Alberi Binari vs Alberi Rosso-Neri: ricerca ed inserimento

Marco Vignozzi

October 28, 2022

1 Introduzione

In questo esperimento analizzo le prestazioni di Alberi Binari di Ricerca e Alberi Rosso-Neri nell'inserimento di un numero crescente di valori, e nella ricerca su un numero crescente di valori, per poi confrontare i risultati ottenuti attraverso dei grafici.

2 Caratteristiche teoriche

Di seguito introduco le caratteristiche teoriche delle strutture dati e degli algoritmi utilizzati, con riferimento alle prestazioni attese.

2.1 Alberi Binari di Ricerca (ABR)

- **Struttura Dati**

Gli ABR sono degli alberi in cui ogni nodo ha al massimo due figli e tutti i valori nel sottoalbero sinistro di ciascun nodo sono minori o uguali al valore del nodo, mentre tutti quelli nel sottoalbero destro sono maggiori. La struttura degli ABR mantiene il valore del nodo iniziale (radice) e ciascun nodo ha come attributi un valore e tre puntatori, due ai nodi figli (destro e sinistro) e uno al nodo padre.

- **Algoritmi**

Gli algoritmi di Ricerca e Inserimento vengono eseguiti in un Tempo O(h) dove h è l'altezza dell'albero.

2.2 Alberi Rosso-Neri (ARN)

- **Struttura Dati**

Gli ARN hanno la struttura degli ABR ma con l'aggiunta di un attributo: il colore del nodo, rosso o nero. Ogni foglia dell'albero ha come figli dei nodi speciali chiamati NIL. Un ARN deve soddisfare le seguenti proprietà:

1. Ogni nodo è rosso o nero.
2. La radice è nera.
3. Ogni foglia è nera (intesa come foglia terminale NIL).
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.

Grazie a queste proprietà un ARN con n nodi ha la caratteristica di avere un'altezza massima di $2\log_2(n+1)$.

- **Algoritmi**

Gli algoritmi di Ricerca e Inserimento per un ARN con n nodi vengono eseguiti in un tempo $O(\log_2 n)$.

2.3 Prestazioni teoriche

- **Inserimento**

Nel caso migliore l'ABR è perfettamente bilanciato e il tempo di esecuzione dell'algoritmo di Inserimento élo stesso sia per ABR che per ARN, quindi $O(\log_2 n)$.
Nel caso peggiore però l'ABR è equivalente a una lista e per ogni Inserimento si dovrà scorrere tutta la lista, rendendo il tempo di esecuzione un O(n) per l'ABR contro l'$O(\log_2 n)$ dell'ARN.

- **Ricerca**

In questo caso i tempi di esecuzione dipendono da più fattori: la presenza del valore cercato, la sua posizione e dalla struttura assunta dall'albero nel caso di ABR. Nel caso peggiore dovremo scorrere tutto l'albero per trovare (o non trovare) il valore obiettivo. Di conseguenza i tempi saranno gli stessi del caso dell'inserimento sia per ABR che per ARN.

3 Codice

3.1 Moduli

Moduli esterni:

- **random**

- **matplotlib.pyplot**
- **timeit**
- **pathlib**

Moduli propri:

1. **BTclass.py**: in questo modulo è contenuta la struttura dati ABR e tutti i suoi metodi.
2. **RBTclass.py**: qui troviamo struttura e metodi per l'ARN.
3. **test.py**: questo modulo si occupa dello svolgimento dei test e contiene metodi per la generazione dei documenti. Utilizza `BTclass.py`, `RBTclass.py` e tutti i moduli esterni.
4. **main.py**: in questo modulo si creano i test e si richiede di stampare i documenti richiesti. Utilizza il modulo `test.py`.

3.2 Classi

Vengono utilizzate le seguenti classi:

- **Node**: questa classe è presente nei moduli 1 e 2 e contiene la struttura del nodo del relativo albero.
  - Nel nodo dell'ABR non è presente l'attributo *parent* che non è necessario visto che ricerca e inserimento sono implementate in maniera ricorsiva.
  - Nel nodo dell'ARN sono presenti anche gli attributi *parent* e *color*.
- **BTclass**: struttura dati per ABR con metodi per inserimento e ricerca, oltre a un metodo per la stampa. Mantiene il nodo radice nell'attributo *root*.
- **RBTclass**: struttura dati per ARN anch'essa con metodi per inserimento, ricerca e stampa.
- **Test**: questa è la classe principale. Quando viene istanziata genera i test richiesti (vengono eseguiti *(end-start)/step* test per tipo) e li salva nell'attributo *tests*, una lista di tuple dove ciascuna rappresenta un test: (tipo albero, tipo test, elementi cercati/inseriti, tempo impiegato).
  Il costruttore richiede di specificare i seguenti parametri:
  - *t_type*: determina l'ordinamento con cui viene creata la lista di numeri da inserire negli alberi. Può essere *"random"*(default) per ordinamento casuale o qualsiasi altra stringa per ordinamento sequenziale.
  - *r_type*: determina il tipo di ricerca e può essere *"success"*(default) per ricerca con successo o qualsiasi altro valore per ricerca senza successo.

- *start*: valore minimo del numero di elementi cercati/inseriti in un test. 1 è il valore di default.
- *max_value*: valore massimo del numero di elementi cercati/inseriti in un test. 10 è il valore di default.
- *step*: distanza tra il numero di elementi cercati/inseriti in un test e il successivo. 1 è il valore di default.
- *dir_name*: nome della cartella in cui viene salvata la documentazione (a partire dal percorso *./doc*). "test" è il valore di default.

3.3 Metodi

- **Modulo BTclass.py**

`get_name(self)`

Restituisce la stringa *"binary tree"*.

`clear(self)`

Cancella l'albero settando *root* a *None*.

`set_root(self, key)`

Assegna il valore `key` alla radice dell'albero come `Node(key)`.

`insert(self, key)`

Riceve un valore `key` da inserire nell'albero e lo passa al metodo di inserimento insieme al nodo iniziale *root*.

`insert_node(self, current_node, key)`

Metodo di inserimento. Riceve il nodo corrente `current_ node` e il valore `key` da inserire. Trova il posto giusto in maniera ricorsiva e lo inserisce come `Node(key)`.

`find(self, key)`

Passa il valore `key` alla routine di ricerca, oltre all'attributo *root* come nodo iniziale.

`find_node(self, current_node, key)`

Metodo di ricerca. Riceve il nodo corrente `current_node` e un valore `key` e lo cerca nell'albero in maniera ricorsiva, ritornando *True* se il valore è presente in uno dei nodi, altrimenti *False*.

`inorder(self)`

Stampa in ordine crescente i valori contenuti nell'albero.

- **Modulo RBTclass.py**

`get_name(self)`

Ritorna la stringa *"red-black tree"*.

`clear(self)`

Cancella l'albero settando *root* a *None*.

`set_root(self, node)`

Riceve un nodo `node` e lo assegna alla radice *root*.

`insert(self, key)`

Riceve un valore `key` da inserire nell'albero e lo passa al metodo di inserimento come *Node(key)* insieme al nodo iniziale *root*.

`left_rotate(self, x)`

Riceve il nodo `x` intorno al quale far girare l'albero.

`right_rotate(self, x)`

Riceve il nodo `x` intorno al quale far girare l'albero.

`insert_node(self, z)`

Metodo di inserimento. Riceve il nodo `z` da inserire, trova il posto giusto e lo inserisce per poi richiamare il metodo di fixup.

`rb_insert_fixup(self, z)`

Riceve il nodo `z` da cui iniziare il fixup e riordina i colori dei nodi dell'albero in modo che rispettino le regole per l'albero rosso nero.

`find(self, key)`

Passa il valore `key` alla routine di ricerca, passandole *root* come nodo iniziale.

`find_node(self, current_node, key)`

Metodo di ricerca. Riceve il nodo corrente `current_node` e un valore `key` e lo cerca nell'albero in maniera ricorsiva, ritornando *True* se il valore è presente in uno dei nodi, altrimenti *False*.

`inorder(self)`

Stampa in ordine crescente i valori contenuti nell'albero.

- **Modulo test.py**

`random_list(n)`

Riceve un valore `n` intero e restituisce una lista di interi nel range da 0 a `n` ordinati in maniera casuale.

`create_tests_list(op, n_list, start, end, step)`

Lista parametri:

- `op`: una stringa che specifica il test da eseguire (*"find"* per la ricerca con successo e *"insert"* per l'inserimento, o un'altra stringa per la ricerca senza successo).
- `n_list`: una lista di valori che vengono usati per inizializzare gli alberi per la ricerca.
- `start`: valore minimo di elementi cercati/inseriti in un test.
- `end`: valore massimo di elementi cercati/inseriti in un test.
- `step`: differenza tra il numero di elementi cercati/inseriti da un test al successivo.

Questo metodo istanzia gli alberi necessari al test e richiama i metodi `test_insert()` o `test_find()` (a seconda del valore di `op`) un numero *(end-start)/step* di volte, aggiungendo ad ogni iterazione il risultato del test eseguito a una lista di tuple che verrà usata come valore di ritorno. Ogni elemento della lista contiene le seguenti informazioni riguardo a un test eseguito: (tipo albero, tipo test, elementi cercati/inseriti, tempo impiegato).

`test_insert(tree, values_list, number_of_entries, r=0)`

Questo metodo effettua il numero di inserimenti richiesti attraverso il parametro `number_of_entries` nell'albero `tree` prendendo gli elementi dalla `values_list`. Viene calcolato il tempo necessario a inserire tutti gli elementi. Il metodo ritorna una lista di tuple che rappresenta il test eseguito.
Questo metodo è utilizzato anche per creare gli alberi su cui effettuare i test di ricerca. In base al tipo di ricerca che vorremo eseguire passeremo il parametro `r` settato a 0 (ricerca con successo) o a 1 (ricerca senza successo).

`test_find(tree, values_list, number_of_searches, r=0)`

Questo metodo effettua il numero di ricerche richieste dal parametro `number_of_searches` nell'albero `tree` cercando gli elementi nella `values_list`. Il parametro `r` serve in questo caso a definire il tipo di ricerca:

```
for x in range(1, number_of_searches, r+1):
   tree.find(values_list[x])
```

Con `r=0` effettuerà la ricerca con successo, cercando gli stessi elementi della `values_list` che erano stati inseriti.
Con `r=1` effettuerà la ricerca senza successo, cercheremo elementi diversi.
Viene calcolato il tempo necessario a cercare tutti gli elementi. Il metodo ritorna la lista di tuple che rappresenta il test eseguito, con la stessa struttura di quella ritornata da `test_insert()`.

`get_values_list(value_type, op_type, tree_type, tests_list)`

Questo metodo estrae una lista di valori dalla lista di tuple `tests__list` e la ritorna. Il valore estratto è definito attraverso i parametri `value_type`, `op_type` e `tree_type`, che permettono di scegliere: valore da estrarre (elementi cercati/inseriti o tempo impiegato), per quale tipo di operazione (inserimento o ricerca, con o senza successo) e per quale tipo di albero. Utilizza una routine di appoggio `choose_value()`.

`choose_value(test_element, value_type)`

Riceve una tupla `test_element` che rappresenta un test e il tipo di valore da estrarre `value_type`. Ritorna il valore scelto che può essere il tempo impiegato (*"time"*) o il numero di elementi cercati/inseriti in quel test (qualsiasi altra stringa).

`create_plot(self)`

Metodo della classe che serve a generare i grafici degli esperimenti.

`create_docs(self)`

Metodo della classe che genera i file di testo nella cartella specificata al momento della creazione del test.

4 Esperimenti

I seguenti esperimenti sono stati condotti su un computer portatile HP envy con caratteristiche: CPU AMD Ryzen 7 5700U 1.80 GHz, scheda video integrata Radeon Graphics e 16GB di RAM.

1. Inserimento e ricerca con successo di un numero crescente di elementi con lista di valori non ordinata.
2. Inserimento e ricerca senza successo di un numero crescente di elementi con lista di valori non ordinata.
3. Inserimento e ricerca con successo di un numero crescente di elementi con lista di valori ordinata.

4.1 Esperimento 1

4.1.1 Descrizione

In questo esperimento vengono eseguiti 44 test di inserimento e 44 di ricerca, dove vengono inseriti/cercati dai 1000 ai 9800 valori con step di 200. La lista dei valori inseriti negli alberi viene generata all'interno del metodo `random_list(n)` (con *n=9800*) in questo modo:

```
r_list = list(range(n))
random.shuffle(r_list)
```

Vengono costruiti gli alberi con questa lista e poi si eseguono i test di inserimento e di ricerca.
Per ognuno dei 45 test di inserimento viene misurato il tempo all'interno del metodo `test_insert(tree, values_list, number_of_entries, r=0)` per la porzione di codice:

```
start = timer()
for x in range(0, number_of_entries, r+1):
    tree.insert(values_list[x])
end = timer()
```

Per ognuno dei 45 test di ricerca viene misurato il tempo all'interno del metodo `test_find(tree, values_list, number_of_searches, r=0)` per la porzione di codice:

```
start = timer()
for x in range(1, number_of_searches, r+1):
    tree.find(values_list[x])
end = timer()
```

I valori così ottenuti sono usati per creare i grafici 1 e 2 in Figura 1. Qui vengono confrontati i tempi di inserimento per ABR e ARN, e quelli di ricerca.

4.1.2 Prestazioni attese

La lista `values_list` è implementata in python come un array dinamico, quindi la stima asintotica del tempo di accesso a un elemento è O(1). Abbiamo un ciclo *for* all'interno del quale vengono eseguiti `tree.find()` o `tree.insert()` r volte, dove r è il numero di elementi cercati o inseriti in un test ($1000 \text{ ¡ } r \text{ ¡ } n$). Di conseguenza la stima asintotica del tempo di esecuzione della parte di codice misurata è $O(r \log_2 n)$ per l'ARN e $O(r \log_2 h)$ per l'ABR, dove h è l'altezza dell'albero.

4.1.3 Analisi

Il grafico in Figura 1 mostra come ABR e ARN abbiano prestazioni simili nell'inserimento non ordinato di valori con tempi massimi di 0.0329s per ARN, contro i 0.0368s dell'ABR (la presenza di picchi potrebbe essere dovuta a conflitti con il sistema operativo). Nella ricerca si nota invece una maggiore distanza nei tempi di esecuzione, dove l'ARN risulta migliore. L'ARN impiega un tempo massimo di 0.0268s contro i 0.0338s dell'ABR. Questo è dovuto alla presenza di nodi a profondità maggiore nell'ABR, laddove invece l'ARN si mantiene sempre bilanciato.

4.2 Esperimento 2

4.2.1 Descrizione

In questo esperimento si esegue lo stesso numero di test dell'esperimento 1 e sullo stesso numero di elementi, ma si ricercano valori che non sono stati inseriti negli alberi. Vengono effettuate le stesse misurazioni con gli stessi metodi.
I valori così ottenuti servono a creare il grafico 3 in Figura 1 e i grafici in Figura 2. Nel primo il confronto è effettuato sui tempi di ricerca senza successo per ABR e ARN. In Figura 2 si confrontano i tempi di ricerca con e senza successo su una stessa struttura dati.

4.2.2 Prestazioni attese

Le prestazioni attese a livello teorico sono le stesse dell'esperimento 1, ma nel caso peggiore ci aspettiamo qualche peggioramento dovuto al dover sempre percorrere l'albero fino alle foglie.

4.2.3 Analisi risultati

I risultati ottenuti mostrano che i tempi di esecuzione nel caso di ricerca senza successo sono sempre maggiori del caso di ricerca con successo. Questo risultato non è ovvio per l'ABR, dove la durata della ricerca dipende dall'altezza del sottoalbero ispezionato. Anche in questo caso l'ABR ha prestazioni peggiori con un tempo massimo di 0.0388s contro i 0.0310s dell'ARN. Anche stavolta (come nell'esperimento 1) la discrepanza è dovuta alla maggiore profondità delle foglie rispetto all'ARN.

4.3 Esperimento 3

4.3.1 Descrizione

In questo esperimento vengono eseguiti 79 test di inserimento e 79 di ricerca, dove vengono inseriti/cercati da 1 a 791 valori con step di 10. La lista dei valori inseriti negli alberi viene generata direttamente nel costruttore della classe `Test` in questo modo:

```
numbers_list = list(range(max_value))
```

dove `max_value=800`. Vengono effettuate le stesse misurazioni con gli stessi metodi usati negli esperimenti 1 e 2. I valori ottenuti servono a creare i grafici in figura 3 che mostrano il confronto tra le due strutture sui tempi di esecuzione di inserimento e ricerca.
NOTA: ho scelto un valore più basso di elementi da inserire visto che con un numero troppo elevato il programma non veniva eseguito, perchè si effettuavano una quantità eccessiva di chiamate ricorsive ai metodi `find()` e `insert()` dell'ABR.

4.3.2 Prestazioni attese

Le prestazioni attese per l'ARN sono le stesse dell'esperimento 1 ovvero $O(n \log_2 n)$, con n numero elementi inseriti. Per quanto riguarda l'ABR stavolta l'inserimento è ordinato, di conseguenza ci si aspetta un tempo di esecuzione $O(n^2)$.

4.3.3 Analisi risultati

Questo esperimento evidenzia i limiti dell'ABR negli algoritmi di ricerca e inserimento. Infatti i tempi peggiorano notevolmente (rispetto all'esperimento 1) per l'ABR per cui si riesce ad apprezzare l'andamento esponenziale dei grafici in Figura 3. L'ARN mantiene invece un andamento pressochè inalterato rispetto agli altri esperimenti. I tempi massimi sono molto più distanti: per l'inserimento otteniamo 0.0029s per l'ARN contro i 0.0619s dell'ABR, per la ricerca 0.0015s per l'ARN contro i 0.0689s dell'ABR.

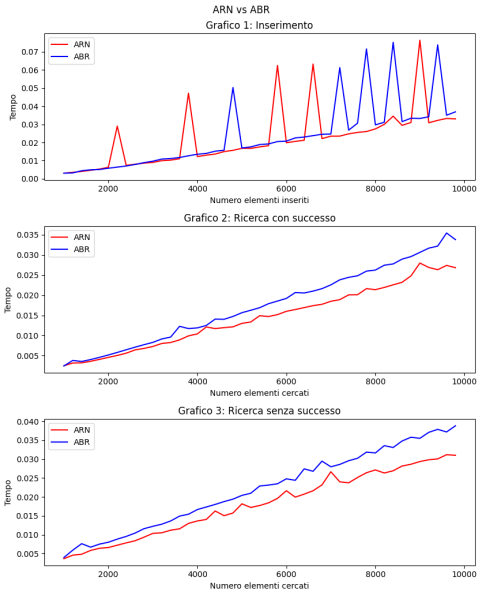

Figure 1: questi grafici mostrano i risultati del confronto tra ARN e ABR per quanto riguarda i tempi di esecuzione

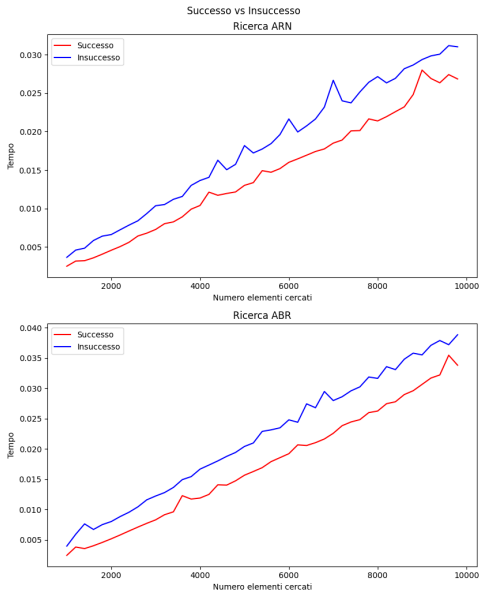

Figure 2: questi grafici mostrano i risultati del confronto tra ricerca con successo e senza successo per uno stesso albero

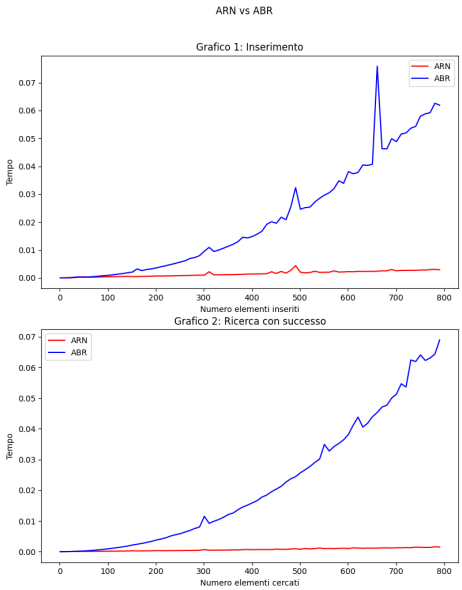

Figure 3: questi grafici mostrano i risultati del confronto tra ricerca con successo e senza successo per uno stesso albero

5 Conclusioni

I risultati ottenuti in questi esperimenti sono fedeli a quelli teorizzati e mostrano come l'ARN sia decisamente migliore nei casi in cui si richiede inserimento di valori ordinati. L'ABR dal canto suo presenta una struttura più semplice, ma resta inferiore all'ARN anche nel caso in cui i valori inseriti non siano ordinati. Una struttura ad ABR risulta quindi indicata solo nei casi in cui non ci sia bisogno di tanti nodi e soprattutto non si effettuino inserimenti ordinati, mentre l'ABR è sempre la scelta migliore.

6 Dati utilizzati

I dati utilizzati negli esperimenti sono presenti nei file di testo allegati. Essi contengono lista di valori utilizzata per poter duplicare l'esperimento e i tempi calcolati per ogni test.