Alberi Di Ricerca

Gennai Gian Maria

2022

1 Introduzione

Gli alberi sono strutture metaforicamente chiamate così perchè partono da una radice e si espandono fino ad arrivare alle foglie, dove il tutto è collegato attraverso dei rami. Noi andremo ad analizzare attraverso dati teorici e speriementali le differenze che troviamo tra *Albero di ricerca binaria* e *Albero rosso-nero*.

2 Albero Binario di Ricerca

L'ABR è formato come tutti gli alberi da una radice, da rami, da nodi intermedi e dalle foglie. Ogni nodo, compresa la radice e ogni foglia, è formato da puntatori che puntano rispettivamente:

- Padre (nodo precedente)
- Figlio destro (nodo di arrivo del ramo destro)
- Figlio sinistro (nodo di arrivo del ramo sinistro)
- Chiave (valore effettivo del nodo)

Dobbiamo precisare che il padre della radice dell'albero è un puntatore a **NIL**(dove Nil mi indica che non c'è altri nodi) e lo stesso vale per i figli destro e sinistro delle foglie. Ogni nodo che ha come figlio destro un nodo con il valore di chiave più grande e un figlio sinistro con il valore di chiave più piccolo rispetto a se stesso.

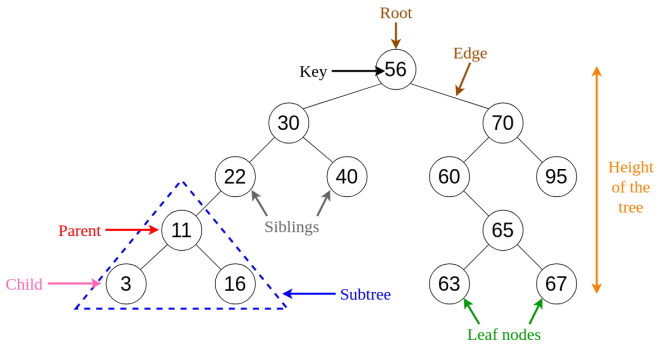

Figura 1: Struttura di un Albero Binario Di Ricerca(Da Google)

2.1 Codice Python per ABR

In allegato a questo documento troviamo un implementazione Python del nostro albero. Nel codice possiamo trovare una classe **Node** che mi inizializza un nodo quindi con i relativi puntatori*(self.key, self.left, self.right)*, dei metodi *get* e *set* per impostare e restituire i valori della chiave e un metodo *getChildren* che restituisce i figli del mio nodo se questi non sono nulli. Dopo di che troviamo la classe **ABR** la quale è quella che invece implementa la struttura dell'intero albero. Al suo interno abbiamo un inizializzazione della radice(*root*) e poi abbiamo vari metodi che utiliziamo per fare le varie operazioni sull'albero quali:

- *insert*: che mi permette di inserire un nuovo nodo all'interno dell'albero tenendo conto di quelli già presenti.
- *find e findNode*: che insieme mi permettono di cercare un nodo e vedere se questo è presente nell'albero. Se l'elemento è presente restituisce **True** in caso contrario, restituisce **False**.
- *inorder*: mi permette di avere in ordine tutti i nodi, per fare questo me ne inserisce uno alla volta dentro un vettore e dopo restituisce il vettore stesso.
- *min e max*: restituiscono il valore minimo e il valore massimo scorrendo o tutto a sinistra per il valore minimo o tutto a destra per il valore massimo.

2.2 Tempo di esecuzione

La struttura di un ABR con n nodi è una struttura che mi permette di fare operazioni di base come *insert, max, min, search* in un tempo pari a O(h) con h che varia da log(n) quando l'albero è ben equilibrato fino ad arrivare ad n quando questo è sviluppato tutto sul ramo destro o sul ramo sinistro. Poi ci sono operazioni come *inorder* che dovendo passare su tutti i nodi una volta ha un tempo fisso pari a O(n). Qui sotto possiamo vedere la differenza che abbiamo tra un aumento lineare ed uno logaritmico(i grafici non vanno sotto zero perchè il tempo non può essere negativo). Tramite il codice Python e grazie alla funzione Timer() è stato possibile calcolare il tempo delle varie operazioni.

3 Albero Rosso-Nero

L'ARN è un albero strutturato in modo molto simile all'ABR con la differenza che ogni nodo ha un attributo in più e troviamo un nodo speciale *Nil*. L'attributo in più che troviamo è *color* il quale ci permette di ideantificare se il nodo è Rosso oppure **Nero**. Il nodo Nil, che sopra abbiamo chiamato "speciale", è perchè questo mi fa da sentinella per tutte le foglie dell'albero, in più il nodo Nil è anche padre della radice dell'albero.
Ogni ARN deve rispettare delle regole:

- Ogni nodo è Rosso o **Nero**.
- La radice è **Nera**.
- Ogni foglia è **Nera**.
- Se un nodo è rosso, allora entrambi i suoi figli sono **Neri**.
- Tutti i cammini da ogni nodo alle foglie contengono lo stesso numero di nodi **Neri**.

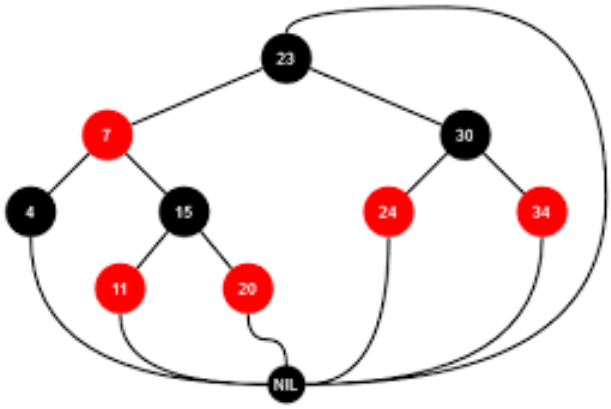

Figura 2: Struttura di un Albero Rosso-Nero(Da Google)

3.1 Codice Python ARN

Allegato a questo file possiamo trovare il codice Python che ci permette di provare il funzionamento dell'ARN e controllare i parametri che ci interessano. L'implementazione mi fornisce una classe che definisce com'è fatto un nodo dell'albero e qui troviamo tutti gli attributi per definire padre(*self.p*), figlio destro e figlio sinistro(*self.right, self.left*), valore del nodo(*self.key*) e colore(*self.color*). Self.color mi restituisce **True** se il nodo è rosso e **False** se il nodo è **nero**. Adesso passiamo alla successiva classe, ovvero quella che mi definisce l'albero e le varie operazioni da fare al suo interno. Come prima cosa è definito il nodo *Nil* e lo mettiamo come padre della radice. I vari metodi che utilizziamo per fare le operazioni sull'albero sono:

- *RBInsert:* Metodo che ci permette di inserire un nuovo nodo all'interno del nostro albero.
- *fixInsert:* Metodo che è chiamato all'interno di *RBInsert* e ci permette di aggiunstare, in base al nuovo nodo inserito, l'albero, in modo da rispettare tutte le regole che sono scritte sopra.
- *RBLeftRotate e RBRightRotate:* Due metodi che si specchiano. Uno ci permette di spostare il figlio destro a sinistra e l'altro ci permette di spostare il figlio sinistro a destra.
- *RBSearch:* Metodo che ci permette di trovare un nodo all'interno dell'albero.
- *inorder*: Metodo che ci permette di inserire in ordine, all'interno di un vettore, tutti gli elementi del mio abero per poi restituirli.
- *min e max*: Due metodi che ci restituiscono il valore minimo e il valore massimo.

Dal codice possiamo notare che è stata utilizzata un implementazione iterativa per tutte le parti di codice che devono ripetersi molte volte e questo principalmente per due motivi; l'implementazione iterativa anche se meno intuitiva, ha un utilizzo minore di risorse e oltre tutto Python dopo un certo numero di iterazioni blocca il procedimento(volendo possiamo aumentare il limite massimo di ricorsioni tramite la libreria *sys* e il metodo *setrecursionlimit()*).

3.2 Tempo di esecuzione

La struttura di un ARN con n nodi è una struttura che mi permette di fare operazioni di base come *insert, max, min, search* in un tempo pari a O(log n). Per l'operazione di inserimento (cancellazione non trattata), il tempo che impieghiamo è pari a O(log n) per RBInsert fino a fixInsert; fixInsert per essere eseguito, al massimo ci vuole un tempo pari a O(log n), perchè il tempo impiegato per fare RBRightRotate e RBLeftRotate al suo interno è pari a O(1).

4 ABR vs ARN

Adesso avendo l'implementazione sia dell'ABR che dell'ARN possiamo chiederci, *qual'è il migliore?*
Per tutti i valori numerici citati nelle prossime sezioni, ho fatto tutti i test su un portatile HP con un processore *AMD A4-9120e* con velocità di clock a 1.5GHz e RAM da 4GB.

4.1 Analisi del codice

Partendo dall'analizzare la struttura del codice notiamo subito che le due strutture sono molto simili. Le differenze partono dal fatto che la struttura dell'ARN si ritrova ad avere delle regole ben precise da rispettare e quindi ogni volta che andiamo ad inserire un nuovo nodo ci troviamo davanti al problema di dover andare a rimettere mano su tutti i nodi precedenetemente inseriti e controllare che effettivamente le varie regole da seguire non siano state violate. Per l'ABR invece l'inserimento non è un problema, il tutto è semplificato dal fatto che i nodi non hanno colore e quindi basta cercare la posizione per l'inserimento confrontando i valori e spostandosi a destra e a sinistra a seconda del valore che abbiamo rispetto a quelli già presenti. Quindi diciamo che il codice scritto per l'ARN è un po' più complesso di quello dell'ABR.

4.2 Analisi del tempo

L'analisi del tempo è fatta su alberi con *10000* nodi con determinati valori; tutte le misurazioni vengono fatte 30 volte e poi viene fatta la media dei valori prima di stampare il grafico.
Per quanto riguarda l'inserimento di elementi all'interno dei due alberi (dove oltre che alla media fatta sulle 30 prove distinte ho anche evitato di prendere ogni singolo tempo di ogni singolo nodo ma ho cercato di livellare un po' il grafico andando a prendere una decina di tempi alla volta e prenderne la media), posso vedere che ho un netto miglioramento per quando riguarda l'inserimento di elementi in ordine all'interno dell'ARN(*Figura 3*), al contrario dell'ABR. Questa differenza è dovuta al fatto che l'ARN riesce tutte le volte ad aggiustare la sua struttura andando a bilanciare la parte destra dell'albero e quella sinistra, cosa che invece mettendo valori in ordine crescente(oppure decrescente) nell'ABR ottengo soltanto un albero che si percorre su tutto il ramo destro (o su tutto il sinistro).
Quando andiamo a confrontare però l'inserimento di un vettore di elementi randomici(*Figura 4*), su entrambi gli alberi, il risultato bene o male non cambia molto e possiamo anche vedere dei miglioramenti da parte dell'ABR. Questo è causato dal fatto che quando anche l'ABR è bilanciato (ovvero per puro caso i valori inseriti mi vanno a bilanciare l'albero da soli), visto che questo non deve fare operazioni del tipo *fixInsert* riesce a risparmiare tempo. Ovviamente come possiamo notare ci sono dei valori all'interno dei grafici che sono al di fuori dei paametri normali ma questo ovviamente era inevitabile ed è dato oltre che alla struttura del codice anche dalle prestazioni del calcolatore.
Per *inorder* alla fine il tempo non è molto significativo, i due tempi sono molto vicini, anche se vediamo che l'ARN ha un tempo maggiore, probabilemte essendo la struttura del metodo ideantica nei due casi, l'aumento del tempo è dovuto al fatto che tutte le assegnazioni nel caso dell'ABR hanno un elemento in meno, il colore (*Figure 5 e 6*).
La cosa si fa interessande andando a vedere i tempi con la ricerca del minimo e del massimo. Per quanto riguarda la ricerca del minimo (*Figura 7*) con i valori non ordinati abbiamo un tempo minore da parte dell'ABR dato probabilemte dal fatto che, come già detto sopra, i valori essendo randomici hanno bilanciato da soli l'albero e in più non hanno, durante l'assegnazione dei valori, da copiare un elemento in più. Lo stesso vale per la ricerca del valore massimo con valori non ordinati(*Figura 8*). Andando però a vedere i tempi per il minimo e il massimo nei valori ordinati risalta subito all'occhio la rapidita con cui l'ARN riesce a trovare il valore massimo(*Figura 9*) in un tempo praticamente nullo rispetto all'ABR e questo perchè i valori sono in ordine crescente e quindi l'ABR non avendoli ordinati in fase di inserimento deve scorrere sul suo unico ramo che è lungo quanto il numero degli elementi inseriti nell'albero. Anche per quanto riguarda la ricerca del valore minimo con i valori ordinati(*Figura 10*), notiamo come al contrario del caso precedente in cui l'ARN riesce ad avere un tempo trascurabile rispetto all'ABR, in questo caso il tempo dell'ARN anche se è maggiore(perchè l'ABR avendo gli elementi ordinati in ordine crescente gli rimane facile trovare il minimo essendo il primo elemento ovvero la radice) non è trascurabile rispetto a quello dell'ABR anche se pratimanete rimane sempre fermo sulla radice.

5 Conclusioni

In conclusione possiamo dire che per quanto riguarda il codice ovviamente l'ABR è più semplice perchè non tiene conto del colore e delle varie regole che ne segueono, invece per quanto riguarda il tempo dobbiamo distinguere il caso in cui ho elementi randomici e quello in cui ho elementi ordinati, come visto sopra delle volte vince uno e delle volte vince l'altro quindi dobbiamo scegliere a seconda delle situazioni anche se effettivamente per non rischiare che anche solo una volta su un milione il tempo mi aumenti in modo sproporzionato posso utilizzare l'ARN.

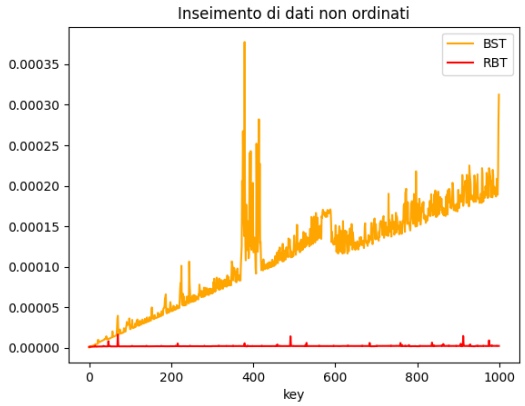

*Figura 3:*Inserimento valori ordinati. ABR, ARN

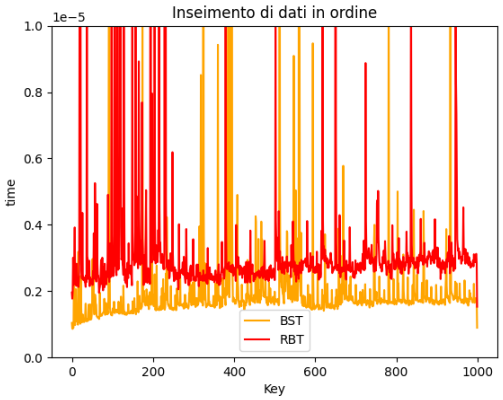

*Figura 4:*Inserimento valori non ordinati. ABR, ARN

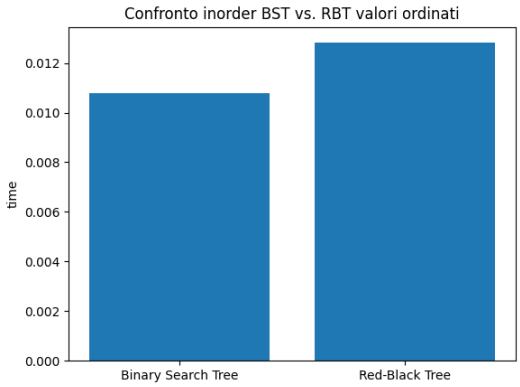

*Figura 5:*Inorder con valori ordinati

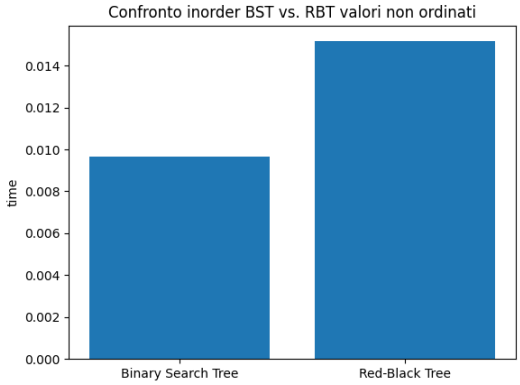

*Figura 6:*Inorder con valori non ordinati

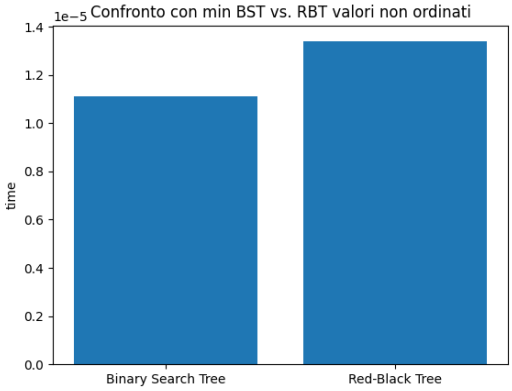

*Figura 7:*Min con valori non ordinati

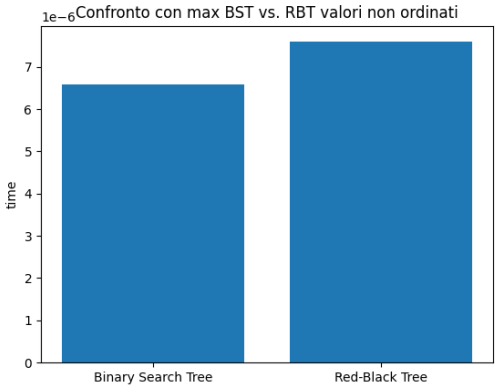

*Figura 8:*Max con valori ordinati

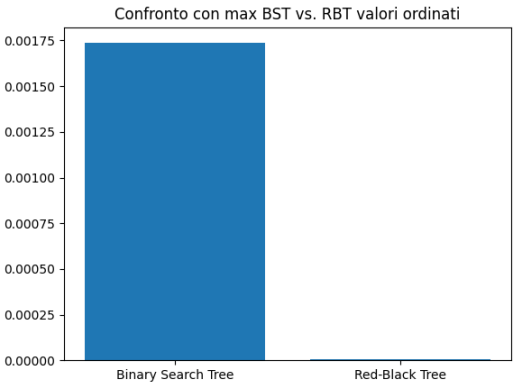

*Figura 9:*Max con valori ordinati

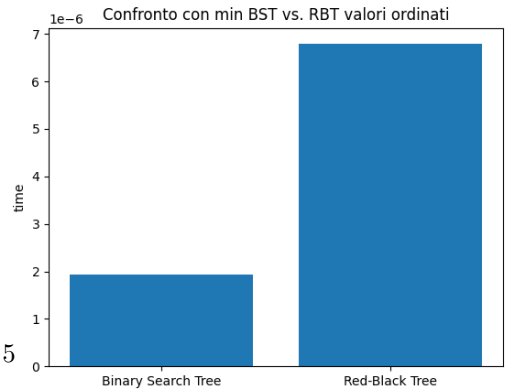

*Figura 10:*Min con valori ordinati