Alberi Binari vs Alberi Rosso-Neri: ricerca ed inserimento

Marco Vignozzi

October 26, 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.

- **Algoritmi**

Gli algoritmi di Ricerca e Inserimento possono essere entrambi 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. 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 attese

- **Inserimento**

Nel caso migliore l'ABR è perfettamente bilanciato e il tempo di esecuzione dell'algoritmo di Inserimento dovrebbe essere pressochè lo 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 dalla posizione del valore cercato, oltre che dalla struttura assunta dall'albero nel caso di ABR. Mettendoci nel caso peggiore della prima condizione avremo che l'elemento cercato si troverà alla massima profondità dell'albero. Di conseguenza i tempi saranno gli stessi del caso dell'inserimento sia per ABR che per ARN.

3 Esperimenti

Per testare le diverse prestazioni di queste strutture dati ho eseguito i seguenti esperimenti:

- Inserimento di un numero crescente di elementi
- Ricerca con successo
- Ricerca senza successo

Per ogni esperimento è possibile scegliere un range di valori da inserire o cercare con un certo step e se questi valori devono essere inseriti in ordine crescente oppure no. Ogni esperimento produrrà un grafico tempo/numero di elementi(cercati o inseriti) sia per l'inserimento, sia per la ricerca. Nel terzo esperimento produrrà inoltre un secondo grafico dove si confronta ciascuna struttura con se stessa per quanto riguarda le prestazioni di ricerca con successo e senza successo.

4 Codice

4.1 Moduli

Nel programma di test sono presenti i seguenti moduli:

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 e RBTclass.py.
4. **main.py**: in questo modulo si creano i test e si richiede di stampare i documenti richiesti. Utilizza il modulo test.py.

4.2 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 era necessario visto che ho implementato sia la ricerca che l'inserimento con la ricorsione.
  - Nel nodo dell'ARN ho aggiunto gli attributi parent e color.
- **BTclass**: struttra dati per ABR con metodi per inserimento e ricerca, oltre a 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*, che è una lista di tuple che rappresentano 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.

4.3 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 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, passandole *root* come nodo iniziale.

`find_node(self, current_node, key)`

Metodo di ricerca. Riceve il nodo corrente 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 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 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 tra un test e il 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 ogni volta il test a una lista di tuple che verrà usata come valore di ritorno. Ogni elemento della lista contiene quindi 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 i *number_of_entries* 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* inseriti in precedenza. 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 avremo inserito.
Con r=1 effettuerà la ricerca senza successo, cercheremo elementi diversi.
Viene calcolato il tempo necessario a inserire tutti i *number_of_entries* elementi.
Il metodo ritorna una lista di tuple che rappresenta il test eseguito.

`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 che valore della tupla estrarre, 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 creare i grafici degli esperimenti.

`create_docs(self)`

Crea i file di testo nella cartella specificata al momento della creazione del test.