Alberi Rosso-Neri vs Alberi Binari di Ricerca

Laboratorio di Algoritmi - Esercizio A

Samuel Bruno

10 Ottobre 2022

Abstract

L'obiettivo di questa relazione è quello di analizzare le differenze tra Alberi Binari di Ricerca e Alberi Rosso-Neri in modo da ottenere un confronto diretto atto a comprendere meglio la disuguaglianza di efficienza delle due implementazioni.

1 Introduzione

Un **Albero Binario di Ricerca** è un albero binario che soddisfa la seguente proprietà:

i.) Se y è nel sottoalbero sinistro di x, allora, $y.key \leq x.key$

ii.) Se y è nel sottoalbero destro di x, allora, $y.key \geq x.key$

Le operazioni di base sono effettuate in un tempo $O(h)$ (dove h = altezza albero).

Un **Albero Rosso-Nero**, invece, è un tipo di Albero Binario di Ricerca bilanciato in cui ad ogni nodo associamo un colore, che può essere rosso o nero. Inoltre deve soddisfare le seguenti proprietà:

i.) Ogni nodo è **rosso** o **nero**

ii.) La radice è **nera**

iii.) Ogni foglia è **nera**

iv.) Se un nodo è **rosso**, allora entrambi i suoi figli sono **neri**

v.) Tutti i cammini da ogni nodo alle foglie contengono lo stesso numero di nodi **neri**

Il confronto verrà effettuato proprio sulle operazioni di base che entrambi gli alberi hanno in comune ed in particolare verrà analizzato attraverso i tempi di esecuzione degli algoritmi. Date le basi, possiamo procedere a descrivere la struttura dati di entrambe.

2 Strutture dati degli algoritmi

Sia gli Alberi Binari di Ricerca che gli Alberi Rosso-Neri presentano le seguenti operazioni fondamentali:

i.) **Tree-Insert(T, z):** Dato un albero T e un nuovo nodo z, inserisce, confrontando opportunamente le chiavi, il nodo z come figlio di un altro nodo all'interno dell'albero.

ii.) **Tree-Search(x, k):** Dato un nodo x e una chiave k, la ricerca parte dal nodo x restituendo infine il nodo la cui chiave è uguale alla chiave k.

iii.) **Tree-Successor(x):** Dato un nodo x, restituisce il più piccolo nodo maggiore del nodo x.

iv.) **Tree-Predecessor(x):** Dato un nodo x, restituisce il più grande nodo minore del nodo x.

v.) **Get-Root(T):** Dato un albero T restituisce la sua radice.

Gli Alberi Rosso-Neri hanno però tre operazioni fondamentali in aggiunta a quelle elencate sopra, che vengono utilizzate come operazioni ausiliarie dell'inserimento; queste procedure, infatti, eliminano le violazioni dovute all'inserimento di una chiave nell'albero così da farne rispettare le proprietà:

i.) **RB-Insert-Fixup(T, z):** Dato un albero T e un nodo z, corregge l'inserimento del nodo z attraverso delle *rotazioni* dei nodi a destra o sinistra. Infine, qualora il colore della radice fosse diventata rossa, viene ripristinata al colore nero.

ii.) **Left-Rotate(T, z):** Dato un albero T e un nodo z, effettua una rotazione sinistra, ovvero il nodo y, figlio destro di z, diventa la nuova radice del sottoalbero e il suo figlio sinistro sarà z; inoltre il precedente figlio sinistro di y diventerà ora il figlio destro di z.

iii.) **Right-Rotate(T, z):** Operazione del tutto speculare alla Left-Rotate(T, z).

In particolare, RB-Insert-Fixup effettua delle correzioni in tre casi: il ***primo*** nel caso in cui lo zio di z è **rosso** viene risolto colorando il padre di **nero**, il nonno di **rosso** e lo zio di **nero** (Figura 1);

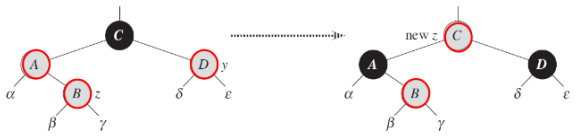

Figure 1: Caso zio di z è rosso

il ***secondo*** nel caso in cui lo zio di z è **nero** e z è un figlio *destro*, allora viene effettuata una rotazione a sinistra tra z e il padre (Figura 2);

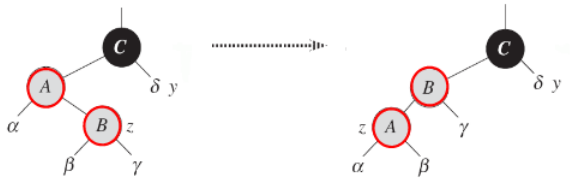

Figure 2: Caso zio di z è nero e z è un figlio destro

il ***terzo*** nel caso in cui lo zio di z è **nero** e z è un figlio *sinistro*, allora viene effettuata una rotazione a destra tra il padre di z e il nonno che vengono anche colorati rispettivamente di **nero** e di **rosso** (Figura 3).

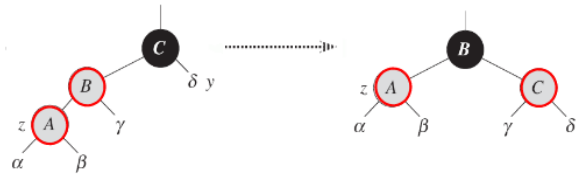

Figure 3: Caso zio di z è nero e z è un figlio sinistro

Per concludere, in questa relazione non viene considerata l'operazione di cancellazione di un nodo.

3 Prestazioni attese

3.1 Alberi Binari di Ricerca

Come già introdotto nella prima sezione, gli Alberi Binari di Ricerca effettuano le loro operazioni di base in un tempo di $O(h)$, dove h=altezza dell'albero. Più nello specifico, prendendo come riferimento la ricerca e l'inserimento, che sono le due operazioni più dispendiose in termini di memoria, si hanno come prestazioni attese le seguenti complessità:

Operazione	Peggiore	Medio	Migliore
Inserimento	$O(n)$	$O(\log n)$	$O(1)$
Ricerca	$O(n)$	$O(\log n)$	$O(1)$

Table 1: Complessità degli Alberi Binari di Ricerca

Questa differenza di complessità dipende dall'ordine di inserimento dei nodi. Il caso peggiore si ha quando un albero è completamente sbilanciato (tutti i nodi a destra o sinistra) e la sua altezza è pari al numero di nodi (n) meno 1. Il caso medio quando un albero è bilanciato e gli elementi sono distribuiti uniformemente fra i sottoalberi. Infine, il caso migliore è banale e semplicistico in quanto si limita per la ricerca di trovare la radice e per l'inserimento di inserirla.
Si può però parlare di ABR **ottimale** il cui caso peggiore di inserimento è $\theta(\log n)$ e avviene quando un albero è **perfettamente bilanciato**.

3.1.1 Alberi Binari di Ricerca Randomizzati

Per sperimentare diverse soluzioni, in questa relazione, ho voluto considerare anche l'applicazione di ABR Randomizzati, nel quale genero una permutazione randomica di elementi finiti presenti in un array che verranno poi inseriti nell'Albero Binario di Ricerca. Per effettuare questa operazione ho utilizzato l'algoritmo di **Fisher–Yates** che, in maniera molto semplice, mescola tutti gli elementi presenti in un array rendendolo di fatto completamente randomico. In termini di complessità, quindi, ci possiamo aspettare dei risultati migliori del normale ABR, considerando che rientra nella categoria del caso medio, essendo l'albero più bilanciato e con elementi maggiormente distribuiti sia nella ricerca che nell'inserimento.

3.2 Alberi Rosso-Neri

Per quanto riguarda gli Alberi Rosso-Neri possiamo affermare che tutte le operazioni fondamentali vengono effettuate con una complessità di $O(\log n)$. Questo perchè un ARN non è nient'altro che un ABR bilanciato. Come conseguenza, gli ARN non sono così distanti in termini di logica e complessità rispetto agli Alberi Binari di Ricerca Randomizzati.

4 Documentazione del progetto

Il programma è formato in totale da 5 files:

i.) **ABRTree.py:** Nel file sono presenti le classi Node e ABRTree, contenente l'implementazione dell'Albero Binario di Ricerca.

ii.) **RBTree.py:** Nel file sono contenute le classi RBNode e RBTree che sono rispettivamente il nodo rosso nero ereditato dal nodo dell'albero di ricerca e l'Albero Rosso-Nero stesso, con tutte le sue rispettive funzioni.

iii.) **TreeComparisonUnbalanced.py:** Nel file vengono comparati, entrambi con lo stesso input, un albero di ricerca completamente sbilanciato e un Albero Rosso-Nero.

iv.) **TreeComparisonRandomized.py:** Nel file vengono comparati, entrambi con lo stesso input *randomizzato*, un albero di ricerca e un Albero Rosso-Nero.

Il progetto presenta quindi la seguente struttura:

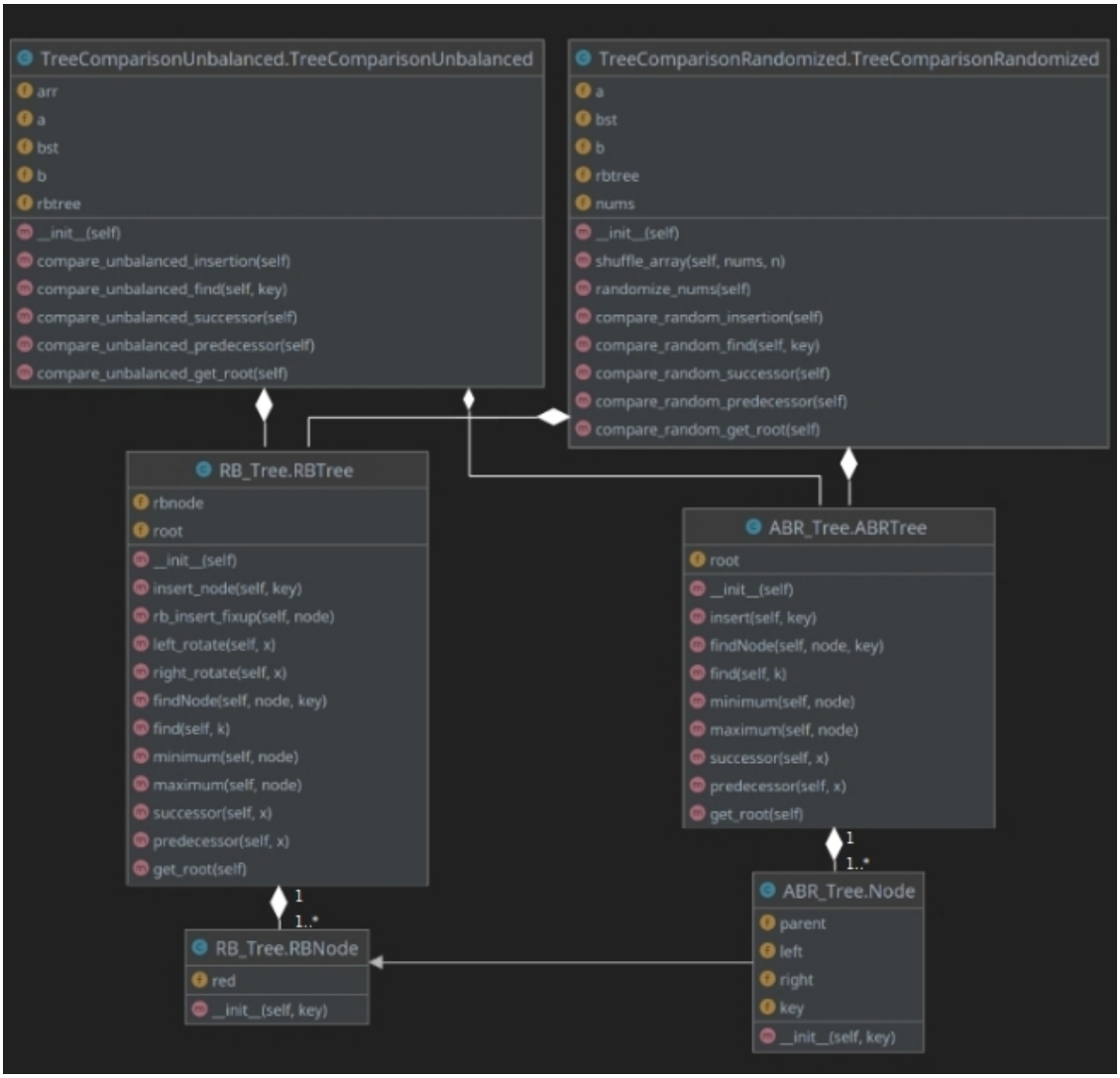

Figure 4: Diagramma delle classi

4.1 Classi e metodi

4.1.1 Classe Node

E' la classe che implementa il singolo nodo di un Albero Binario di Ricerca. I suoi attributi sono: key, parent, left, right.

4.1.2 Classe ABRTree

E' la classe che implementa l' Albero Binario di Ricerca e lo fa inserendo al suo interno un elemento di Node, che rappresenta per l'appunto un nodo. Questa classe fornisce i seguenti metodi:

- **insert:** Data una chiave, inserisce un nodo all'interno dell'albero.
- **findNode:** Ricerca una chiave confrontando ogni nodo.
- **find:** Richiama la funzione findNode a cui passa il nodo di partenza (la radice) e un elemento k.
- **minimum:** Dato un nodo di partenza, restituisce il valore minimo del suo sottoalbero.
- **maximum:** Dato un nodo di partenza, restituisce il valore massimo del suo sottoalbero.
- **successor:** Dato un nodo restituisce il suo successore. Utilizza al suo interno la funzione minimum.
- **predecessor:** Dato un nodo restituisce il suo predecessore. Utilizza al suo interno la funzione maximum.
- **get-root:** Ritorna la radice dell'albero.

4.1.3 Classe RBNode

E' la classe che implementa il singolo nodo di un Albero Rosso-Nero. **Eredita** gli attributi della classe Node e come suo attributo aggiuntivo ha color: un valore booleano che rappresenta il colore di un nodo; di default il colore di un nodo è nero.

4.1.4 Classe RBTree

E' la classe che implementa l' Albero Rosso-Nero ed il principio è lo stesso dell' ABR. I suoi metodi sono gli stessi, con l'aggiunta di **rb-insert-fixup**, **left-rotate** e **right-rotate**, dei quali è già stato spiegato il funzionamento nella *Sezione 2*.

4.1.5 Classe TreeComparisonUnbalanced

In questa classe avviene il confronto tra l' Albero Rosso-Nero ed l' Albero Binario di Ricerca con un input di numeri positivi interi inseriti sequenzialmente così da far sbilanciare quest'ultimo albero. Presenta i seguenti metodi:

- **compareInsertion:** Richiama le funzioni di insert sia dell'Albero Rosso-Nero che dell'Albero Binario di Ricerca e ne calcola i tempi di esecuzione.
- **compareFind:** Richiama le funzioni di find sia dell'Albero Rosso-Nero che dell'Albero Binario di Ricerca e ne calcola i tempi di esecuzione. Viene dato in input un valore nel main (per semplicità, il valore esiste) e una volta trovato viene salvato in due variabili per ogni albero, così da venire utilizzato successivamente per ricercare il successore e il predecessore.
- **compareSuccessor:** Sfruttando il valore trovato precedentemente con la find viene richiamata la funzione del successore sia dell'Albero Rosso-Nero che dell'Albero Binario di Ricerca e ne calcola i tempi di esecuzione.
- **comparePredecessor:** Stessa funzione descritta sopra ma utilizzando la funzione del predecessore.
- **compareGetRoot:** Richiama la funzione get-root sia dell'Albero Rosso-Nero che dell'Albero Binario di Ricerca e ne calcola i tempi di esecuzione.

4.1.6 Classe TreeComparisonRandomized

In questa classe avviene il confronto tra l' Albero Rosso-Nero ed l' Albero Binario di Ricerca, stavolta però con un input di numeri randomizzato secondo l'algoritmo di Fisher–Yates in modo da rendere l'ABR il più bilanciato possibile. I metodi sono gli stessi della classe precedente.

5 Risultati Sperimentali

Gli esperimenti sono stati effettuati su una sequenza di 10000 interi positivi privi della presenza di duplicati. I test sono stati eseguiti su un HP Pavilion DV6 le cui specifiche sono:
Processore Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz 3.10 GHz Dual-core Quad-thread
RAM 8 GB 1333 Mhz
Sistema Operativo EndeavourOS - kernel Linux

Le misurazioni effettuate riguardano il tempo di esecuzione dei processi nella CPU e i vari test sono stati eseguiti per più di cinque volte in modo tale da verificare la costanza dei risultati.
Nella seguenti tabelle elenco gli ultimi 5 test eseguiti (con i risultati espressi in μs):

UNBALANCED		ABR	ARN	Migliore
TEST N° 1	INSERT (I)	4.360.684	88.219	ARN
	FIND (F)	9,87	6,71	ARN
	SUCCESSOR (S)	2,02	2,63	ABR
	PREDECESSOR (P)	1,81	2,38	ABR
	GET_ROOT (R)	1,68	1,43	ARN
TEST N° 2	I	4.367.455	88.890	ARN
	F	10,13	6,50	ARN
	S	2,52	2,51	ARN
	P	1,73	2,40	ABR
	R	1,59	1,40	ARN
TEST N° 3	I	4.462.681	89.667	ARN
	F	9,51	6,74	ARN
	S	2,75	2,54	ARN
	P	1,86	2,48	ABR
	R	1,68	1,79	ABR
TEST N° 4	I	4.494.323	89.015	ARN
	F	9,97	6,87	ARN
	S	2,63	2,36	ARN
	P	1,87	2,44	ABR
	R	1,51	1,53	ABR
TEST N° 5	I	4.460.715	87.761	ARN
	F	10,57	6,63	ARN
	S	2,24	2,62	ABR
	P	1,77	2,45	ABR
	R	1,97	1,31	ARN

Figure 5: Test Unbalanced ABR

RANDOMIZED		ABR	ARN	Migliore
TEST N° 1	INSERT	26.294	48.934	ABR
	FIND	8,54	9,20	ABR
	SUCCESSOR	2,18	2,24	ABR
	PREDECESSOR	2,80	2,12	ARN
	GET_ROOT	1,50	1,41	ARN
TEST N° 2	I	26.322	49.663	ABR
	F	7,43	5,66	ARN
	S	3,50	2,71	ARN
	P	4,11	2,17	ARN
	R	1,46	1,40	ARN
TEST N° 3	I	27.008	49.709	ABR
	F	14,08	6,15	ARN
	S	2,42	2,32	ARN
	P	1,79	2,17	ABR
	R	1,56	1,42	ARN
TEST N° 4	I	27.281	48.597	ABR
	F	8,94	8,25	ARN
	S	2,18	2,38	ABR
	P	2,21	1,93	ARN
	R	1,45	1,43	ARN
TEST N° 5	I	28.752	49.617	ABR
	F	13,62	5,47	ARN
	S	2,85	2,28	ARN
	P	1,90	2,66	ABR
	R	1,55	1,71	ABR

Figure 6: Test Randomized ABR

5.1 Analisi dei risultati

5.1.1 Unbalanced ABR

Come possiamo vedere dalla *Figura 5* un albero sbilanciato (in questo caso-limite lo è completamente) è **molto** svantaggioso e impiega parecchio più tempo di calcolo nelle operazioni di inserimento e ricerca rispetto all'Albero Rosso-Nero. Come costante, però, in favore del **ABR**, troviamo l'operazione della ricerca del **Predecessore**, data dal fatto che l'input dell'albero sono numeri interi in ordine crescente e non avendo un sottoalbero sinistro la funzione trova come predecessore semplicemente il padre del nodo dato in input. Nelle restanti operazioni però possiamo vedere come i risultati siano, nella maggior parte dei casi, simili come tempistica e come non sempre l'Albero Rosso-Nero sia per forza più veloce.

5.1.2 Randomized ABR

Nella *Figura 6*, invece, vengono rappresentati i risultati dell'albero randomizzato. Qui al contrario di prima possiamo notare come **ogni** inserimento finisca parecchio in anticipo rispetto all' Albero Rosso-Nero, mentre per la ricerca 3 volte su 4 rimane più veloce in quest' ultimo rispetto all'albero randomizzato. Gli altri risultati sono simili alla tabella Unbalanced. In conclusione nella Randomized gli Alberi Binari di Ricerca sono decisamente più valorizzati e più presenti in percentuale rispetto a prima rendendoli quindi una valida alternativa agli Alberi Rosso-Neri.

5.2 Analisi delle complessità

Analizzando le complessità dei metodi nell'operazione di **inserimento** (*Figura 7 e Figura 8*), notiamo come nel caso dell'*ABR sbilanciato* questo tenda sempre in maniera lineare con l'avanzare degli elementi inseriti. Essendo per l'appunto il caso peggiore possibile ($O(n)$), si nota la congruenza con quando detto nella *Sezione 3*. Per quanto riguarda l'**ABR randomizzato** risulta chiaro quanto questo metodo sia migliore rispetto alla controparte sbilanciata, riuscendo ad equiparare il costo dell'**Albero Rosso-Nero**; risultando, quindi, entrambi coerenti con le prestazioni attese $O(\log(n))$. Da tenere conto che i risultati sono stati acquisiti analizzando un set di soli 300 elementi. Quando questo set aumenta numericamente, anche le prestazioni vengono accentuate, rendendo ancora più accurate e chiare le analisi dei costi computazionali, che risultano sempre coesi con quando già discusso. Tutti i tempi nei grafici sono espressi in *secondi*:

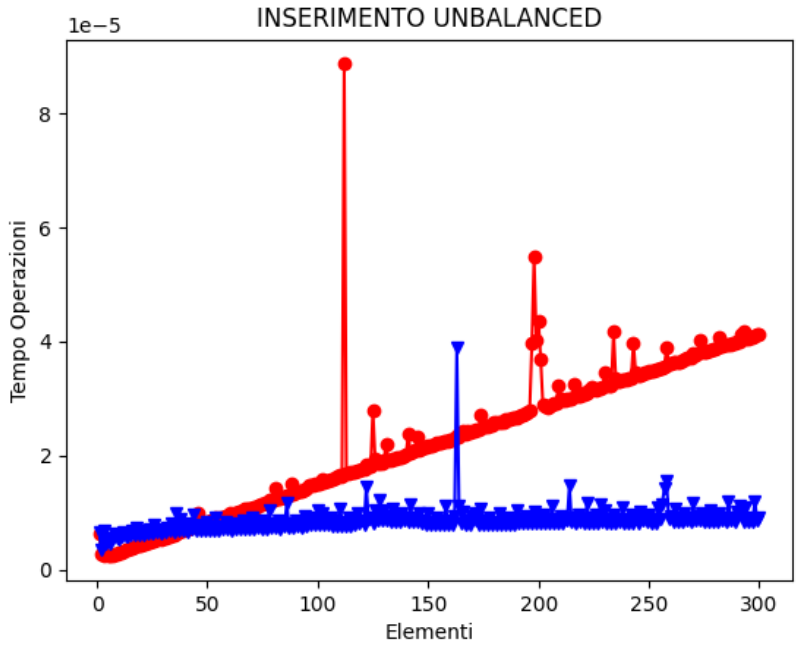

Figure 7: In **rosso** l'Abero Binario di Ricerca *Unbalanced* ed in **blu** l'Albero Rosso-Nero

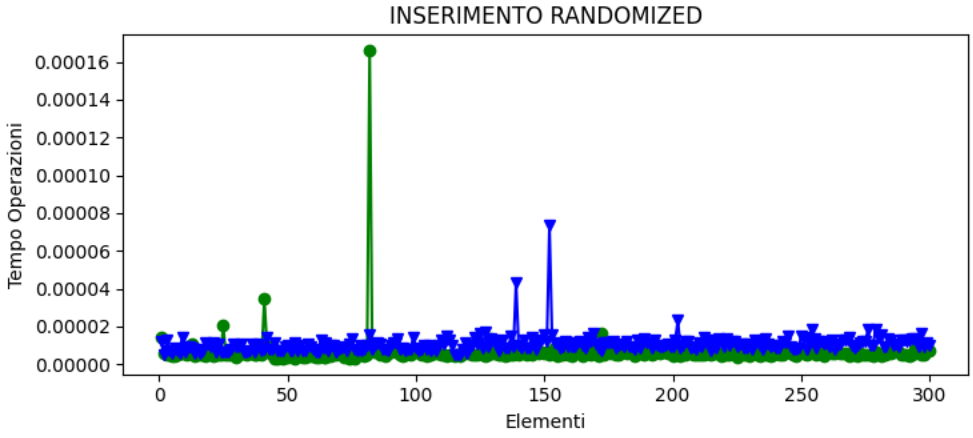

Figure 8: In **verde** l'Abero Binario di Ricerca *Randomized* ed in **blu** l'Albero Rosso-Nero

Per l'operazione di ricerca, anche qui, la situazione risulta essere conforme a quanto detto precedentemente. Nel caso dell'*ABR sbilanciato* il risultato va sempre totalmente a favore dell'Albero Rosso-Nero (*Figura 9*), mentre l'**ABR randomizzato** (*Figura 10*) è conforme al risultato dell'ARN.

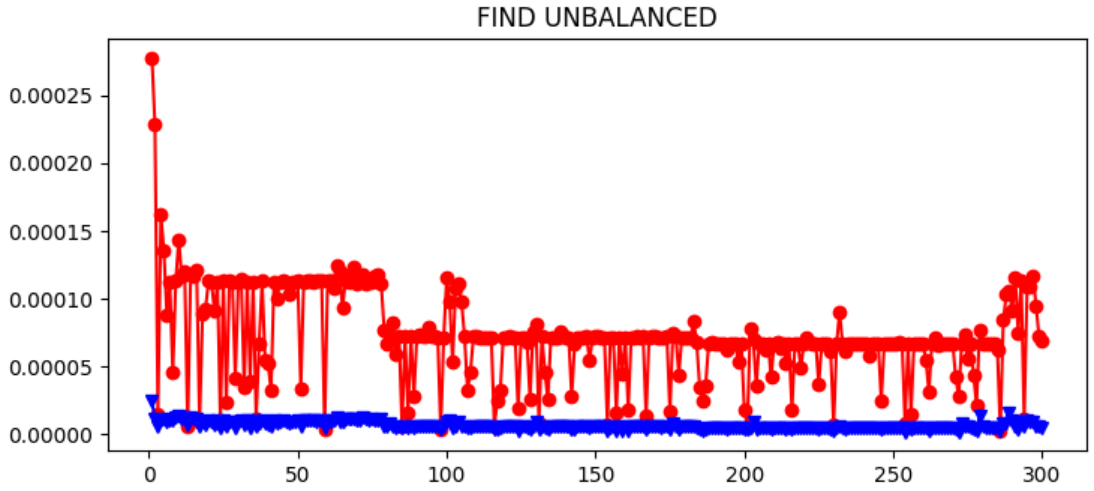

Figure 9: In **rosso** l'Abero Binario di Ricerca *Unbalanced* ed in **blu** l'Albero Rosso-Nero

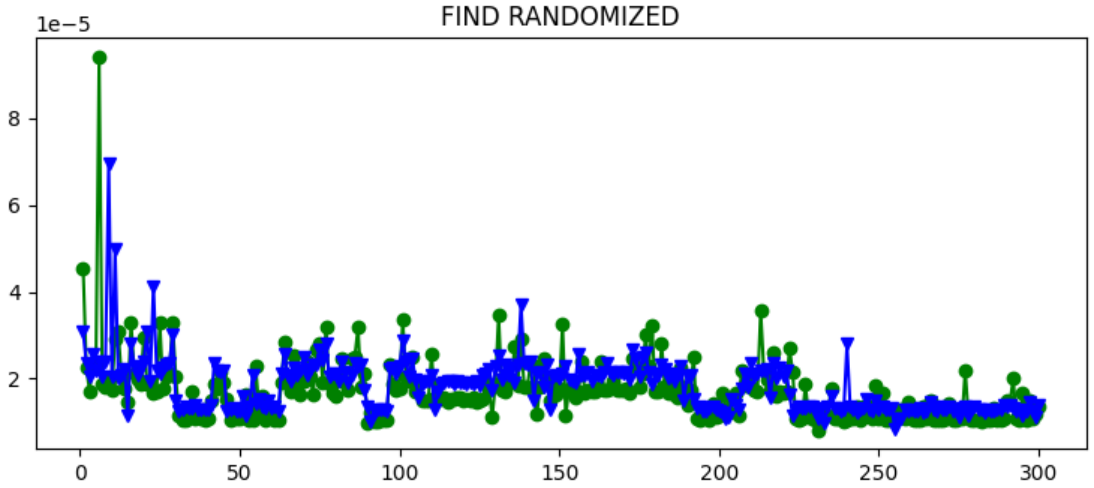

Figure 10: In **verde** l'Abero Binario di Ricerca *Randomized* ed in **blu** l'Albero Rosso-Nero

Analizzando, adesso, l'operazione *predecessor* si osserva che i risultati sono coerenti con quanto scoperto nelle tabelle di *Figura 5 e 6*. Questa, infatti, è **l'unica operazione** per cui l'**ABR sbilanciato** risulta sempre leggermente più efficiente dell'ARN, mentre l'**ABR randomizzato** si alterna con l'**ARN**, mantenendo entrambi regolari i loro costi. In ogni caso, le operazioni vengono effettuate con un costo praticamente **costante**.

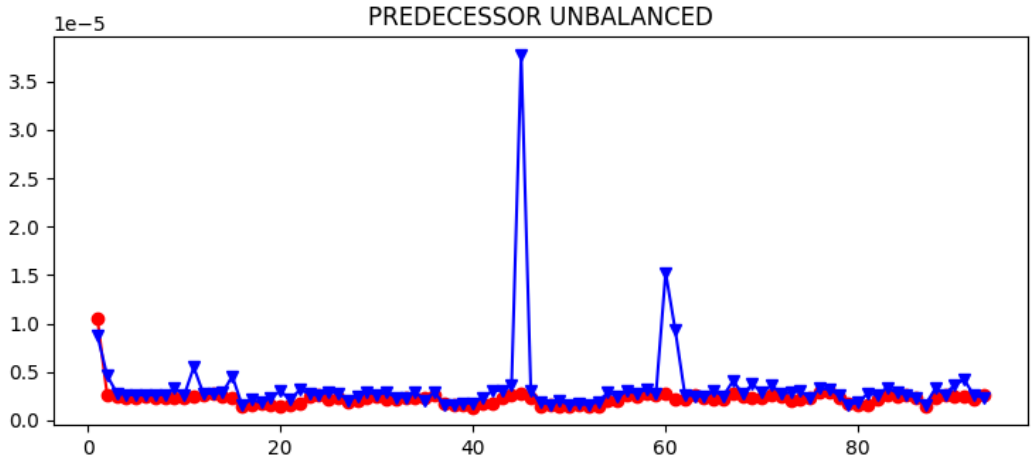

Figure 11: In **rosso** l'Abero Binario di Ricerca *Unbalanced* ed in **blu** l'Albero Rosso-Nero

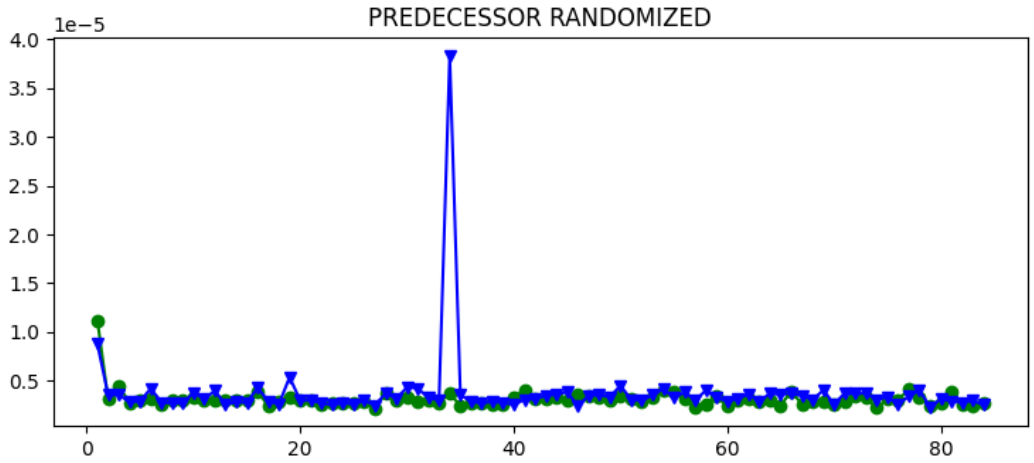

Figure 12: In **verde** l'Abero Binario di Ricerca *Randomized* ed in **blu** l'Albero Rosso-Nero

Infine, non ho incluso i grafici delle operazioni *successor* e *get-root* poichè risultano essere dei casi banali e poco interessanti.

6 Conclusione

In conclusione dagli esperimenti effettuati nella maggior parte dei casi è preferibile optare per l'uso di un **Albero Rosso-Nero** invece di uno **Binario di Ricerca**, essendo per sua natura un albero auto-bilanciato. Abbiamo però visto come in alcuni casi i risultati siano abbastanza simili (complici anche alcuni accorgimenti come la casualità dell'input) e come una struttura dati molto semplice come quella degli **ABR** possa riuscire quasi, o del tutto, ad equiparare strutture più complesse.