Alberi Binari di Ricerca vs Alberi Rosso-Neri

Rebecca Ceccatelli

giugno 2022

1 Introduzione

L'intento di questa relazione è evidenziare le differenze tra gli Alberi Binari di Ricerca e gli Alberi Rosso-Neri. La differenza più significativa su cui qui si è scelto di concentrarsi è l'altezza. Questa, come vedremo, porterà a conseguenze anche sulla complessità temporale delle operazioni legate a tali strutture dati.

2 Nozioni teoriche fondamentali e descrizione del problema

Gli Alberi Binari di Ricerca (abbreviazione: BST) sono alberi binari che, in ogni loro nodo **x**, rispettano la seguente proprietà: il sottoalbero sinistro di **x** contiene solo nodi con chiavi minori della chiave del nodo **x**, e il sottoalbero destro di **x** contiene solo nodi con chiavi maggiori della chiave del nodo **x**. Ogni nodo di un Albero Binario di Ricerca contiene i seguenti attributi: *key, father, left, right, information.*
Alberi Rosso-Neri (abbreviazione: RBT) sono strutture dati aumentate a partire dagli Alberi Binari di Ricerca: sono identici a questi ultimi, se non per il fatto che ogni nodo contiene un attributo aggiuntivo: il colore, *color.*

In generale, l'*altezza* di un albero si definisce come il numero di archi che si trovano nel cammino più lungo dalla radice ad una foglia. Si può dimostrare che le operazioni tipiche su BST e RBT (quali inserimento, ricerca e cancellazione) hanno una complessità temporale $O(h)$.
La grande differenza tra BST e RBT è che, detto n il numero di nodi dell'albero, l'attributo *color* nei nodi di un RBT permette di garantire che la sua altezza sia sempre un $O(\log_2 n)$. Questa proprietà è enunciata nel seguente lemma:

Lemma 2.1: l'altezza massima di un Albero Rosso-Nero con n nodi interni è $2 \log_2 (n + 1)$.

La conseguenza diretta di questa proprietà è che le operazioni tipiche di un RBT richiedono una complessità temporale che non è più un generico $O(h)$, bensì, più precisamente, un $O(\log_2 n)$.

3 Esperimenti

3.1 Scopo degli esperimenti

Lo scopo degli esperimenti che verranno successivamente illustrati è avvalorare la seguente tesi: quando si devono compiere operazioni con complessità temporale *O(h)*, gli Alberi Rosso-Neri sono da preferire agli Alberi Binari di Ricerca perchè i primi sono in grado di offrire prestazioni migliori.
A questo fine, si è scelto di adottare l'inserimento come operazione campione, e di evidenziare il differente comportamento delle due strutture dati quando queste ricevono in input la peggiore configurazione possibile: una serie di chiavi da inserire ordinate in senso crescente.

3.2 Descrizione degli esperimenti e prestazioni attese

Oltre ad alcuni test che verificano il corretto funzionamento e l'affidabilità dei metodi nelle classi che rappresentano le due strutture dati, in questa relazione i test più significativi sono quattro:

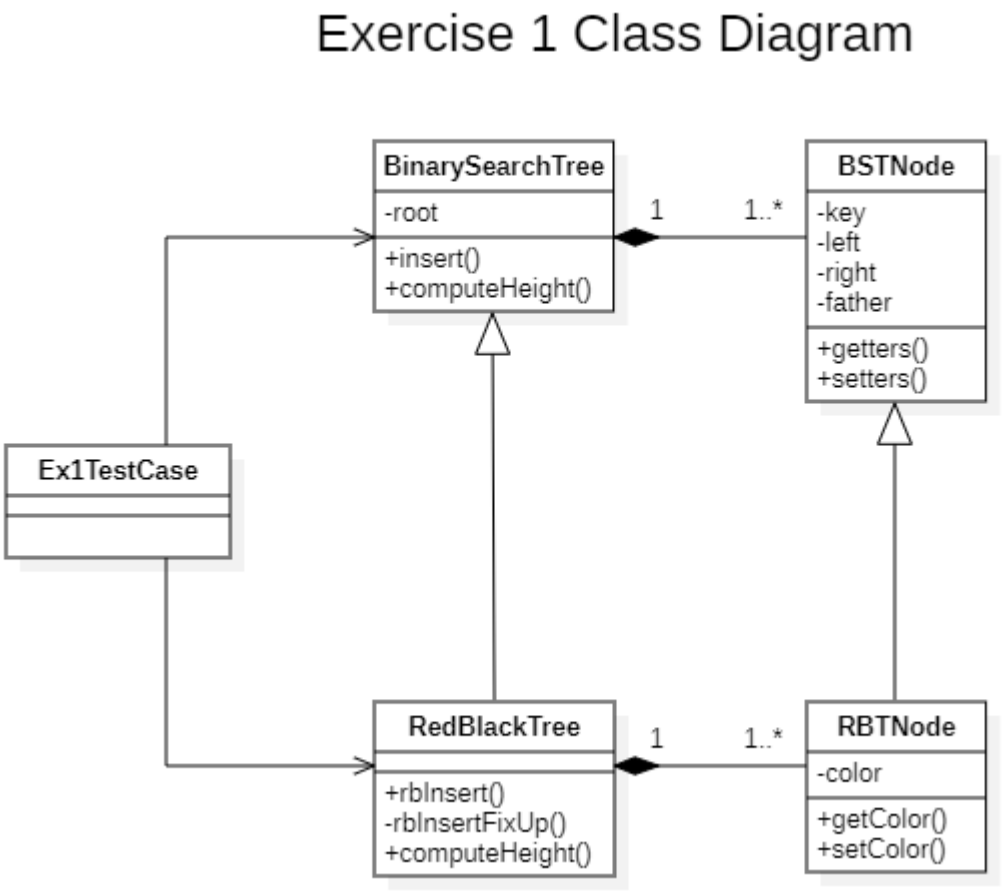
- Il primo test (Test1) vuole mostrare, generando un grafico, qual è l'andamento dell'altezza di un BST nel caso in cui questo riceva in input la peggiore configurazione possibile: una serie di chiavi da inserire ordinate in senso crescente.
  - Prestazione attesa: ci si aspetta che l'andamento dell'altezza sia lineare.
- Il secondo test (Test2) vuole mostrare, generando un grafico, qual è l'andamento dell'altezza di un RBT nel caso in cui questo riceva in input la peggiore configurazione possibile: una serie di chiavi da inserire ordinate in senso crescente.
  - Prestazione attesa: ci si aspetta che l'andamento dell'altezza sia logaritmico.
- Il terzo test (Test3) vuole mettere a confronto, generando un grafico, gli andamenti appena visti dell'altezza di un BST e di un RBT.
  - Prestazione attesa: ci si aspetta che la differenza tra i due andamenti sia evidente nel grafico, e che sia favorevole all'RBT.
- Il quarto test (Test4) vuole mettere a confronto, generando un grafico, la complessità temporale degli algoritmi di inserimento di un BST e di un RBT.
  - Prestazione attesa: ci si aspetta che la complessità temporale dell'algoritmo di inserimento in un BST sia lineare mentre quella del rispettivo algoritmo in un RBT sia logaritmica, e che questa differenza sia evidente nel grafico.

3.3 Sviluppo del codice

Al fine di realizzare gli esperimenti precedentemente descritti è stato sviluppato del codice in Python per implementare le strutture dati interessate e per generare automaticamente un vasto campione di dati da utilizzare per tracciare i grafici tramite il pacchetto matplotlib. La piattaforma di test ha operato su un processore Intel a 2 GHz, su sistema operativo Windows.

4 Documentazione del codice

4.1 Diagramma di classe



4.2 Classi e Unittest

- Classi:
  - **BSTNode**: rappresenta il nodo di un BST. Custodisce gli attributi *key* (un valore numerico), *left, right, father* (altri nodi di tipo BSTNode). Implementa i metodi *getters* e *setters* per tutti gli attributi.
  - **RBTNode**: rappresenta il nodo di un RBT. Eredita dalla classe BSTNode e, in aggiunta, custodisce l'attributo *color*, che è vincolato ad assumere solo due valori ("red" o "black"). Implementa il *getter* e il *setter* per il suo attributo *color*.
  - **BinarySearchTree**: implementa un Albero Binario di Ricerca composto da nodi di tipo BSTNode. Custodisce l'attributo *root* (un nodo di tipo BSTNode). Implementa i seguenti metodi:
    - *setRoot(newNode)::void*: aggiorna il valore di *self.root* a *newNode*.
    - *insertNode(currentNode, newNode)::void*: si occupa del corretto inserimento del nodo nell'albero nel caso in cui il nodo radice sia già stato impostato, scendendo nell'albero fino a trovare la posizione giusta.
    - *insert(newNode)::void*: utilizza i precedenti metodi *setRoot()* e *insertNode()* per gestire il caso generale di inserimento.
    - *findNode(currentNode, key)::Bool, BSTNode*: cerca nell'albero il nodo che custodisce l'attributo *key* e ritorna un booleano che indica l'esito della ricerca insieme ad un nodo di tipo BSTNode, eventualmente nullo se la ricerca è fallita.
    - *recursiveSearch(key)::Bool, BSTNode*: utilizza il precedente metodo *findNode()* per propagare la ricerca lungo tutto l'albero.
    - *inOrder(v)::void*: compie l'attraversamento in-order del sottoalbero con radice nel nodo *v*.
    - *inOrderWalk(v)::void*: utilizza il precedente metodo *inOrder()* per compiere l'attraversamento in-order a partire dalla radice dell'albero.
    - *getMaximum(currentNode)::BSTNode*: ritorna il nodo con *key* massima all'interno del sottoalbero con radice nel nodo *currentNode*.
    - *computeHeight(v)::int*: ritorna l'altezza del sottoalbero con radice nel nodo *v*.
  - **RedBlackTree**: implementa un Albero Rosso-Nero composto da nodi di tipo RBTNode. Eredita dalla classe BinarySearchTree: alcuni metodi vengono lasciati intatti, di altri viene fatto l'override e ne vengono anche definiti di nuovi. Implementa i seguenti metodi:
    - *setRoot(newNode)::void*: viene fatto l'override, qui si setta anche il valore di *self.root.color* a "black" di default.

    - *leftRotate(x)::void*: implementa la rotazione a sinistra nell'albero facendo perno sul nodo x.
    - *rightRotate(x)::void*: implementa la rotazione a destra nell'albero facendo perno sul nodo x.
    - *rbInsertFixup(insertNode)::void*: utilizza i precedenti metodi *leftRotate()* e *rightRotate()* per aggiustare l'albero e garantire così il mantenimento dell'altezza logaritmica anche dopo un nuovo inserimento.
    - *insertNode(currentNode, newNode)::void*: viene fatto l'override per poter gestire anche il caso dei nodi sentinella.
    - *insert(newNode)::void*: viene ereditato, utilizza il precedente metodo *insertNode()* di cui è stato fatto l'override.
    - *rbInsert(newNode)::void*: utilizza il metodo ereditato *insert()* per inserire il nodo *newNode* nell'albero e poi utilizza anche il precedente metodo *rbInsertFixup()* per sistemare eventuali sbilanciamenti.
    - *inOrder(v)::void*: viene fatto l'override per poter gestire anche il caso dei nodi sentinella, che devono essere ignorati.
    - *inOrderWalk(v)::void*: viene fatto l'override per poter utilizzare il precedente metodo inOrder() sovrascritto.
    - *computeHeight(v)::int*: viene fatto l'override per poter gestire anche il caso dei nodi sentinella.

- Unittest, file *Tests*: contiene i tests necessari per verificare il corretto funzionamento delle classi precedenti e per svolgere gli esperimenti.
    - *testComputeBSTHeight()* e *testComputeRBTHeight()*: test preliminari che verificano il corretto funzionamento dei metodi *computeBSTHeight()* e *computeRBTHeight()* su vari alberi, rispettivamente nella classe **BinarySearchTree** e nella classe **RedBlackTree**.
    - *testWorstCaseInBST()*: genera un grafico per verificare l'andamento lineare dell'altezza in un BST che riceve in ingresso la peggiore configurazione possibile di input, i.e. una serie di chiavi da inserire ordinate in senso crescente.
    - *testWorstCaseInRBT()*: genera un grafico per verificare l'andamento lineare dell'altezza in un RBT che riceve in ingresso la peggiore configurazione possibile di input, i.e. una serie di chiavi da inserire ordinate in senso crescente.
    - *testPlotBothHeights()*: genera un grafico che mette a confronto l'andamento delle altezze in un BST e in un RBT che ricevono entrambi lo stesso input dei test precedenti, i.e. una serie di chiavi da inserire ordinate in senso crescente.
    - *testTimeComplexity()*: genera un grafico che mette a confronto la complessità temporale degli algoritmi di inserimento in un BST e in un RBT.

– *computeMovingAverage(array)::[]*: helper method che calcola la media mobile dei dati nel vettore *array*.

5 Presentazione e analisi dei risultati sperimentali

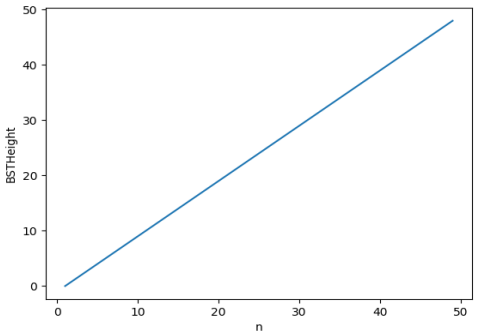


Figure 1: Inserimento di una serie di chiavi crescente: caso peggiore per un BST.

5.1 Inserimento di una serie di chiavi crescente in un BST

Il grafico in *figura 1* conferma l'andamento lineare dell'altezza in un BST a cui è stata data in input la peggiore configurazione possibile.

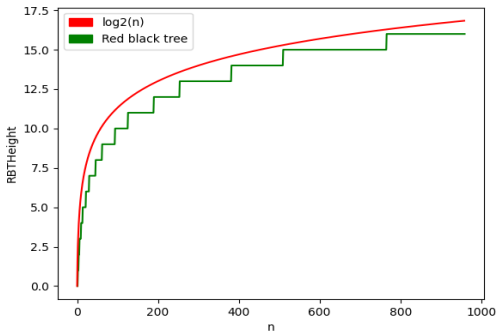


Figure 2: Inserimento di una serie di chiavi crescente in un RBT.

5.2 Inserimento di una serie di chiavi crescente in un RBT

Il grafico in *figura 2* conferma l'andamento logaritmico dell'altezza in un RBT a cui è stata data in input la configurazione peggiore. Due osservazioni significative:

- E' evidente che la funzione $\log_2 n$ (qui moltiplicata per una costante $c \approx 2$) è un limite superiore per l'altezza.
- E' interessante notare che la curva che descrive l'andamento dell'altezza è a scalini: questo accade perchè l'altezza non si incrementa ad ogni inserimento ma, intuitivamente, quando un livello viene significativamente riempito.

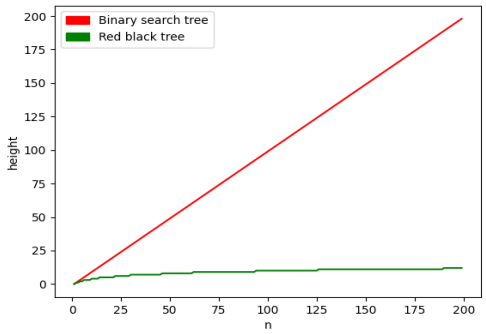


Figure 3: Paragone tra l'altezza in un BST e in un RBT.

5.3 Paragone tra l'altezza in un BST e in un RBT

Il grafico in *figura 3* conferma in maniera evidente il differente andamento dell'altezza in un BST e in un RBT: l'altezza in un RBT si mantiene significativamente più bassa.

5.4 Complessità temporale del metodo *insert()* in un BST e in un RBT

Il grafico in *figura 4* evidenzia che l'andamento della complessità temporale dell'operazione di inserimento in un BST è lineare (così come lo è quello della sua altezza) e che quello in un RBT è logaritmico (così come lo è quello della sua altezza). Si osserva che i picchi e le irregolarità nel grafico sono da interpretare come rumore.

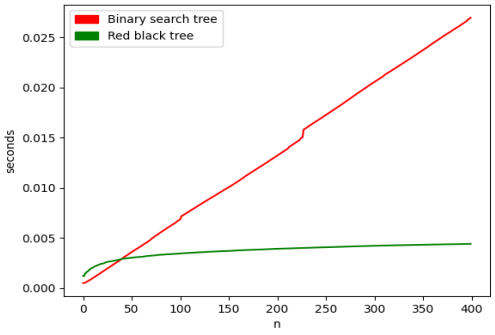


Figure 4: Complessità temporale del metodo *insert()* in un BST e in un RBT.

6 Conclusioni

Gli esperimenti confermano la tesi: gli Alberi Rosso-Neri hanno generalmente altezza minore rispetto agli Alberi Binari di Ricerca e, conseguentemente, per le operazioni con complessità temporale $O(h)$ sono da preferire perchè in grado di offrire prestazioni migliori.