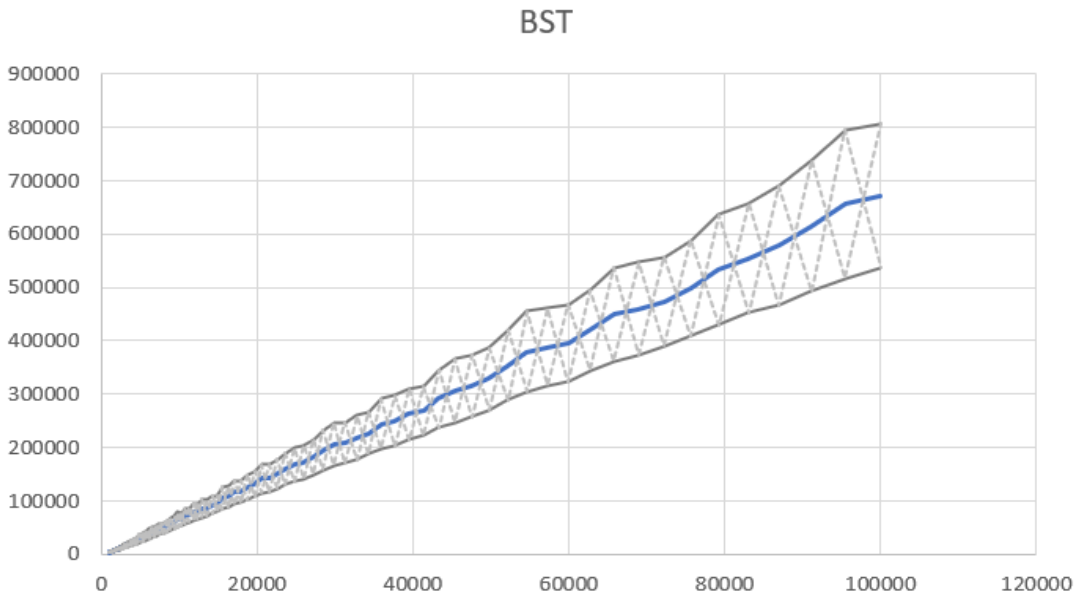
Per il calcolo della varianza è stata riservata una matrice per contenere 10 tempi di risposta per ciascun set di operazioni. La misura per ognuno di questi set è stata quindi rieseguita per 10 volte. Una volta ottenute le 10 misure si procede calcolando lo scarto quadratico medio mediante la seguente formula:

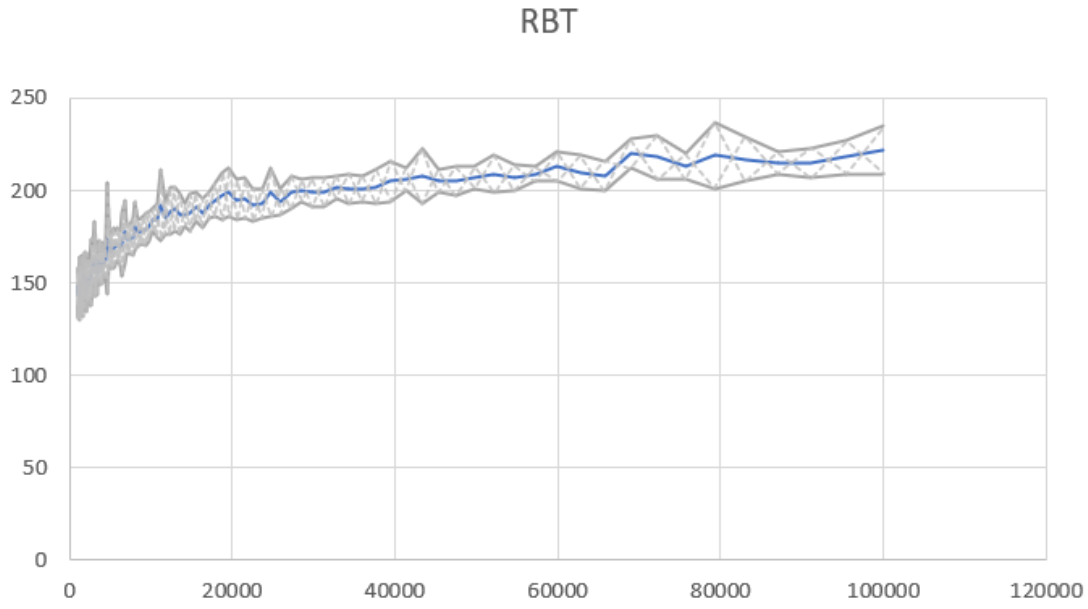
$$\sigma = \sqrt{\frac{1}{(N-1)} \sum_{i=1}^{N} (x_i - \bar{x})^2}$$

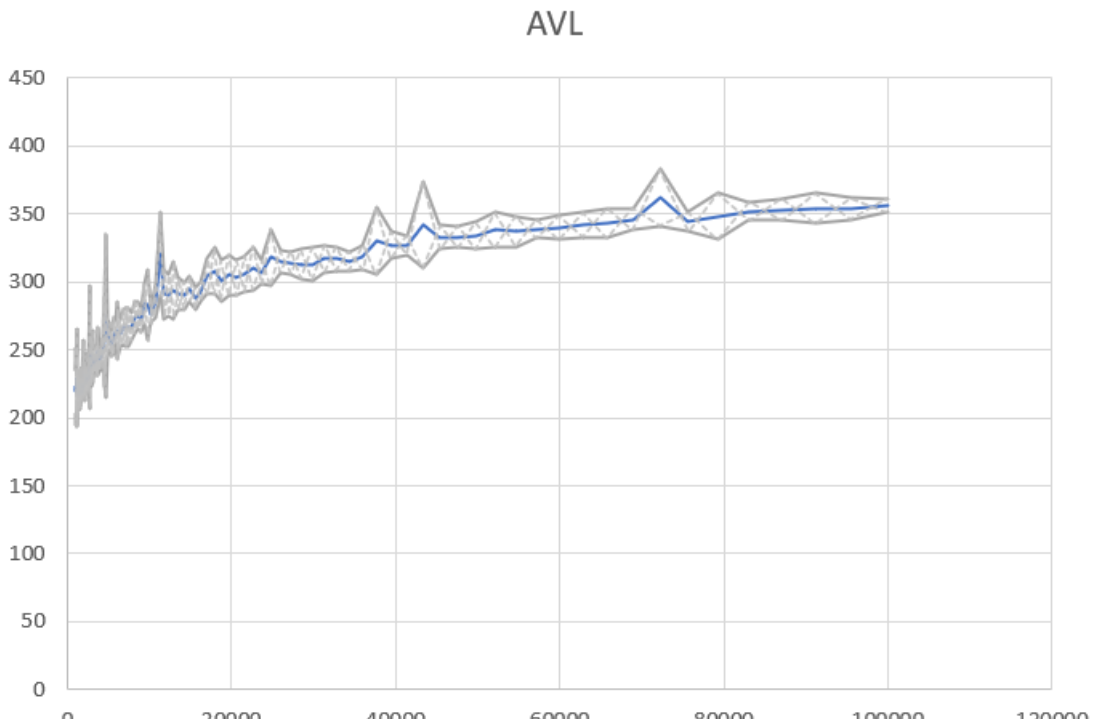
per ognuna delle 10 misure di ogni set si sottrae il tempo medio e si eleva tutto al quadrato. Si sommano tutti i 10 valori ottenuti e si divide per 9, infine tutto sotto radice. Per memorizzare il valore dello scarto quadratico medio per ogni stringa è stato utilizzato un array di long long.

Le dimensioni dei 100 set, i tre tempi medi e i relativi scarti quadratici sono stati salvati in un file di testo.

È stata considerata solo la varianza degli alberi nel caso pessimo in quanto nel caso medio sono stati ottenuti dei risultati poco indicativi.


BST
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
20000
40000
60000
80000
100000
120000


RBT
250
200
150
100
50
0
0
20000
40000
60000
80000
100000
120000


AVL
450
400
350
300
250
200
150
100
50
0

Dai grafici ottenuti viene confermata la stretta somiglianza tra gli alberi AVL e RBT, entrambi con tempo medio e varianza logaritmici, quest'ultima a distanza breve e costante dalla linea dei tempi medi (in blu).

La varianza relativa al BST invece aumenta in modo costante. La retta descritta dagli scarti quadratici medi, però, ha una maggiore pendenza rispetto a quella della linea dei tempi medi. All'aumentare del numero di operazioni quindi i tempi di esecuzione potranno divergere sempre di più dal tempo medio.