Analisi dei tempi di esecuzione di operazioni di inserimento e ricerca in Alberi Binari

Andrea Cantarutti (141808)
Francesco Bombassei De Bona (144665)

5 maggio 2021

Indice

1 Introduzione **3**

2 Alberi binari di ricerca **4**
2.1 Binary Search Tree 4
2.2 Adelson-Velsky and Landis Tree 4
2.3 Red-Black Tree 5

3 Raccolta dei Dati **5**
3.1 Ottenimento della risoluzione del Clock 5
3.2 Warmup della Java Virtual Machine 6
3.3 Programmazione funzionale 6
3.4 Modellazione del foglio Excel 7
3.5 Ottenimento dei tempi di esecuzione 7
3.5.1 Generazione dell'input 7
3.5.2 Popolamento dell'input 7
3.5.3 Stopwatch monotonico 7
3.5.4 Esecuzione e calcolo del tempo ammortizzato 7

4 Aspettative Teoriche **8**

5 Analisi dei tempi di esecuzione **8**
5.1 Variazione del tempo di esecuzione in scala lineare 9
5.2 Variazione del tempo di esecuzione in scala doppiamente logaritmica 10

6 Conclusioni **11**

1 Introduzione

Il presente elaborato propone l'analisi dei tempi di esecuzione di operazioni di inserimento e ricerca effettuate su alberi binari di tipo Binary Search, Adelson-Velsky and Landis, Red Black. Vengono, di seguito, discussi gli aspetti implementativi adottati e i risultati ottenuti in relazione alle aspettative teoriche.

2 Alberi binari di ricerca

Si definisce **albero binario** una struttura dati di tipo gerarchico organizzata in nodi aventi al più due successori ed un unico predecessore. In particolare, gli **alberi binari di ricerca** costituiscono una tipologia di alberi binari in cui, per ogni nodo x appartenente ad un albero T, si verificano le seguenti proprietà:

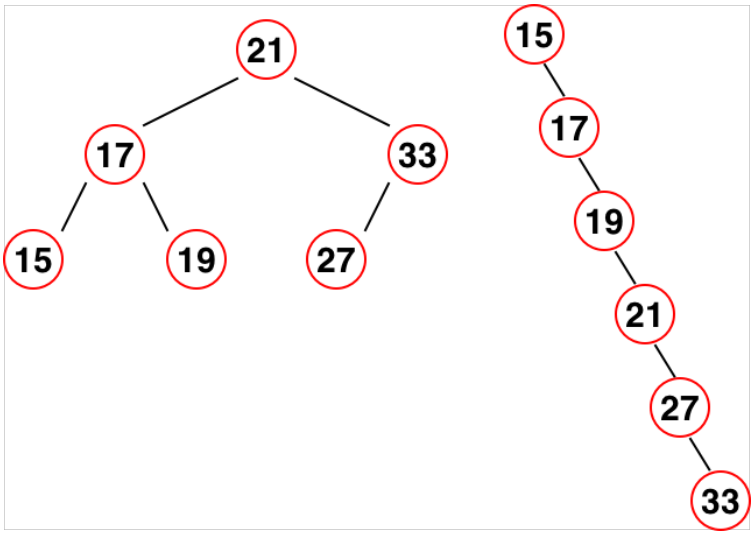
- Se y è un nodo appartenente al **sotto-albero sinistro** di x, allora $y.key < x.key$
- Se y è un nodo appartenente al **sotto-albero destro** di x, allora $y.key > x.key$

Ad integrare tale definizione sono i concetti di altezza e bilanciamento, applicabili ad ogni nodo $x \, \epsilon \, T$. La prima, fornisce un parametro atto a descrivere la distanza massima fra il nodo radice e i nodi terminali di un albero. Il secondo, invece, descrive la differenza fra le altezze dei due sottoalberi destro e sinistro di un nodo radice.

2.1 Binary Search Tree

I Binary Search Tree (BST) prevedono un'implementazione degli alberi binari di ricerca basata unicamente sulla loro definizione teorica. Pertanto, ogni inserimento si basa sul confronto della chiave del nuovo nodo con le chiavi dei nodi già presenti all'interno della struttura dati, a partire dal nodo radice.

La principale limitazione dei BST è dovuta al fatto che l'inserimento di una sequenza di nodi ordinati in base alla loro chiave comporta la formazione di un albero completamente degenere.



Un generico BST (sinistra) e uno completamente degenere (destra)

Di conseguenza, le operazioni prese in esame prevedono il seguente costo asintotico:

Operazione	**Caso Peggiore**	**Caso Medio**
Ricerca	$\Theta(n)$	$O(\log n)$
Inserimento	$\Theta(n)$	$O(\log n)$

2.2 Adelson-Velsky and Landis Tree

Gli alberi binari di tipo **AVL** definiscono un'implementazione degli alberi binari di ricerca che sfrutta i concetti di *bilanciamento*, *altezza di un nodo*, *rotazione sinistra* e *rotazione destra* al fine di garantire un forte bilanciamento tra il sotto-albero sinistro e destro di ogni nodo x appartenente all'albero T. In particolare, Le procedure di rotazione vengono applicate al fine di ri-bilanciare un albero o sotto-albero le cui altezze dei nodi figli differiscono di ± 1. Definito, infatti, il bilanciamento come funzione $b(x)$, si osserva come:

$$\forall x \, \epsilon \, T \; (b(x) = h(x.left) - h(x.right) \quad \wedge \quad |b(x)| \leqslant 1).$$

L'introduzione di procedure di verifica ed eventuale correzione del bilanciamento rendono impossibile la formazione di alberi binari sbilanciati e/o degeneri, riducendo conseguentemente il costo delle procedure di inserimento e ricerca nel caso peggiore. In particolare, la struttura dati si presta ad impieghi che coinvolgono un maggior numero di operazioni di ricerca, in quanto meno costose dal punto di vista computazionale.

Si prevede il seguente costo asintotico per le operazioni su **AVL Tree**:

Operazione	**Caso Peggiore**	**Caso Medio**
Ricerca	$O(\log n)$	$O(\log n)$
Inserimento	$O(\log n)$	$O(\log n)$

2.3 Red-Black Tree

I **Red-Black Tree** costituiscono un'ulteriore implementazione di alberi binari bilanciati, basata sull'introduzione della proprietà `color` per ogni nodo (che può assumere i valori `red` o `black`) e delle le seguenti proprietà:

- Le foglie `NIL` hanno colore nero
- Ogni nodo con proprietà `color = red` ha due figli con proprietà `color = black`
- Per ogni nodo x lungo ogni cammino verso un nodo foglia, si contano lo stesso numero di nodi con proprietà `color = black`
- Il nodo radice ha, inizialmente, proprietà `color = black`.

Al fine di mantenere le proprietà descritte, in seguito all'inserimento di un nodo vengono applicate eventuali rotazioni che permettono all'albero di mantenersi bilanciato. La presenza di una distinzione tra **nodi rossi** e **neri** permette inoltre che:

$$\forall x \, \epsilon \, T \; |b(x)| \leqslant 2.$$

Di conseguenza, i Red-Black Tree si prestano ad impieghi che coinvolgono un maggior numero di operazioni di inserimento. Si prevede il seguente costo asintotico:

Operazione	**Caso Peggiore**	**Caso Medio**
Ricerca	$O(\log n)$	$O(\log n)$
Inserimento	$O(\log n)$	$O(\log n)$

3 Raccolta dei Dati

Di seguito vengono descritte le strategie adottate al fine di ottenere tempi di esecuzione relativi ad input casuali di diverse dimensionalità con errore relativo complessivo $\varepsilon \leqslant 1\%$.

3.1 Ottenimento della risoluzione del Clock

Al fine di ottenere misurazioni precise e la minimizzazione dell'errore relativo complessivo, è stata tenuta in considerazione la risoluzione del clock durante l'esecuzione delle operazioni di ricerca ed inserimento sugli alberi binari. L'ottenimento di quest'ultima è stato delegato al metodo `getResolution()`, contenuto all'interno della classe `Resolution`.

Per duecento iterazioni, vengono richiesti un valore temporale iniziale e finale fintanto che questi non differiscono. Soddisfatta la condizione, la differenza fra il tempo iniziale e quello finale viene inserita in una struttura dati `Vector<Long>`. Al termine delle iterazioni, viene restituita la mediana del vettore.

L'implementazione della funzione viene illustrata di seguito.

```java
public static Long getResolution(){
    long start, end, res;
    Vector<Long> resolutions = new Vector<>();
    for(int i=1; i<=200; i++) {
        start = System.nanoTime();
        do {
            end = System.nanoTime();
        } while (start == end);
        res = end - start;
        resolutions.add(res);
    }

    Collections.sort(resolutions);
    return resolutions.get(100);
}
```

Il valore ottenuto viene successivamente moltiplicato per 101 al fine di ottenere un parametro **maxError** atto a garantire un errore relativo inferiore o pari all'1% durante il calcolo dei tempi di esecuzione.

Supponendo, infatti, il verificarsi della seguente condizione (dove $\hat{x}$ indica il tempo di esecuzione misurato, x il tempo di esecuzione effettivo ed R la risoluzione del clock):

$$\hat{x} \, \epsilon \, [x - R, x + R]$$

È possibile indicare l'errore relativo come di seguito:

$$\varepsilon \, = \, \frac{\hat{x} - x}{x} \implies \varepsilon \, \epsilon \, [\frac{\hat{x} - R - x}{x}, \frac{\hat{x} + R - x}{x}] \implies \varepsilon \, \epsilon \, [-\frac{R}{x}, \frac{R}{x}]$$

Di conseguenza, al fine di ottenere un errore relativo ε inferiore o pari all'1% è necessario che si verifichi la seguente condizione:

$$\frac{R}{x} \leqslant 0.01$$

Il successivo sviluppo algebrico comporta la seguente disequazione:

$$x \geqslant \frac{R}{0.01} \implies \hat{x} \geqslant \frac{R}{0.01} + R \implies \hat{x} \geqslant R \cdot (\frac{1}{0.01} + 1) \implies \hat{x} \geqslant 101R$$

3.2 Warmup della Java Virtual Machine

La JVM sfrutta il meccanismo di lazy loading delle classi al fine di ottimizzare lo spazio di memoria richiesto durante il runtime: le classi vengono caricate dinamicamente in base alla porzione di codice che viene utilizzata. Tuttavia, la raccolta di dati temporali relativi all'esecuzione degli algoritmi d'interesse richiede la riduzione del lazy loading durante le misurazioni. A tal fine, gli algoritmi vengono lanciati con input costituito da vettori generati casualmente per permettere alla JVM di caricare tutte le classi necessarie.

Nel passaggio dal meccanismo di warmup all'esecuzione della misurazione dei tempi vi sono comunque valori anomali dovuti al funzionamento a alla gestione della memoria da parte della JVM. Tuttavia, la loro effettiva influenza sulla valutazione dei tempi d'esecuzione degli algoritmi è minima.

3.3 Programmazione funzionale

Nonostante il linguaggio Java si appoggi al paradigma **Object Oriented**, al fine di evitare variazioni nel tempo di esecuzione è stato scelto di seguire un'implementazione di tipo funzionale per le tre tipologie di alberi binari. Le operazioni di inserimento, ricerca e di eventuale ribilanciamento sono state implementate come metodi statici che restituiscono, in output, il puntatatore al nodo radice o, eventualmente, al nodo manipolato all'interno della struttura dati.

3.4 Modellazione del foglio Excel

La comunicazione tra il codice e il file `Time.xlsx` preposto al raccoglimento dei dati avviene tramite la libreria **Apache POI**, che permette una completa modellazione del foglio attraverso le interfacce:

- `Workbook` (per la modellazione completa del file)
- `Sheet` (per la gestione di molteplici fogli)
- `Row` (per la manipolazione delle righe di ogni foglio)
- `Cell` (per la manipolazione di ogni singola cella).

Il file viene dapprima introdotto tramite un oggetto di tipo `FileInputStream` e, successivamente, utilizzato come destinazione per l'inizializzazione di un Workbook. Per ogni dimensione di input viene predisposto un nuovo foglio, nel quale vengono raccolti i tempi di esecuzione dei tre algoritmi calcolati sulla base degli indici preposti. Al termine dell'esecuzione il foglio viene salvato, reso accessibile e i dati calcolati possono essere visualizzati e analizzati tramite le funzionalità offerte dal software, applicando le opportune statistiche riassuntive.

3.5 Ottenimento dei tempi di esecuzione

Al fine di ottenere i dati necessari, è stata predisposta, per ognuno dei tre alberi presi in esame, la misurazione di cinquanta diverse esecuzioni per ogni dimensione dell'input.

3.5.1 Generazione dell'input

Al fine di valutare il tempo di esecuzione in base alla variabilità della dimensione degli alberi, sono stati generati input la cui cardinalità cresce esponenzialmente nel corso dell'esecuzione. Ad ogni iterazione, la dimensione dell'input viene calcolata tramite la funzione:

$$size = 1.116^n \cdot 100.$$

Dove *size* indica la dimensione dell'input ed n l'indice dell'attuale iterazione. In particolare, alla prima ripetizione si avranno input di dimensione $size = 100$, mentre all'ultima ($n = 100$) si avranno input di dimensione $size = 5233054$.

3.5.2 Popolamento dell'input

Al fine di ottenere un'aleatorietà sufficiente alle finalità dell'analisi, gli input sono stati popolati da valori interi generati pseudo-casualmente dall'algoritmo lineare congruenziale *Mersenne Twister*, sviluppato nel 1997 da Makoto Matsumoto e Takuji Nishimura.

L'immenso periodo, che corrisponde al valore $2^{19937} - 1$ (**Numero primo di Mersenne**), combinato alla notevole efficienza, rende l'algoritmo particolarmente veloce nella generazione dei numeri pur garantendone l'equi-distribuzione e l'uniformità nei risultati.

3.5.3 Stopwatch monotonico

I tempi di esecuzione e la risoluzione del clock sono stati ottenuti tramite lo stopwatch monotonico `public static long nanoTime()` contenuto all'interno della classe `System` dello **Standard Java Development Kit**. Quest'ultima, appoggiandosi alla sorgente di tempo ad alta risoluzione della Java Virtual Machine, restituisce una misurazione di tempo attuale in **nanosecondi**.

3.5.4 Esecuzione e calcolo del tempo ammortizzato

L'esecuzione del programma, delegata alla classe `Time.java`, prevede, per ogni albero, la seguente procedura di esecuzione:

- Viene generato un insieme casuale di n chiavi, dove n corrisponde alla dimensionalità descritta dalla funzione esponenziale presentata in precedenza
- In ogni albero, per ogni chiave `k` generata viene eseguita:
  - La ricerca di un nodo con chiave analoga all'interno dell'albero
  - L'inserzione di un nuovo nodo con chiave `k` nel caso in cui la ricerca restituisca un risultato nullo
- L'esecuzione viene ripetuta fintanto che il limite imposto dalla costante `maxError` (computata all'inizio del programma) risulta rispettato. Questa, infatti, definisce il tempo minimo di esecuzione da garantire affinché l'errore relativo non superi il limite ε

- Il tempo ottenuto a seguito di una singola esecuzione viene, infine, diviso per il numero di operazioni di ricerca ed eventuale inserimento effettuate, al fine di ottenere il **tempo ammortizzato al variare di** n
- La procedura viene ripetuta per ogni input e le misurazioni ottenute vengono raccolte nel file `Time.xlsx`, successivamente usato per il calcolo di opportune statistiche riassuntive

In quanto parte dell'effettiva procedura di inserzione di una chiave, il tempo di esecuzione ottenuto tiene conto del tempo di creazione di ogni nodo a partire dalla chiave casuale. Tuttavia, il tempo di inizializzazione degli alberi è stato escluso dalla misurazione. Lo stopwatch acquisisce il primo timestamp, infatti, solo successivamente alla costruzione dell'albero.

4 Aspettative Teoriche

Sulla base delle caratteristiche implementative discusse in precedenza, sono state formulate aspettative in relazione ai concetti noti dal punto di vista teorico. Si tiene presente, in particolare, come l'input di chiavi prodotte da un algoritmo come **Mersenne Twister**, a causa della sua equi-distribuzione, porti alla generazione di un vettore che tende al bilanciamento.

Di conseguenza, l'aspettativa è che i risultati ottenuti illustrino un andamento che rispecchia, per ogni tipo di albero, un complessità nel caso medio appartenente a $O(\log n)$.

5 Analisi dei tempi di esecuzione

Il workbook `Time.xlsx` presenta un foglio di calcolo per ognuno degli input utilizzati. In particolare, il nome di ogni foglio corrisponde alla cardinalità dell'input utilizzato. Per ogni gruppo di cinquanta distinte esecuzioni sono state calcolate **mediana**, **varianza** e **deviazione standard**. Infine, i dati sono stati riassunti in un ulteriore foglio di calcolo che, per ogni riga, presenta:

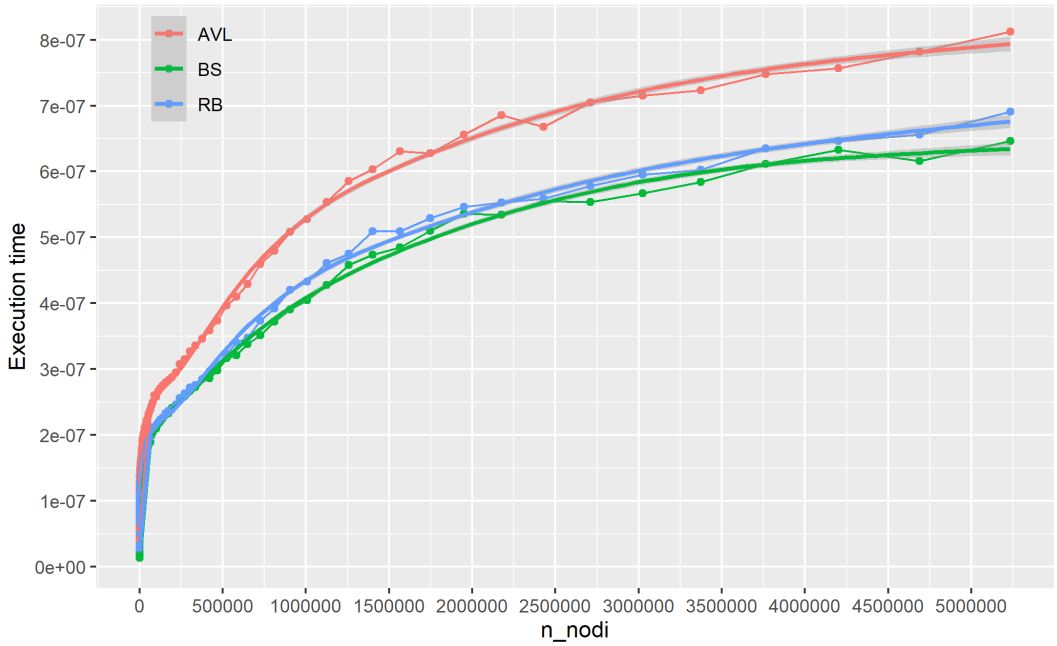
Cardinalità dell'input	**Mediana BST**	**Mediana AVL**	**Mediana RBT**

Lo script `R/plots.Rmd` effettua nell'ordine:

- La lettura di `Time.xlsx` con conseguente conversione del foglio di calcolo riassuntivo in un omonimo file `.csv`
- La produzione di un grafico che illustra la variazione del tempo di esecuzione in rapporto alla crescita della dimensionalità dell'input
- La produzione di un grafico analogo al precedente con ascisse e ordinate espresse in scala *doppiamente logaritmica*.

Di seguito vengono presentati i risultati ottenuti.

5.1 Variazione del tempo di esecuzione in scala lineare



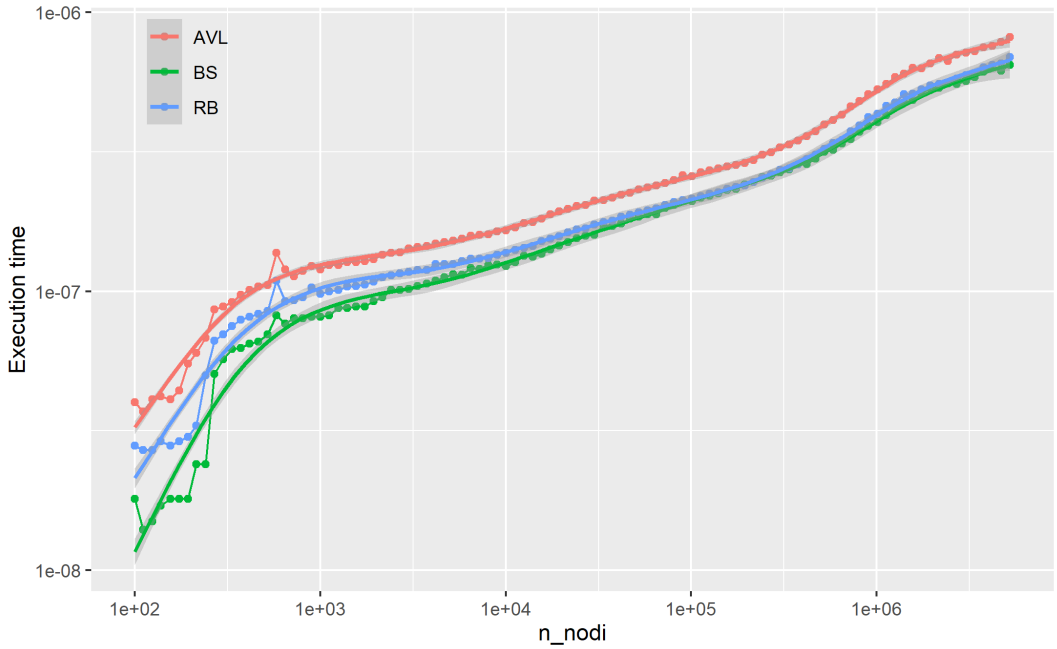
Il grafico prodotto porta ad un immediato riscontro con le aspettative teoriche presentate al punto 4: tutti gli algoritmi presentano una crescita di tipo **logaritmico** del tempo di esecuzione all'aumentare della cardinalità dell'input. Questo si dimostra coerente con la complessità asintotica dei tempi di ricerca ed inserimento che, per ognuno dei tre alberi, è $O(\log n)$ nel **caso medio**.

Si osserva, tuttavia, come gli alberi che hanno dimostrato un comportamento più efficiente siano stati i **Binary Search Tree**. Il comportamento dei Red-Black Tree è stato simile a quello dei precedenti, anche se con un costo leggermente maggiore. I meno efficienti sono stati gli AVL Tree, il cui tracciato sul grafico raggiunge i tempi di esecuzione più elevati.

Come riferito in precedenza, è importante notare come, grazie alla sua equi-distribuzione e uniformità, l'input casuale generato tramite Mersenne Twister generi un albero binario **tendente al bilanciamento**. La maggiore efficienza dei BST è quindi dovuta alla mancanza di procedure di controllo e di ribilanciamento che, considerando input preventivamente bilanciati, permette all'algoritmo di inserzione di essere eseguito in meno tempo rispetto agli altri alberi. Per le stesse motivazioni è bene, tuttavia, tenere presente come, nel caso dei Binary Search Tree, ad un minimo sbilanciamento dell'input (e quindi dell'albero) sia ragionevole aspettarsi un notevole incremento dei tempi di esecuzione.

La differenza osservata fra **AVL Tree** e **Red-Black Tree** è, invece, attribuibile alle differenze che i due alberi presentano in relazione ad operazioni di ricerca e di inserzione. La procedura di esecuzione prevede la ricerca di n nodi all'interno dell'albero e l'inserimento degli eventuali m nodi non individuati dalla precedente operazione di ricerca. Considerando l'ampio dominio nel quale vengono generate le chiavi casuali, è ragionevole considerare situazioni nelle quali $n = m$. Si osserva come l'inserzione nei **RBT** sia particolarmente vantaggiosa (come descritto al punto 2.3) grazie alla proprietà che permette uno sbilanciamento di al più due unità. Considerando input già bilanciati, la procedura di `Fix` effettuata nell'implementazione dei Red Black Tree risulta decisamente più performante di quella richiesta negli AVL Tree, la quale richiede che $|b(n)| \leqslant 1$. Pertanto, quest'ultimi si dimostrano meno vantaggiosi dal punto di vista computazionale nel caso medio.

5.2 Variazione del tempo di esecuzione in scala doppiamente logaritmica



Producendo lo stesso grafico in scala doppiamente logaritmica, si ottiene una conferma delle osservazioni effettuate in precedenza. I tre tipi di albero, infatti, presentano lo stesso costo asintotico e, in particolare, i **Binary Search Tree** e i **Red Black Tree** manifestano un comportamento molto simile (sopratutto in input di elevata dimensionalità) e più efficiente rispetto a quello degli **AVL Tree** nel caso medio.

Il grafico, tuttavia, non rispetta compeltamente le aspettative: computando una funzione con crescita di carattere logaritmico su un piano cartesiano con ascisse e ordinate espresse in scala anch'essa logaritmica, si suppone di poter osservare un andamento lineare. La curvatura individuata in riferimento ad input di cardinalità compresa fra 100 e 1000 non risponde alle caratteristiche attese. Osservando, però, l'andamento della restante parte del grafico, si è concluso che tale anomalia possa essere attribuibile ad un insieme di problematiche:

- Lazy loading della Java Virtual Machine
- Limitazioni dello stopwatch utilizzato con input di dimensionalità ridotta
- Altre problematiche relative all'instabilità dell'ambiente di esecuzione del codice (variazioni nelle prestazioni dovute ad interrupt di sistema, modifiche della priorità di scheduling dei processi).

6 Conclusioni

In seguito all'analisi presentata è stato possibile individuare le caratteristiche che contraddistinguono i tre diversi tipi di alberi binari implementati, osservando un riscontro pratico della loro applicazione. Ogni implementazione di un albero binario di ricerca presenta degli aspetti maggiormente efficienti rispetto alle altre ed è pertanto indispensabile selezionare accurattamente la struttura dati più adatta a soddisfare obiettivi specifici.

Si osserva, infine, come, per input generati in forma pseudo-casuale, gli alberi che presentano maggiore efficienza nelle operazioni di ricerca e inserimento siano i **Binary Search Tree**.