UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze

Dipartimento di Ingegneria dell'Informazione

Laboratorio di Algoritmi e Strutture Dati

Autore:
Jacopo Mechi

N° Matricola:
7047035

Corso principale:
Algoritmi e Strutture Dati

Docente corso:
Simone Marinai

Indice

Elenco delle figure

1 Introduzione generale

1.1 Progetto asseganto

Chiavi duplicate in alberi binari di ricerca

1.2 Breve descrizione dello svolgimento del esercizio

Suddivideremo la descrizione del esercizio 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

1.3 Specifiche della piattaforma di test

- **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, la piattaforma in cui il codice è stato scritto è il text editor **NVIM v0.9.2-dev-68+gff689ed1a** e 'girato' sulla shell **zsh 5.9**. La stesura di questo testo è avvenuta con le stesse modalita.

Parte I
Chiavi duplicate in alberi binari di ricerca

Esercizio

- Vogliamo confrontare vari modi per gestire chiavi duplicate in ABR:
  - implementazione ”normale” (senza accorgimenti particolari)
  - utilizzando un flag booleano
  - mantenendo una lista di nodi con chiavi uguali
- Per fare questo dovremo:
  - Scrivere i programmi Python (no notebook) che:
    - implementano quanto richiesto
    - eseguono un insieme di test che ci permettano di comprendere vantaggi e svantaggi delle diverse implementazioni
  - Svolgere ed analizzare opportuni esperimenti
  - Scrivere una relazione (in LATEX) che descriva quanto fatto
  - Nota: le strutture dati devono sempre essere implementate nel progetto; non si possono utilizzare librerie sviluppate da altri o copiare codice di altri

2 Spiegazione teorica del problema

2.1 Introduzione

In questa sezione vedremo una rapida infarinatura teorica sugli alberi binari di ricerca e sulle operazioni di inserimento e di ricerca. Per i nostri esperimenti utilizzeremo queste due operazioni poiche' il codice del inserimento lo avremo necessariamente implementato per popolare gli albberi mentre utilizzeremo l'operazione di ricerca dato che alla base di ogni operazione eseguibile su un ABR c'e' una ricerca (compreso l'inserimento) e quindi i tempi di esecuzione avranno lo stesso andamento per qualsiasi operazione.

2.2 Aspetti fondamentali

Un albero binario di ricerca (esempio in figura 1) e' un tipologia particolare di albero binario con le seguenti caratteristiche:

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.

Descriviamo ora le caratteristiche delle due operazioni che prenderemo in considerazione:

- Per l'inserimento, 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. Vediamo ora la differenza tra le varie implementazioni che andremo adanalizzare, tenendo a mente che esse si differenziano solo per come gestiscono le chiavi duplicate mentre hanno lo stesso comportamento negli altri casi (inserimento destro se la chiave del nuovo node e maggiore e sinistro se la chiave e minore).
  - Classica
  - Flag booleana

  – Lista concatenta di nodi con chiavi uguali

- Nel operazione di ricerca 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.

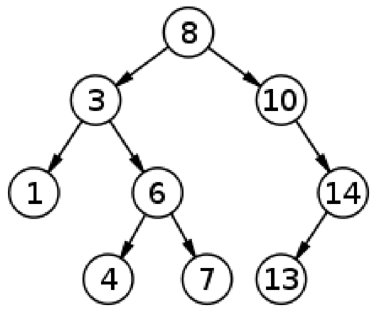

Figura 1: Albero binario di ricerca

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, il caso in cui l'albero sia completamente sbilanciato da un lato, dando cosi origine ad una lista, 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. Poiche andremo a sperimentare su varie metodologie di inserimento e utile ipotizzare come questi avranno impatto sulle prestazioni del albero. In particolare ci aspettiamo di vedere, in caso di un elevato numero di chiavi ripetute, una migliore prestazione nella ricerca, dovuta alla ridotta altezza del albero per quanto riguarda il metodo di inserimento basato sulla lista concatenata e allo stesso modo un altezza inferiore anche per il metodo con flag booleana che dovrebbe rendere piu bilanciat l'albero e quindi ridurne l'altezza. Al contrario il maggior numero di operazioni necessare dovrebbe rendere il metodo con lista, il piu lento nel inserimento di nuovi dati.

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

Figura 2: Complessità degli algoritmi di ABR

Il nostro obiettivo in questo test è verificare sperimentalmente la veridicità delle varie complessità descritte nella figura 2 e capire sotto quali condizioni un albero è più conveniente di un altro confrontandoli, a parità di numero di chiavi, in base al tempo reale che impiegano ad eseguire le operazioni.

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.

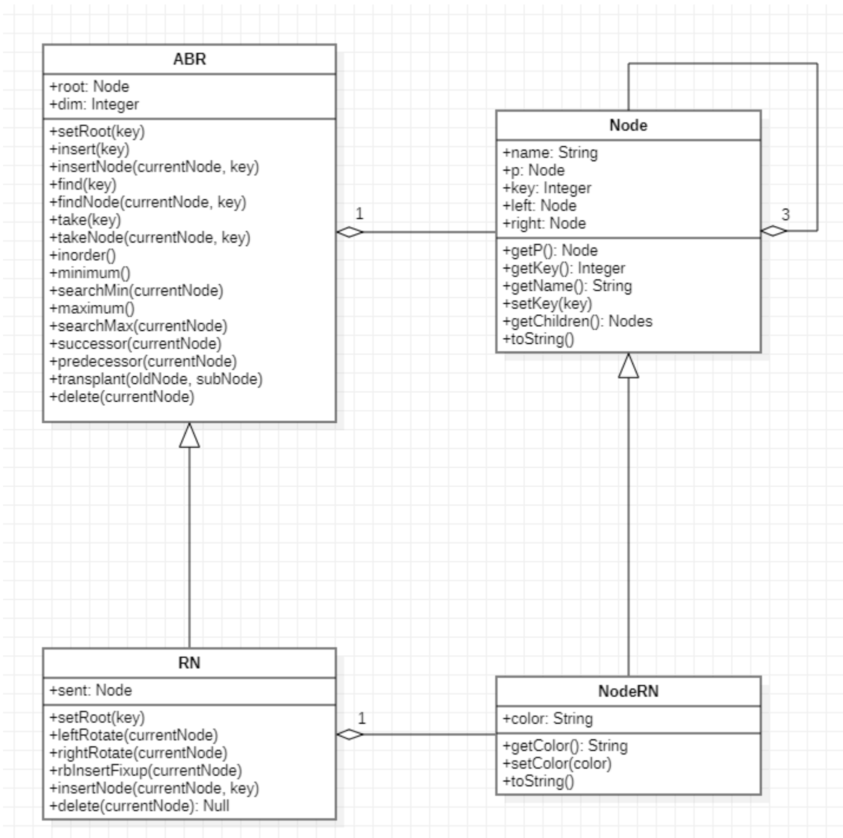

Figura 3: Diagramma delle classi ABR e RN

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.

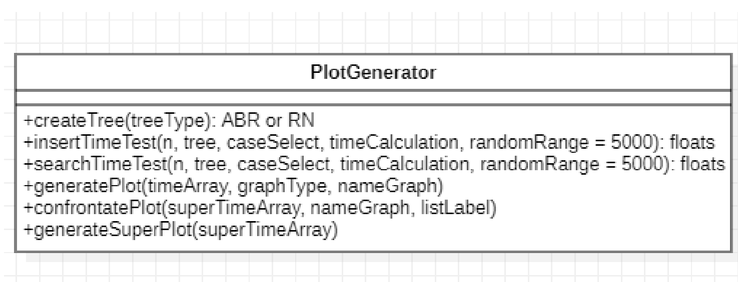

Figura 4: Diagramma della classe PlotGenerator

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

4.1 Dati utilizzati

Per gli esperimenti che andremo ad eseguire utilizzeremo diversi tipi di dataset per poter coprire il maggior numero possibile di casi che si potrebbero incontrare nel effettivo utilizzo di questi algoritmi. Un dataset di test e costituito da un insieme di ”sotto-dataset” che rendono i test completi. Il dataset e’ quindi coposto da 3 tipi di array:

- Random Dataset (RD) ovvero dataset di interi generati casualmente con numeri tra 0 e la dimensione del array
- High Repetitions Dataset (HRD) ovvero dataset di interi generati casualmente con numeri tra 0 e la meta della dimensione del array, cosi da aumentare il numero di ripetizioni
- Worst Case Dataset (WCD) ovvero un HRD ordinato in modo crescente, cosi da ottenere un albero completamente sbilanciato

Ognuno di questi array viene generato in diverse dimensioni (100, 200, 300,...,1000 elementi) in modo da poter vedere la complessita delle operazioni al crescere del dataset. Infine ognuno di questi test viene eseguito 500 volte, per poter minimizzare l’errore dovuto a fattori esterni al programma calcolando la media del tempo impiegato per ogni test.

4.2 Misurazioni

La funzione di benchmark riceve in input il dataset su cui deve eseguire il test e il metodo di inserimento che dovra testare. Per ogni array nel dataset quindi costruisce un albero con il metodo di inserimento specificato tralasciando l’ultimo elemento. Procedera poi a cronometrare il tempo impiegato per l’inserimento del ultimo elemento simulando percio il tempo impiegato ad inserire un dato in un albero gia costruito. In seguito cronometra la ricerca di un intero generato casualmente tra 0 e la dimensione del array fratto 10, cosi che la possibilita di una ricerca con successo sia maggiore ma lasciando comuqnue la possibilita di una ricerca falllita. Questi test vengono eseguiti come detto in precedenza 500 volte per ogni dimensione e viene calcolata una media per ogni dimensione ottenendo come output due array. Un array contenente la media dei tempi di inserimento per ogni dimensione e uno contenente la media dei tempi di ricerca per ogni dimensione.

4.3 Risultati sperimentali e commenti analitici

4.3.1 Inserimento

Iniziamo subito notando come i risultati del test che vediamo in figura **??** confermino quanto ipotizzato dalla nostra analisi teorica 2.3 ottenendo ... come andamento delle operazioni di inserimento su dataset RD e HRD che rappresentano il caso medio. Andando piu in dettaglio con l’analisi possiamo notare come i casi medi indifferentemente dal tipo di dataset abbaino un andamento molto simile a parita di algoritmo utilizzato per l’inserimento coronando l’algoritmo classico come il piu rapido seguito poi dalla lista concatenata e infine dalla flag booleana. A differenza di quanto potesse essere ipotizzabile, data la maggiore complessita delle operazioni per la creazione della lista concatenata, l’algoritmo con lista risulta piu prestante di quello booleano probabilmente aiutato dal fatto che incorporando i nodi simili in uno solo riduce l’altezza del albero e quindi il numero di cicli necessari l’inserimento, a differenza della flag booleana che si limita a cercare di bilanciare meglio l’albero, mantenendo pero il numero di nodi invariato.

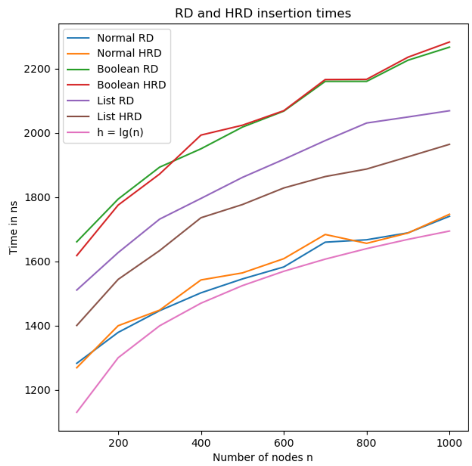

Figura 5: Inserimento su RD e HRD

Anche per quanto riguarda i risultati mostrati in figura **??** possiamo notare come gli esperimenti abbiano comfermato la nostra analisi teorica mostrando un andamento ... per tutti gli algoritmi nel caso peggiore. Non ci sorprende vedere come l'algoritmo basato sulla lista concatenata risulti il piu rapido in quanto riducendo il numero di nodi riduce anche l'altezza del albero (che nel caso peggiore risulta essere una lista) permettendo cosi di ottenere i risultati migliori. Risulta interessante pero notare come l'algoritmo classico risulti piu prestante di quello con flag booleana.

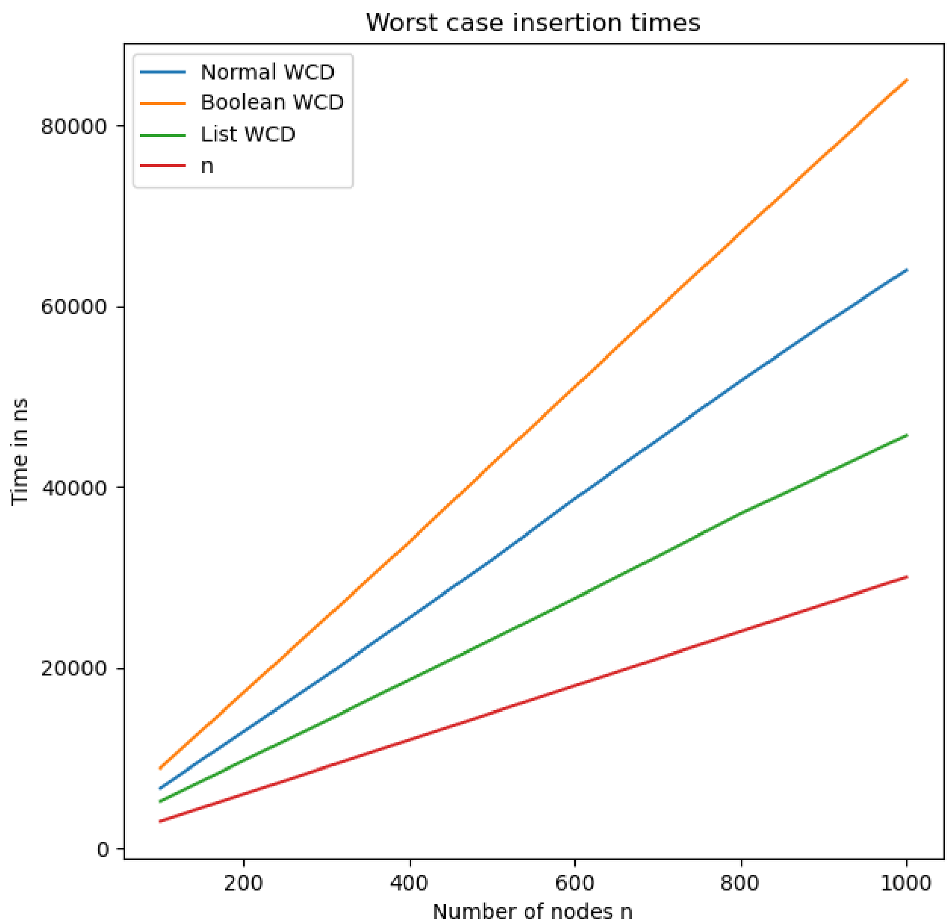

Figura 6: Ricerca

4.3.2 Ricerca

Per quanto riguarda le prestazioni della ricerca mostrate in figura **??** possiamo notare come i risultati sperimentali abbiano confermato le nostre ipotesi con un andamento ... per i dataset RD e HRD. Salta subito al occhio come tutti gli algoritmi abbiano prestazioni migliori nel caso di dataset ad alte ripetizioni dato dal fatto che sono stati ottimizzati appositamente per questa evenienza e non ci sorprende vedere come sia su dataset RD che HRD l'algoritmo con lista concatenata risulti il piu prestante. Aggiungiamo inoltre che su dataset RD i le prestazioni sono quasi identiche per tutti gli algoritmi con un leggero miglioramento per quanto riguarda l'algoritmo a lista condatenata mentre per i dataser HRD l'algoritmo a lista concatenata risulta decisamente piu prestante rispetto agli altri due che danno risultati molto simili.

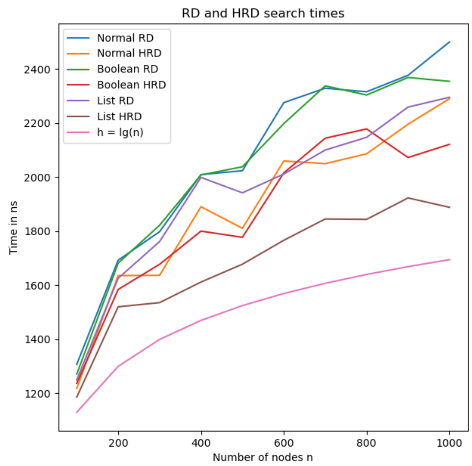

Figura 7: Ricerca su RD e HRD

Come per i casi precedenti anche su dataset WCD i risultati in figura **??** sono concordanti con la nostra previsione di una complessita ... . Nuovamente l'algoritmo con lista concatenata risulta nettamente piu prestante degli altri anche se vediamo un leggero miglioramento di prestazioni da parte del algoritmo con flag booleana rispetto a quello con impplementazione classica.

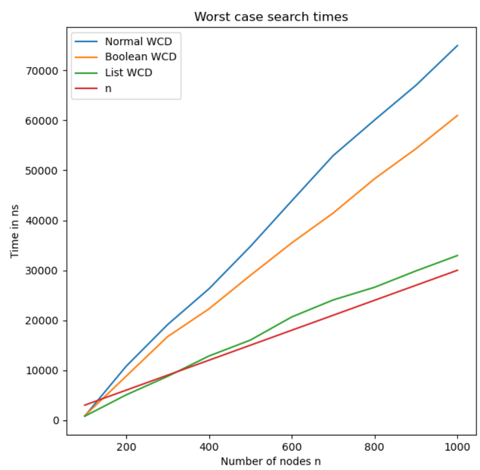

Figura 8: Ricerca su WCD

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 **??** della sezione 4.2. Tutti gli altri metodi ci mettono tempo lineare sempre basato sulla formula **??**.
- 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.

Riferimenti bibliografici

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