UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Alberi Rosso-Neri vs Alberi Binari di Ricerca

Autore:
Billi Marco

N° Matricola:
XXXXXXX

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Contents

List of Figures

1 Introduzione generale

1.1 Breve descrizione dello svolgimento degli esercizi

Per ogni esercizio suddividiamo la sua descrizione in 4 parti fondamentali:

- **Spiegazione teorica del problema** : qui è dove si descrive il problema che andremo ad affrontare in modo teorico partendo dagli assunti del libro di Algoritmi e Strutture Dati e da altre fonti.
- **Documentazione del codice** : in questa parte spieghiamo come il codice dell'esercizio viene implementato
- **Descrizione degli esperimenti condotti** : partendo dal codice ed effettuando misurazioni varie cerchiamo di verificare le ipotesi teoriche
- **Analisi dei risultati sperimentali** : dopo aver svolto i vari esperimenti riflettiamo sui vari risultati ed esponiamo una tesi

1.2 Specifiche della piattaforma di test

La piattaforma di test sarà la stessa per ogni esercizio che vedremo. Partiamo dall'hardware del computer fondamentale da conoscere per questo esercizio:

- **CPU** : Intel Core I7 8700 3.2 GHz 6 core 12 thread
- **RAM** : Crucial Ballistix 16GB DDR4 3600MHz
- **SSD** : Western Digital Green 120GB
- **Disco di memoria** : Western Digital Blu 1TB 7200RPM

Il linguaggio di programmazione utilizzato sarà Python e la piattaforma in cui il codice è stato scritto e 'girato' è l'IDE **PyCharm Edu 2021.3.1**. La stesura di questo testo è avvenuta tramite l'utilizzo dell'editor online **Overleaf**.

Part I
Alberi Rosso-Neri vs Alberi Binari di Ricerca

Esercizio A

- Vogliamo analizzare le differenze tra Alberi Binari di Ricerca e Alberi Rosso-Neri
- Per fare questo dovremo:
  - Scrivere i programmi Python che implementano l'albero RN a partire dal codice per ABR visto a lezione
    - Probabilmente l'implementazione dell'ABR vista a lezione deve essere completata con eventuali metodi o attributi mancanti
    - Per alberi RN non è obbligatorio considerare la cancellazione, ma l'inserimento non si può omettere
  - Scrivere i programmi per eseguire un insieme di test che ci permettano di comprendere vantaggi e svantaggi di ABR e RN
  - Scrivere una relazione che descriva quanto fatto

2 Spiegazione teorica del problema

2.1 Introduzione

In questo parte del progetto si descrive l'implementazione degli alberi binari di ricerca e degli alberi rosso neri e li mettiamo a confronto valutando la complessità computazionale di alcuni dei metodi che hanno logicamente in comune. Ci concentreremo, in particolare, sui metodi di inserimento e ricerca perchè sono fondamentalmente i metodi più importanti e con ogni probabilità quelli maggiormente utilizzati quando si fa riferimento a degli alberi. Non confronteremo invece il metodo di cancellazione il quale è difficilmente implementabile negli alberi rosso neri.

2.2 Aspetti fondamentali

Nell'esperimento considereremo come caso peggiore la catena lineare mentre come caso medio prenderemo la costruzione in modo casuale dell'albero (essenzialmente vuol dire che il numero che andrò ad inserire di volta in volta è casuale). Da qui in poi ci riferiremo agli alberi binari di ricerca come ABR e agli alberi rosso neri come RN. Useremo anche alcune parti fondamentali della nomenclatura della teoria della complessità computazionale come $O(\cdot)$ e $\Theta(\cdot)$. Ci riferiremo anche ad n come numero di nodi dell'albero ed ad $h = \log n$ come altezza dell'albero. Non utilizzeremo altre strutture dati particolari all'infuori degli alberi in questione. Descriviamo brevemente i metodi che andremo ad analizzare in particolare:

- Inserimento
  - ABR: iniziando dalla radice dell'albero si sceglie ricorsivamente su quale ramo spostarsi basandoci sul confronto tra la chiave della foglia in cui siamo e il valore che vogliamo inserire. Arrivati in fondo all'albero creeremo un nuovo nodo (se destro o sinistro rispetto all'ultima foglia dipende dall'implementazione. Nel nostro caso destro se il valore da inserire è maggiore del valore della foglia e sinistro viceversa). Non c'è nessun ribilanciamento dell'albero.
  - RN: fa essenzialmente la stessa cosa dell'ABR con la differenza che ad ogni inserimento l'albero si ribilacia secondo le regole degli RN. Il metodo di ribilanciamento lo appuriamo come trasparente ai fini del progetto.
- Ricerca
  - ABR: per ogni foglia prima confrontiamo il valore che stiamo cercando con quella della suddetta foglia. Se il valore è uguale allora ritorneremo un boolean di conferma.

Se raggiunta l'ultima foglia dell'albero non trovassimo il valore allora ritorneremo un boolean di fallimento. Altrimenti se non ci troviamo in nessuna delle due situazioni appena descritte scendiamo l'albero con lo stesso metodo dell'inserimento. Ovviamente per fare la ricerca partiremo dalla radice dell'albero.

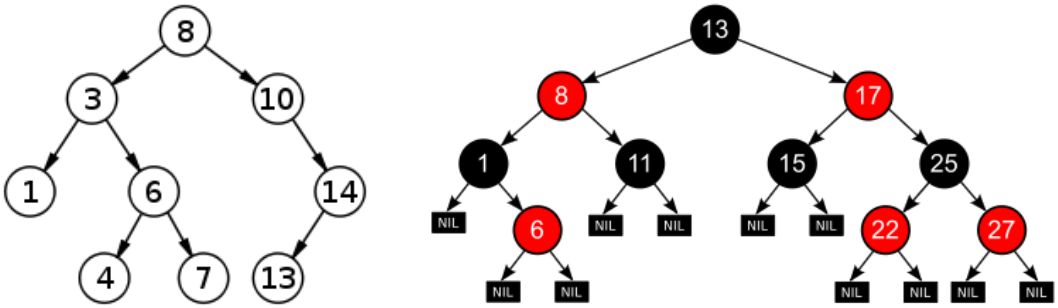
- RN: identico ad ABR

Ora vorrei descrivere le particolarità di queste due tipologia di alberi partendo da ABR. Le proprietà di un ABR (esempio in figura 1a) sono:

1. Il sottoalbero sinistro di un nodo x contiene soltanto i nodi con chiavi minori della chiave del nodo x.
2. Il sottoalbero destro di un nodo x contiene soltanto i nodi con chiavi maggiori della chiave del nodo x.
3. Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due ABR.

Le proprietà di un RN (esempio in figura 1b) sono:

1. Ogni nodo è rosso o nero.
2. La radice è nera.
3. Ogni foglia (NIL) è nera.
4. Se un nodo è rosso, allora entrambi i suoi figli sono neri.
5. Per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri.
6. RN ha tutte le proprietà di ABR.



(a) Albero binario di ricerca (b) Albero rosso nero

Figure 1: Tipologie di alberi

2.3 Assunti ed ipotesi

In un ABR le operazioni di base richiedono un tempo proporzionale all'altezza dell'albero. L'altezza attesa di un ABR costruito in modo casuale è $O(h)$ quindi le operazioni elementari svolte su questo tipo di albero richiedono in media $\Theta(h)$. Nel caso peggiore da noi descritto come la catena lineare l'altezza è $\Theta(n)$ e quindi ci aspettiamo che le operazioni elementari richiedano $\Theta(n)$ per essere svolte. Per vedere la complessità degli algoritmi più importanti di ABR basati sul caso peggiore e sul caso medio si richiama alla figura 2 facendo particolare attenzione ai metodi in rosso che sono quelli su cui andremo a svolgere gli esperimenti.

	Complessità al caso peggiore	Complessità al caso medio
Spazio	$\Theta(n)$	$\Theta(n)$
Inserimento	$O(n)$	$O(h)$
Ricerca	$O(n)$	$O(h)$
Cancellazione	$O(n)$	$O(h)$

Figure 2: Complessità degli algoritmi di ABR

Nel caso invece di alberi RN le operazioni di base richiedono sempre tempo proporzionale all'altezza dell'albero con la differenza che il caso peggiore e il caso medio hanno complessità $O(h)$. Le operazioni sono estremamente veloci se l'altezza dell'albero è piccola, ma al crescere dell'altezza si ha un peggioramento delle prestazioni tanto da avvicinarsi a quelle di una lista concatenata. Descriviamo le varie complessità degli algoritmi in figura 3.

	Complessità al caso peggiore	Complessità al caso medio
Spazio	$\Theta(n)$	$\Theta(n)$
Inserimento	$O(h)$	$O(h)$
Ricerca	$O(h)$	$O(h)$

Figure 3: Complessità degli algoritmi di RN

Il nostro obiettivo in questo test è verificare sperimentalmente la veridicità delle varie complessità descritte nelle figure 2 e 3 e capire sotto quali condizioni un albero è più conveniente di un altro confrontandoli, a parità di spazio, in base al tempo reale che ci mettono a completarsi.

3 Documentazione del codice

3.1 Schema del contenuto e interazione tra i moduli

Per svolgere i nostri esperimenti ho, prima di tutto, scritto il codice delle strutture dati a cui faremo riferimento. In questo caso le classi hanno nome **ABR** per gli alberi binari di ricerca e **RN** per gli alberi rosso neri. Le classi **Node** e **NodeRN** sono le classi che descrivono i nodi. La classe **Node** è pensata per essere 'usata' dalla classe **ABR** mentre la classe **NodeRN** per **RN**. Notiamo che **RN** è una sottoclasse di **ABR** per il semplice motivo che gli alberi rosso neri sono degli alberi binari di ricerca con specifiche regole in più descritte nel capitolo 2.2. Stessa cosa vale per le classi **Node** e **NodeRN** (**NodeRN** è una sottoclasse di **Node**). Entrambe le tipologie di nodi si aggregano con i rispettivi alberi (Basta solo l'istanza del nodo della radice). Importante notare come la classe **Node** (e quindi conseguentemente la classe **NodeRN**) abbia un vincolo di aggregazione ricorsivo di moltiplicità 3. Questo è dovuto al fatto che un oggetto della classe **Node** deve avere al suo interno le istanze del nodo padre e dei nodi figli.

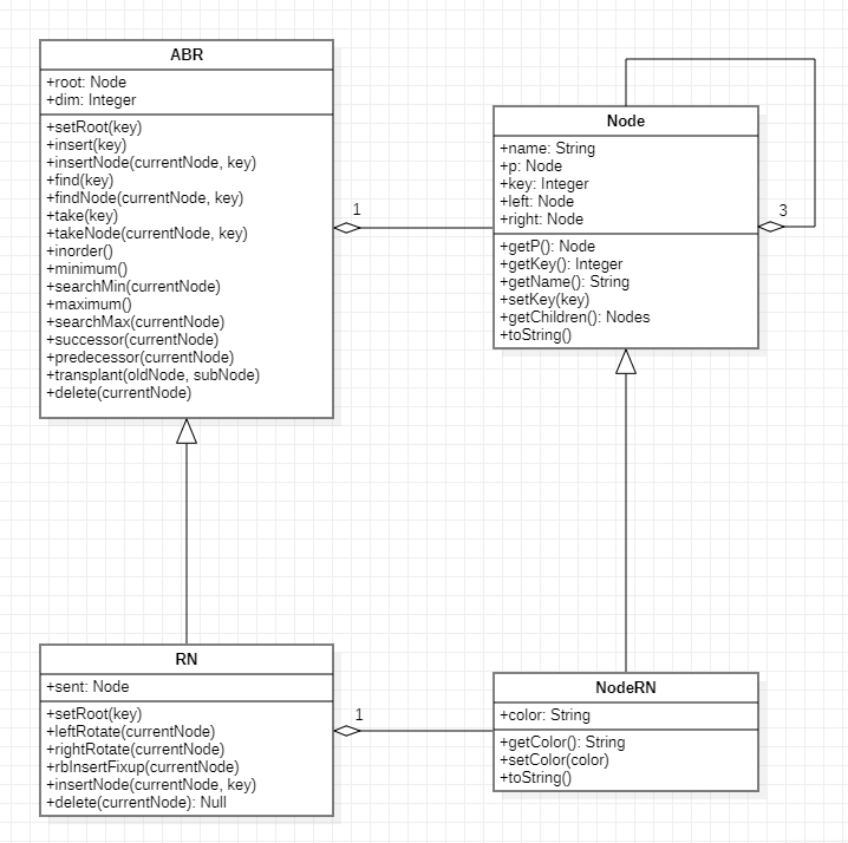


Figure 4: Diagramma delle classi ABR e RN

In più ho creato una classe semplicemente al fine di svolgere questo esperimento. **PlotGenerator** è una classe utile per la generazione di infografiche utili per vedere la funzione che si evolve nel tempo dei due metodi che andrò ad analizzare.

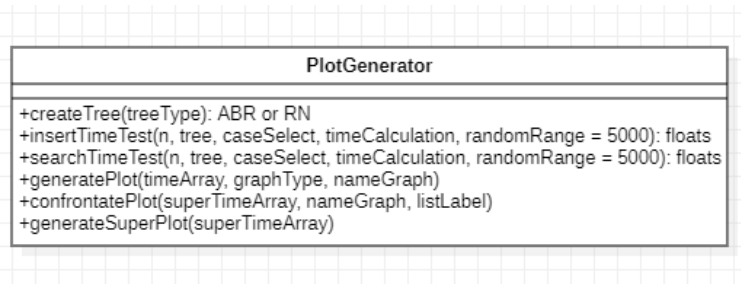


Figure 5: Diagramma della classe PlotGenerator

3.2 Analisi delle scelte implementative

Con la descrizione delle varie classi e delle interazioni tra loro vanno comprese anche alcune delle scelte implementative utili al funzionamento delle strutture dati in questione. Partendo dalla classe **RN** si può notare l'attributo *sent*. In questo attributo instanzieremo un oggetto di tipo **NodeRN** con le seguenti qualità:

- nessun padre
- colore nero
- chiave -1

Questa sentinella sarà usata come 'padre' della radice e come 'figlio' di tutti quei nodi che non hanno qualcuno dei 'figli'. Ho inserito anche un attributo *dim* nelle classi degli alberi che servirà per il calcolo delle infografiche. Per il resto non ci sono grandi differenze implementative rispetto a quello che un qualsiasi libro di Algoritmi e Strutture Dati introduce.

3.3 Descrizione dei metodi implementati

In questa parte descriverò le funzionalità di ogni metodo delle classi di cui finora abbiamo parlato.

- **Node**
  - **getP()** : restituisce il nodo padre *p*.
  - **getKey()** : restituisce l'attributo *key*.
  - **getName()** : restituisce la stringa *name*.
  - **setKey(key)** : modifica l'attributo *key* della classe con il valore del parametro.
  - **getChildren()** : restituisce un array di nodi (più precisamente *left* e *right* in quest'ordine).
  - **toString()** : stampa a schermo gli attributi del nodo.
- **NodeRN**
  - come sottoclasse di **Node** può usare tutti i metodi della superclasse e fare override di alcuni di essi.
  - **getColor()** : restituisce la stringa *color*.
  - **setColor(color)** : modifica la stringa *color* della classe con il valore del parametro.
  - **toString()** : stampa a schermo gli attributi del nodo più l'attributo color.
- **ABR**
  - **setRoot(key)** : instanzia un oggetto **Node** su *root*.
  - **insert(key)** : chiama il metodo **setRoot(key)** se *root* è NULL altrimenti chiama il metodo **insertNode(currentNode, key)**. Incrementa *dim*.
  - **insertNode(currentNode, key)** : attraversa l'albero ricorsivamente fino a che non trova una foglia libera. A questo punto crea un istanza di **Node** con chiave *key*.
  - **find(key)** : ritorna il valore del metodo **findNode(self.root, key)**.
  - **findNode(currentNode, key)** : attraversa l'albero ricorsivamente e se trova un nodo con valore uguale a *key* allora ritorna TRUE altrimenti ritorna FALSE.
  - **take(key)** : ritorna il valore del metodo **takeNode(self.root, key)**.
  - **takeNode(currentNode, key)** : attraversa l'albero ricorsivamente e se trova un nodo con valore uguale a *key* allora ritorna *currentNode* altrimenti ritorna FALSE.
  - **inorder()** : stampa a schermo tramite un attraversamento simmetrico dell'albero tutti gli attributi dei nodi dell'albero.
  - **minimum()** : restituisce NULL se l'attributo *root* è NULL altrimenti chiama **searchMin(self.root)**.
  - **searchMin(currentNode)** : scende ricorsivamente l'albero usando come attributo *currentNode.left*. Se con la verifica *currentNode.left* è NULL allora restituisce *currentNode*.
  - **maximum()** : restituisce NULL se l'attributo *root* è NULL altrimenti chiama **searchMax(self.root)**.

  - **searchMax(currentNode)** : scende ricorsivamente l'albero usando come attributo *currentNode.right*. Se con la verifica *currentNode.right* è NULL allora restituisce *currentNode*.
  - **successor(currentNode)** : restituisce l'oggetto di tipo **Node** con chiave *key* con il valore immediatamente successivo al valore della chiave di *currentNode* all'interno dell'albero.
  - **predecessor(currentNode)** : restituisce l'oggetto di tipo **Node** con chiave *key* con il valore immediatamente precedente al valore della chiave di *currentNode* all'interno dell'albero.
  - **transplant(oldNode, subNode)** : sostituisce il sottoalbero con radice *oldNode* con il sottoalbero di radice *subNode*.
  - **delete(currentNode)** : chiama il metodo **transplant(oldNode, subNode)** per la sostituzione del sottoalbero mantenendo la validità delle proprietà dell'ABR.

- **RN**
  - come sottoclasse di **ABR** può usare tutti i metodi della superclasse e fare override di alcuni di essi.
  - **setRoot(key)** : instanzia un oggetto **Node** su *root* che avrà come padre e figli l'attributo *sent*.
  - **leftRotate(currentNode)** : operazione locale che effettua una rotazione tra *currentNode* e *currentNode.right*.
  - **rightRotate(currentNode)** : operazione locale che effettua una rotazione tra *currentNode* e *currentNode.left*.
  - **rbInsertFixup(currentNode)** : modifica della struttura dell'albero che usa i due metodi **leftRotate(currentNode)** e **rightRotate(currentNode)** oltre a modificare gli attributi *color* dei vari nodi secondo le regole dello RN.
  - **insertNode(currentNode, key)** : simile al metodo di **ABR** con la chiamata del metodo **rbInsertFixup(currentNode)** alla fine.
  - **delete(currentNode)** : metodo non implementato. Restituisce NULL.

- **PlotGenerator**
  - **createTree(treeType)** : restituisce un'istanza di una delle tipologie di albero. Se *treeType* è FALSE restituisce un'istanza di un oggetto **ABR** viceversa se TRUE restituisce un'istanza di un oggetto **RN**.
  - **insertTimeTest(n, tree, caseSelect, timeCalculation, randomRange=5000)** : restituisce una lista di float contenente i tempi che ci vogliono per l'esecuzioni in serie del metodo **insert(key)**. Se *caseSelect* è FALSE prenderemo il caso della catena lineare viceversa il caso randomico se TRUE. Nel caso randomico i numeri che verrano generati all'interno dell'albero andranno da 0 a *randomRange*. Se *timeCalculation* è FALSE il calcolo del tempo avviene tramite la differenza tra il tempo di fine e il tempo di inizio più il tempo precedentemente registrato nell'array viceversa il calcolo del tempo avviene tramite la differenza tra il tempo di fine e il tempo di inizio diviso la dimensione dell'albero *dim* più il tempo precedentemente registrato nell'array se TRUE.
  - **searchTimeTest(n, tree, caseSelect, timeCalculation, randomRange=5000)** : restituisce una lista di float contenente i tempi che ci vogliono per l'esecuzioni in serie del metodo **find(key)**. Il funzionamento di *caseSelect*, *timeCalculation* e *randomRange* è uguale ad **insertTimeTest(n, tree, caseSelect, timeCalculation, randomRange=5000)**.
  - **generatePlot(timeArray, graphType, nameGraph)** : genera un infografica che rappresenta la funzione nel tempo data da *timeArray*. *nameGraph* è il nome che verrà assegnato al grafico. *graphType* offre due tipi di rappresentazione per il grafico: se FALSE offre una rappresentazione continua viceversa discreta se TRUE.
  - **confrontatePlot(superTimeArray, nameGraph, listLabel)** : genera un infografica data da più funzioni. Il numero di funzioni è uguale alla dimensione di *superTimeArray*. Simile a **generatePlot(timeArray, graphType, nameGraph)** ma con un solo tipo di rappresentazione, quella continua. *listLabel* è una lista contenente i nomi di ogni label che vanno assegnanati alle funzioni.
  - **generateSuperPlot(superTimeArray)** : genera delle infografiche con tutti i possibili confronti utili alla nostra sperimentazione. *superTimeArray* è la lista che contiene tutti i *timeArray* elementari (cioè tutte le possibili combinazioni esistenti).

4 Descrizione degli esperimenti condotti e analisi dei risultati sperimentali - Esercizio A

4.1 Dati utilizzati

L'esperimento che ho svolto si divide prima di tutto per il numero di nodi che sono andato ad inserire nei due alberi. Quindi divideremo l'esperimento in 4 parti:

- alberi con **33** elementi
- alberi con **100** elementi
- alberi con **300** elementi
- alberi con **900** elementi

Il numero degli elementi scelto è basato semplicemente su tre cose:

- Far vedere il caso discreto con alberi con pochi elementi (nel nostro caso quello da 33).
- Fare una prova con alberi con tre volte tanto gli elementi della prova precedente (è arbitraria la moltiplicazione per 3).
- Non superare i 1000 elementi in un albero per l'impossibilità di avere più di un totale di ricorsioni (Il motivo è che il compilatore da il seguente errore: RecursionError: maximum recursion depth exceeded while getting the str of an object).

Prendendo il caso della randomizzazione ho deciso di fare un test per verificare se il range in cui prendo i numeri influenzi il tempo che ci mette per lo svolgimento dell'algoritmo. Questo caso è stato testato con gli alberi da 900 elementi per avere una miglior idea delle tempistiche. I range che ho usato sono i seguenti:

- **0-100**
- **0-1000**
- **0-10000**

Per il resto delle prove invece utilizzeremo un range di base **0-5000**. Il caso a cui farò più attenzione sarà invece quello con 300 elementi in cui confronterò i vari grafici e cercherò di capire a parità di operazioni quanto tempo ci mettono i metodi ad eseguirsi con successo dipendentemente dagli alberi di scelta. Nel caso di 100 elementi invece mostrerò i diversi metodi di calcolo del tempo.

4.2 Misurazioni

I metodi di calcolo dei tempi nei metodi **insertTimeTest** e **searchTimeTest** sono di due tipi. Innanzitutto dobbiamo prendere il tempo prima dell'esecuzione del metodo e quello dopo la sua esecuzione.

Questo fa si che per calcolare i tempi si possano usare:

$$timeArray[x+1] = (end - start) + timeArray[x]; \tag{1}$$

che calcola il tempo netto che ci vuole ogni qualvolta si aggiunga un nodo in più all'albero e lo somma al precedente. Oppure possiamo usare quest'altro metodo di misurazione:

$$timeArray[x+1] = ((end - start)/tree.dim) + timeArray[x]; \tag{2}$$

che calcola sempre il tempo che ci vuole ogni qualvolta si aggiunga un nodo in più all'albero che però viene diviso per la dimensione attuale dell'albero ed somma il risultato al precedente. Questo ultimo tipo di calcolo sarà di particolare interesse per il calcolo della complessità computazionale del nostro algoritmo. Per decidere quale metodo calcolare si richiama alla sezione 3.3.

```
#decide come prendere i tempi
#False - tempo regolare
#True - tempo in base al numero di nodi
if(not timeCalculation):
    timeArray[x+1]=(end-start)+timeArray[x];
else:
    timeArray[x+1]=((end-start)/tree.dim)+timeArray[x];
```

Figure 6: Codice per il calcolo dei tempi

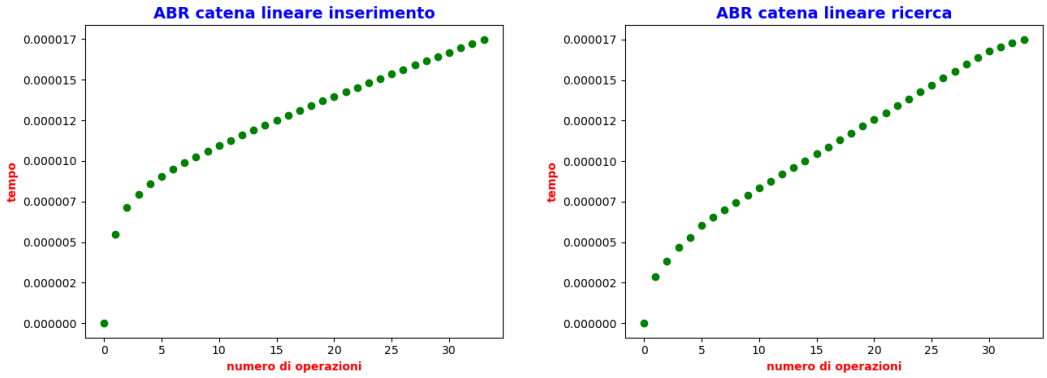
IMPORTANTE : Il primo valore di ogni funzione parte da 0. Quel valore è standard e non rappresenta il tempo di nessuna operazione eseguita. Partire dal valore 1 per avere una miglior analisi. I motivi di tale valore sono :

- partenza delle funzioni tutte dal tempo 0
- calcolo del tempo ricorsivo

4.3 Risultati sperimentali e commenti analitici

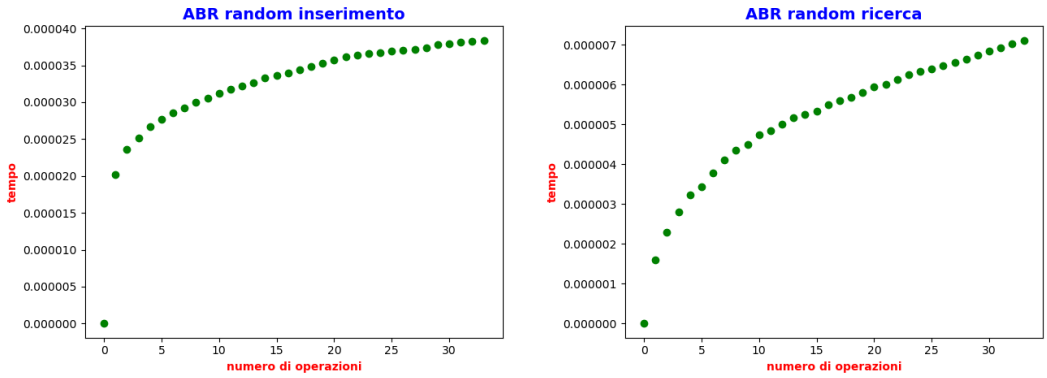
4.3.1 alberi con 33 elementi

Come detto nella sezione 4.2 il primo elemento è una misura presa di base quindi non farà parte delle considerazioni. Notiamo come nel caso in figura 7 la tendenza è lineare quindi possiamo dire che i due metodi hanno complessità $\Theta(n)$. Nei casi nelle figure 8,9 e 9 si osserva invece che la tendenza è $O(h)$. Il calcolo del tempo è effettuato con la formula 2 del paragrafo 4.2.



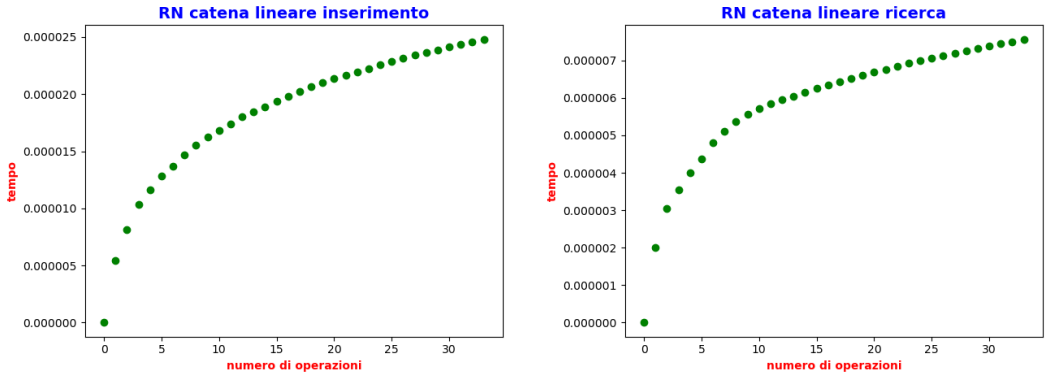
(a) Inserimento (b) Ricerca

Figure 7: ABR catena lineare



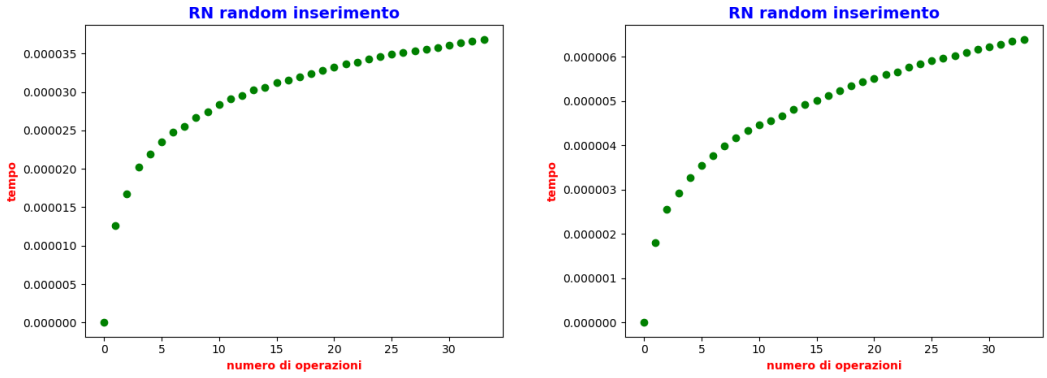
(a) Inserimento (b) Ricerca

Figure 8: ABR random



(a) Inserimento (b) Ricerca

Figure 9: RN catena lineare

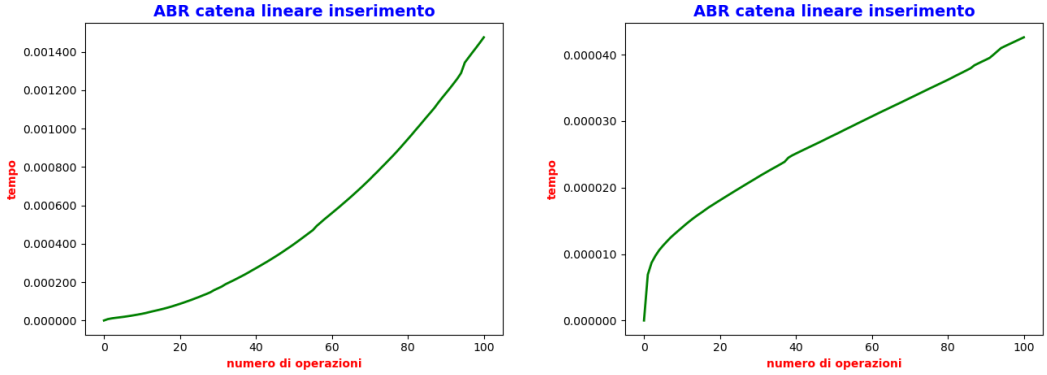


(a) Inserimento (b) Ricerca

Figure 10: RN random

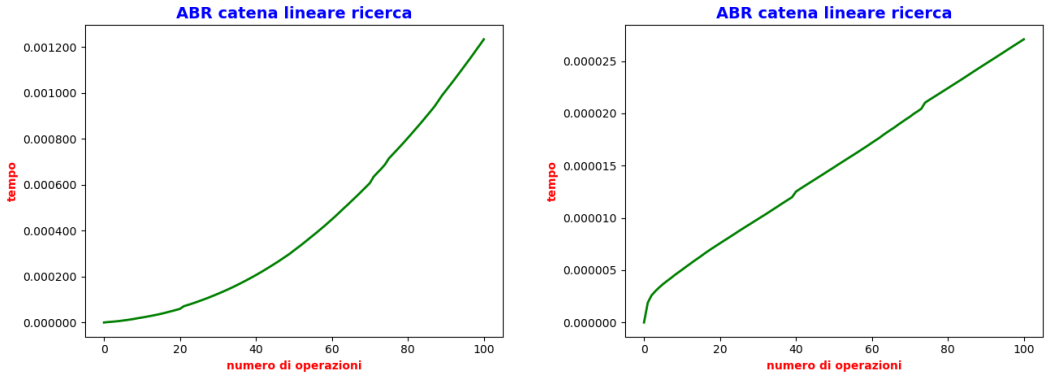
4.3.2 alberi con 100 elementi

In questo paragrafo si descrive i vari metodi di calcolo. Dai grafici sottostanti notiamo che le complessità sono più o meno simili apparte che nelle figure 11 e 12 le quali sono i grafici delle funzioni dell'ABR nel caso della catena lineare. Il fatto che l'albero alla fine dell'operazione di inserimento non si "aggiusta" rende i tempi di inserimento esponenzialmente più lunghi con l'aggiunta di nuovi nodi mentre nelle altre sezioni vediamo che più o meno i tempi hanno sempre la stessa complessità computazionale. Tutte le sezioni **a** delle figure sono state calcolate con la formula 1 mentre le sezioni **b** con la formula 2 (Per le formule guardare la sezione 4.2).



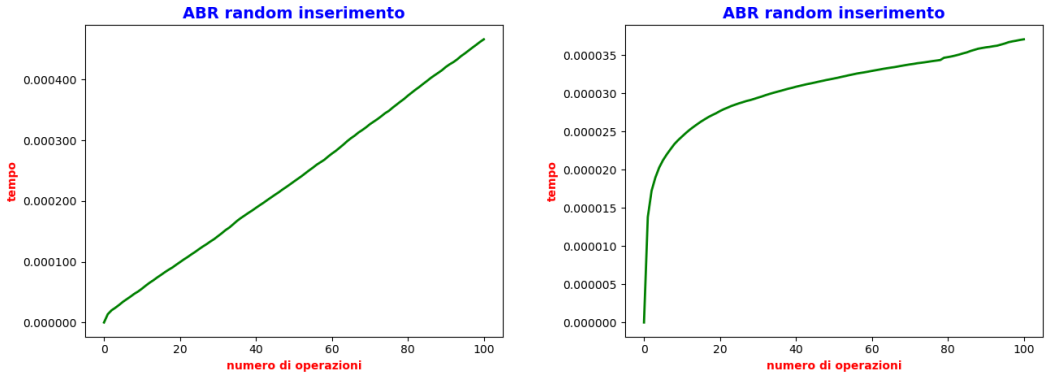
(a) Formula 1 (b) Formula 2

Figure 11: ABR catena lineare inserimento



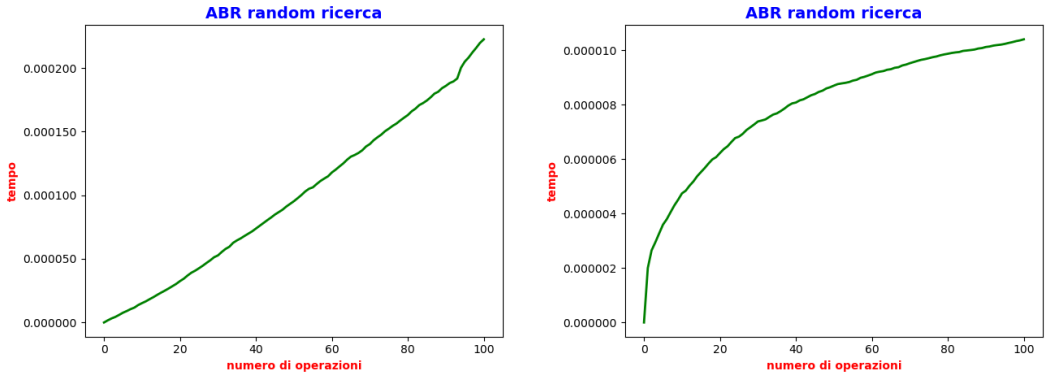
(a) Formula 1 (b) Formula 2

Figure 12: ABR catena lineare ricerca



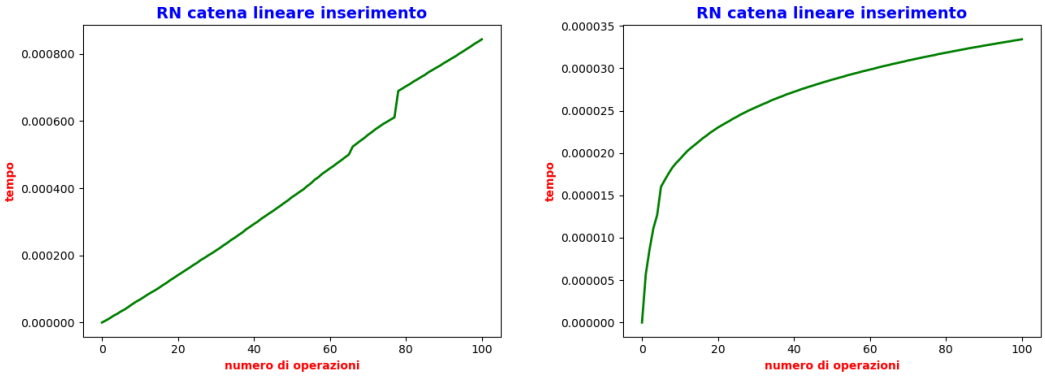
(a) Formula 1 (b) Formula 2

Figure 13: ABR random inserimento



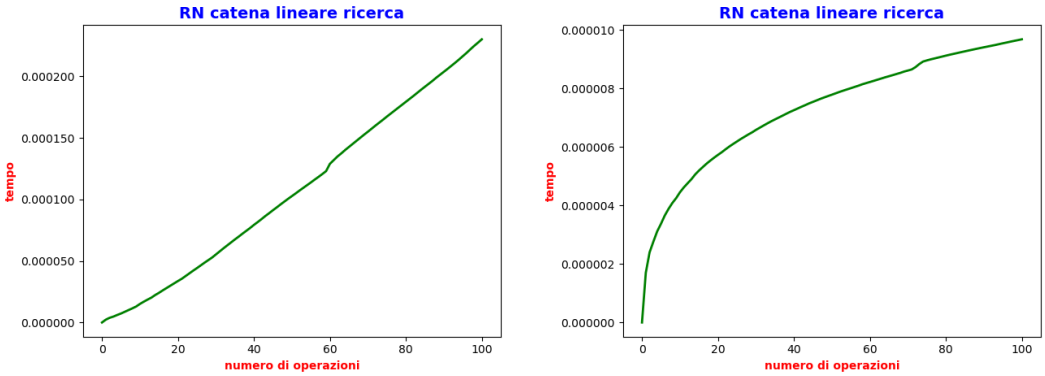
(a) Formula 1 (b) Formula 2

Figure 14: ABR random ricerca



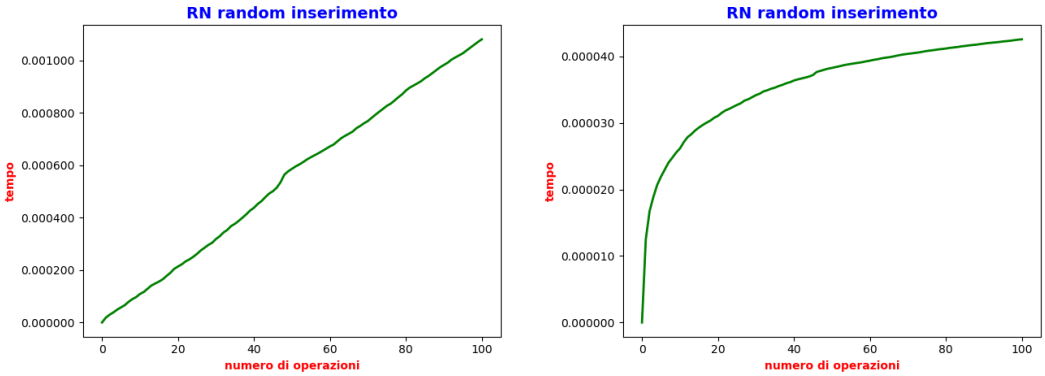
(a) Formula 1 (b) Formula 2

Figure 15: RN catena lineare inserimento



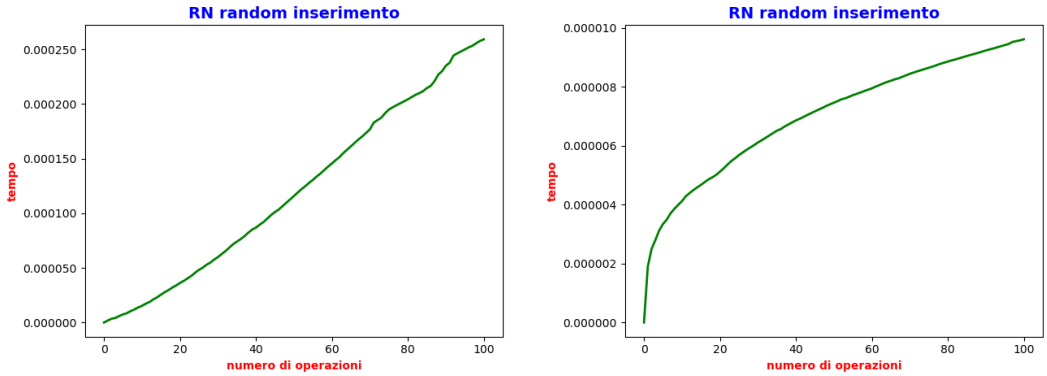
(a) Formula 1 (b) Formula 2

Figure 16: RN catena lineare ricerca



(a) Formula 1 (b) Formula 2

Figure 17: RN random inserimento



(a) Formula 1 (b) Formula 2

Figure 18: RN random ricerca

4.3.3 alberi con 300 elementi

In questa sezione si confrontano i tempi dei vari metodi rispetto ad alcuni fattori che possono essere determinanti per comprendere meglio il funzionamento degli alberi da noi presi in questione. In figura 19 possiamo notare che nei casi dell'inserimento e della ricerca nel caso dell'ABR conviene randomizzare la scelta dei valori dei nodi arrivati all'intersezione tra le funzioni perchè le funzioni del tempo date dalla randomizzazione hanno complessità computazionale $O(h)$ mentre nel caso della catena lineare la complessità è $\Theta(n)$. Invece niente sembra si possa dire nel caso dell'inserimento in RN visto che la complessità computazionale rimane la stessa a prescindere del fatto che le chiavi dei nodi vengano prese randomicamente o come una catena lineare e le curve sembrano stare in qualche modo parallele (probabilmente è dovuto al fatto che il riequilibrio dell'albero RN influisce parecchio sulle tempistiche, probabilmente più del tipo di inserimento che andiamo ad effettuare). Aggiungiamo un'osservazione: sulla ricerca RN il tempo dato dalla randomizzazione sembra alla lunga migliore di quello dato dalla catena lineare. Questo è dovuto però al nodo che andiamo a cercare : infatti il nodo che viene cercato è quello che viene inserito poco prima nell'albero. Questo significa che per fare la ricerca nel caso della catena lineare il metodo dovrà arrivare in fondo all'albero, cosa che non è detta nel caso della randomizzazione. Passando all'inserimento con metodo della catena lineare in figura 20 possiamo evidenziare sempre la differenza data proprio dalla complessità computazionale, che nel caso di ABR è evidentemente maggiore rispetto a quella in RN. Stessa cosa vale nella ricerca con catena lineare. Notiamo adesso l'inserimento randomico : qui è evidente come la curva data da RN tenda ad aumentare rispetto a quella di ABR. Questo perchè RN deve riequilibrare ogni volta l'albero e ad ogni aggiunta il riequilibrio ci mette sempre di più a svolgersi. Questa cosa ovviamente non vale per gli ABR e questo li rende migliori a livello di tempistiche alla lunga rispetto ad RN. Questo però andrà probabilmente ad influenzare la ricerca nel caso randomico che come si vede infatti è migliore nel caso RN proprio perchè c'è un riequilibrio dell'albero e questo consente in media di trovare le chiavi dei nodi più velocemente rispetto ad un ABR. Nella figura 21 possiamo verificare quale metodo ci mette più o meno tempo a seconda del caso e della tipologia di albero. Notiamo che apparte nei casi dati da ABR con catena lineare le altre combinazioni hanno stessa complessità $O(h)$ e nessuna combinazione prevale significativamente sull'altra in nessuno dei due metodi. Con quest'ultimo confronto possiamo redarre e confermare le tabelle 2 e 3 della sezione 2.3

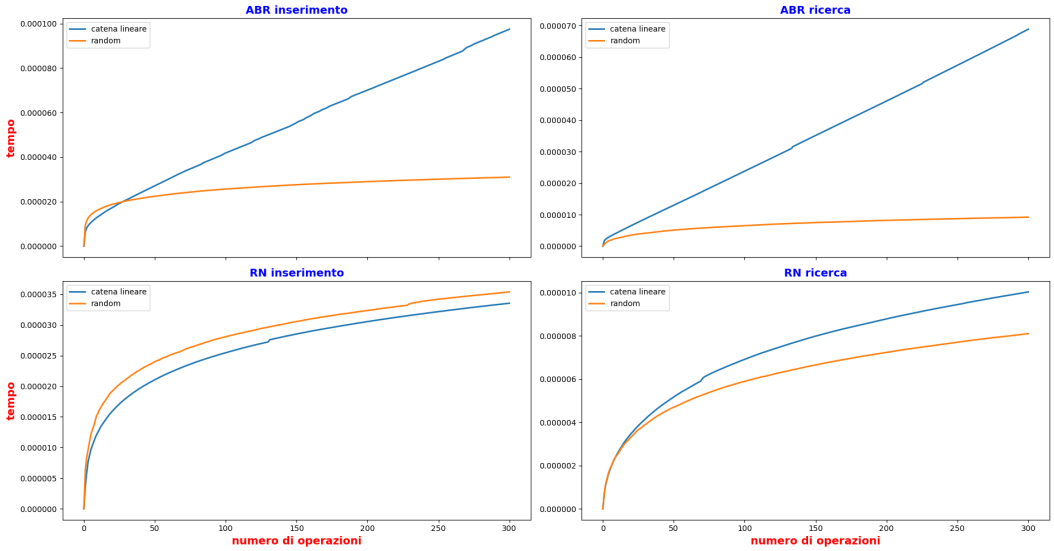


Figure 19: Confronto dei metodi rispetto al tipo di caso

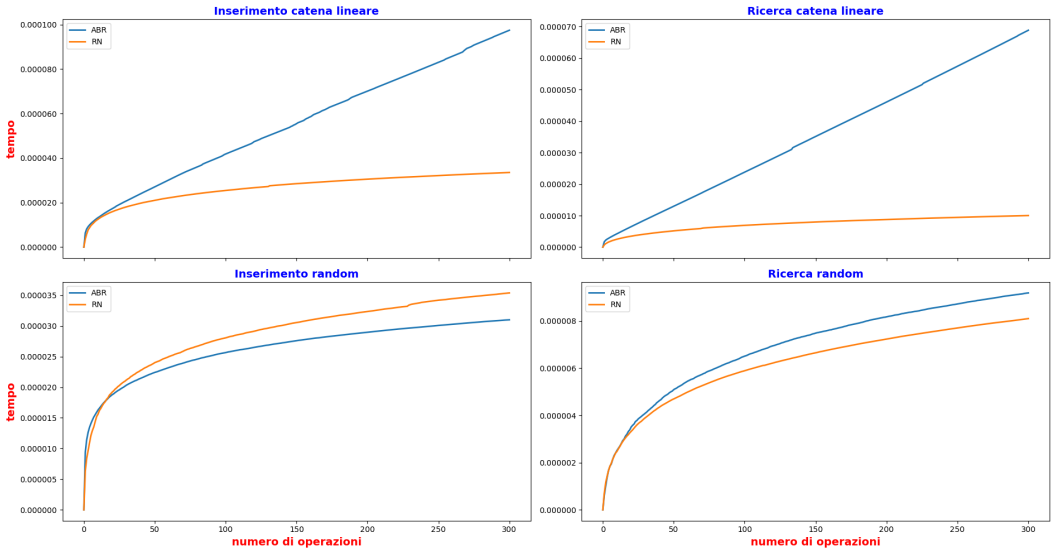


Figure 20: Confronto dei metodi rispetto all'albero

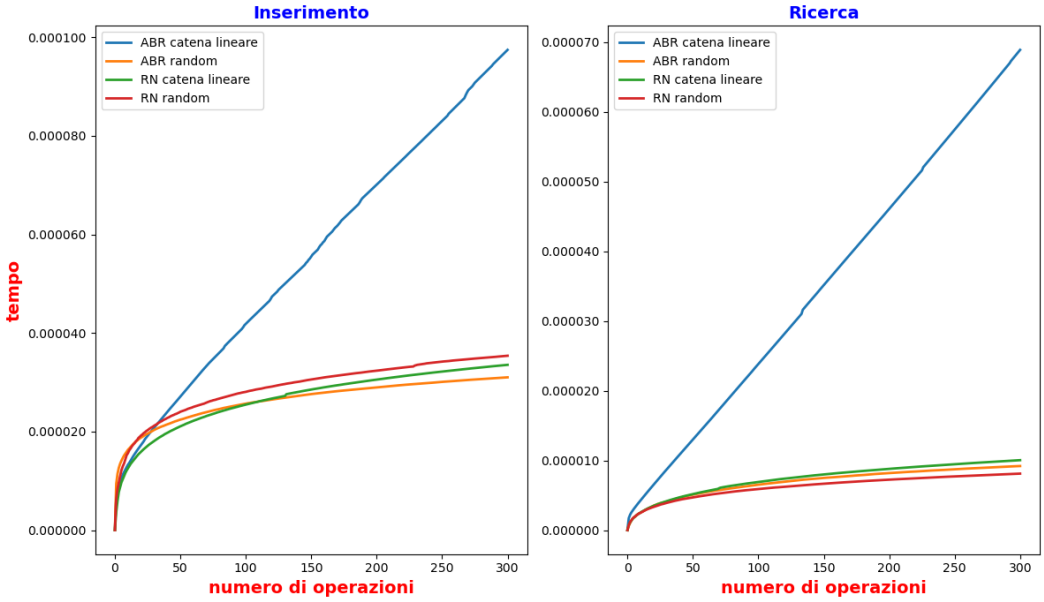
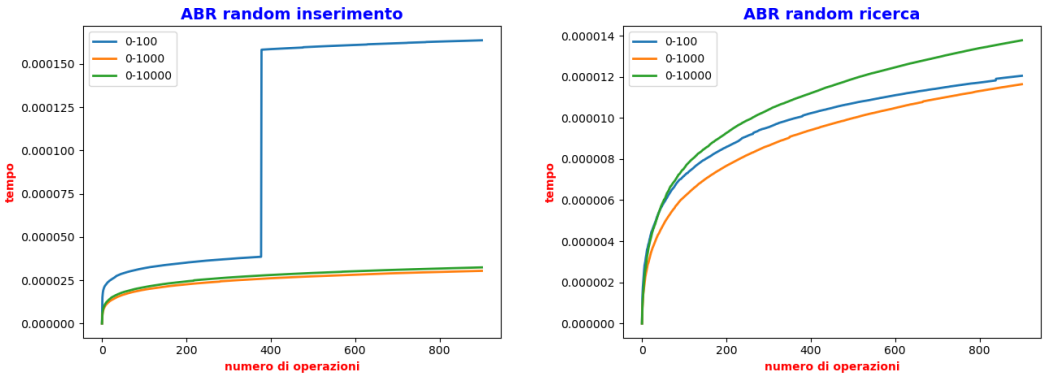


Figure 21: confronto generale dei metodi

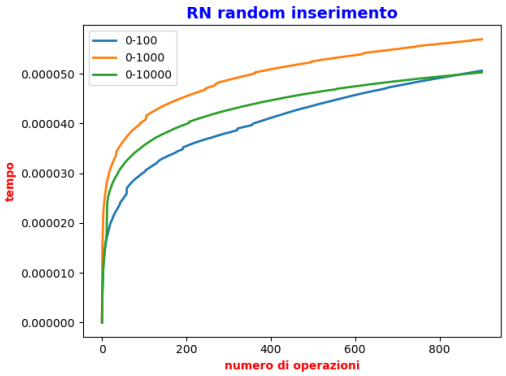
4.3.4 alberi con 900 elementi

In questo test vogliamo verificare nel caso dei metodi random se i tempi di svolgimento delle operazioni, dato un diverso range da dove vengono prese le chiavi che vengono assegnati ai nodi, a parità di numero di nodi varia. Guardando le figure 22a, 22b, 23a e 23b sembrano esserci variazioni significative che possono dipendere dal range di numeri presi, soprattutto in figura 22a(c'è da capire se sia dovuto ad un bug), anche se dobbiamo far notare che il tempo di svolgimento delle operazioni è talmente veloce che difficilmente possiamo notare qualche differenza sostanziale. Non sembra quindi esserci alcuna evidenza che dimostri che il cambiamento del range preso influenzi i tempi di svolgimento di un algoritmo. Per verificare quindi se ci sia una qualche correlazione data dobbiamo svolgere lo stesso test più volte e vedere tramite dati (che assumeremo essere il tempo più alto trovato per ogni nostra funzione) se ci sia una sorta di dipendenza tra range e tempo. Per fare ciò dovremmo utilizzare un po' di statistica. In particolare useremo il concetto di mediana. Il motivo del perchè scegliamo la mediana è perchè ci possono essere valori che influiscono estremamente sul valore della media e che quindi altereranno inevitabilmente il risultato finale. La mediana infatti non risente dei valori estremi che possono generarsi. Il numero di prove che verranno svolte per prendere i valori mediani deve essere abbastanza alto da garantirci una sicurezza nel prendere il valore mediano. Nel nostro caso saranno 200 prove (il valore scelto è arbitrario ma abbastanza alto). Come vediamo dalla tabella in figura 24 il valore in microsecondi varia leggermente in ogni tipologia di operazione, soprattutto per quanto riguarda la ricerca in cui sembra sempre aumentare all'aumentare del range preso, ma essendo una variazioni in microsecondi estremamente piccola possiamo dire che il range da cui prendiamo i valori per i nodi non influisce in maniera rilevante in nessun tipo di operazione.

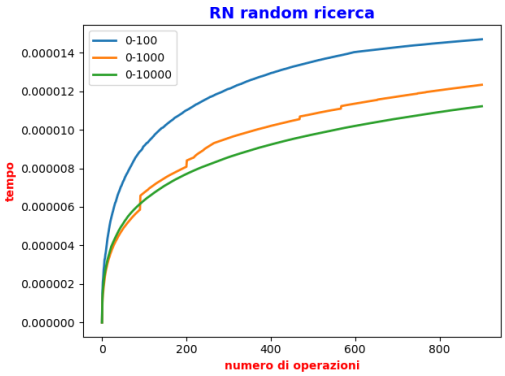


(a) ABR inserimento random

(b) ABR ricerca random



(a) RN inserimento random



(b) RN ricerca random

	ABR inserimento	ABR ricerca	RN inserimento	RN ricerca
0-100	30.4	11.4	42.5	9.4
0-1000	30.2	13.0	42.5	10.7
0-000	31.0	13.4	43.5	11.0

Table 1: Tabella di rappresentazione dei valori mediani per operazione dopo 200 test con valori dati in microsecondi

4.4 Tesi e sintesi finale

Come già descritto nella sezione 4.3 possiamo dare le seguenti conclusioni:

- I metodi inserimento e ricerca nel caso della catena lineare con albero ABR hanno complessità $\Theta(n)$. In tutti gli altri casi descritti la complessità è $O(h)$.
- I metodi inserimento e ricerca nel caso della catena lineare con albero ABR ci mettono un tempo esponenziale all'aumentare del numero di nodi inseriti basato sulla formula 1 della sezione 4.2. Tutti gli altri metodi ci mettono tempo lineare sempre basato sulla formula 1.
- Nel caso dell'inserimento randomico si può notare che ABR è migliore di RN (Questo è probabilmente dovuto al riequilibrio dell'albero RN). Questa tendenza sembra invertirsi nel caso della ricerca randomica (Infatti avendo riequilibrato in albero in media ci vorrà meno tempo a cercare un nodo all'interno di un RN rispetto ad un ABR). Questo fa notare che il riequilibrio dell'albero RN può essere utile per lo svolgimento di alcune operazioni che potrebbero metterci molto più tempo senza riequilibrio.
- Il tempo delle operazioni randomiche non sembra variare significativamente a seconda del range di numeri che possono essere usati come chiave nei nodi degli alberi. L'affermazione precedente è dovuta al fatto che dopo aver effettuato un numero elevato di test non si è rilevato un aumento o un decremento eccessivo nei tempi dati dalle varie mediane calcolate per ogni tipologia di operazione valutata.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009) Introduzione agli algoritmi e strutture dati Terza edizione, McGraw Hill.