Commento dei dati sperimentali

Gli esperimenti sono stati effettuati con varie distanze tra gli elementi. Per distanza di intende quanti numeri ci sono tra una chiave e la sua successiva (presi ordinati).

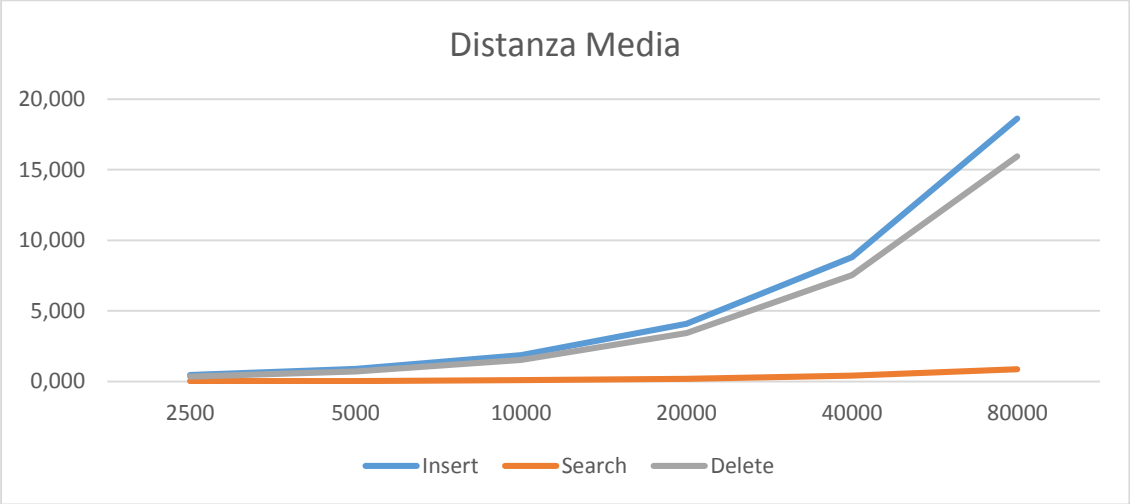
Triple casuali

Prese delle triple casuali, nel momento in cui andiamo ad inserire elementi all'interno dell'array (sia se questi abbiano distanza molto piccola o molto grande), i dati sperimentali ci permettono di affermare che all'aumentare del numero di elementi, il tempo per poterli inserire aumenta notevolmente. Questo è dovuto al fatto che, avendo iniziato i test con 2500 numeri da inserire, all'interno dell'array si andranno per forza a creare degli avl e in questo modo, ogni volta che inseriamo un elemento (O(1)), potrebbe essere necessario dover bilanciare tutto quanto l'albero. All'aumentare degli elementi, il bilanciamento richiederà sempre più tempo, in quanto la struttura da dover bilanciare sarà più complessa. Tutto ciò può esser confermato dai dati sperimentali ottenuti.

Per quanto riguarda il search invece, le cose sono molto diverse, poiché la struttura degli avl è predisposta in modo tale che la ricerca richieda al più un tempo O(logn) e infatti risulta essere veloce anche con un numero considerevole di elementi.

Possiamo notare che anche il delete è un operazione molto costosa, e richiede all'incirca lo stesso tempo che impiega l'insert a parità di elementi

Dal grafico della media possiamo notare che l'insert e il delete hanno un andamento asintotico molto simile(leggermente più lento l'insert), mentre il search risulta essere molto prestante.

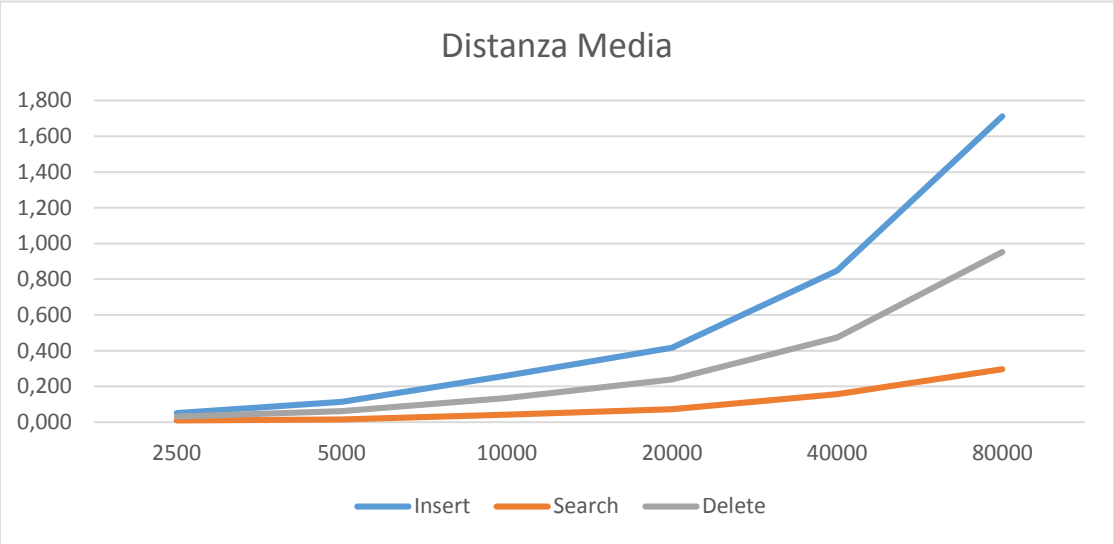


Triple oriented

Vedere il file allegato per la spiegazione dettagliata dell'algoritmo usato.

Usando una tripla scelta su misura in base alle key che dobbiamo inserire, è stato possibile ottimizzare il codice, riuscendo a dividere in modo equo tra tutte le celle dell'array gli elementi inseriti senza andare ad usare la struttuta dati degli avl. In questo modo gli elementi per cella saranno al più 5, e l'insert, il search e il delete risulteranno essere molto più rapidi.

Solamente l'insert e il delete risulteranno non pienamente ottimizzati quando la distanza tra le chiavi è 1, in quanto, per i dati imposti dalla traccia, si verranno a creare avl da 7 elementi, per cui si "perde" del tempo a creare la struttura dati avl a partire dalle liste per quanto riguarda l'insert e il delete. Il search però risulta essere comunque molto veloce, in quanto è una funzione ottimizzata per questo tipo di struttura dati.

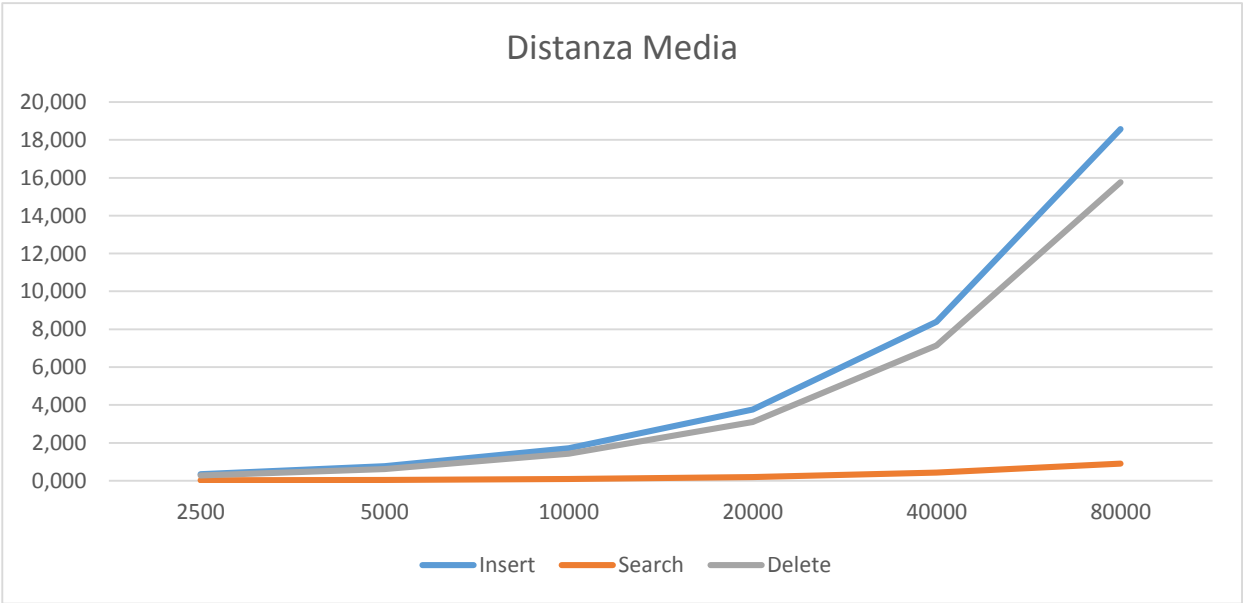


Tripla con D grande

Per questo esperimento è stata presa una tripla con un d molto grande, in questo modo l'array di appoggio risulterà avere un gran numero di celle a disposizione per gli elementi e le relative strutture dati.

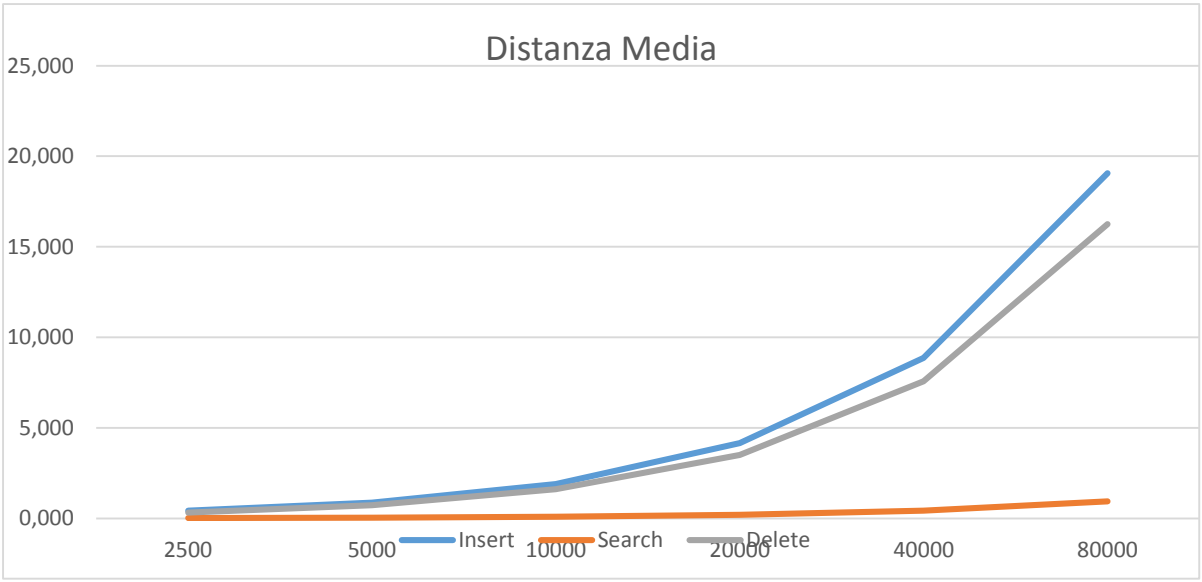
Il comportamento risulta essere molto simile a quello delle triple casuali, infatti, anche se abbiamo un maggior numero di celle per inserire gli elementi, queste non vengono sfruttate a pieno, poiché gli elementi vengono inseriti in base al valore della loro key nel corrispondente insieme.

Come per le triple casuali l'insert e il delete risulteranno più lenti, in quanto alla fine di queste operazioni potrebbe essere richiesto il bilanciamento, cosa che invece non avviene per il search, che negli avl avviene nel tempo peggiore in O(logn).



Tripla con D piccolo

In questo caso invece, il numero di celle a disposizione risulta essere il più piccolo possibile e nel momento in cui andiamo ad inserire le key all'interno del nostro array, si verranno a creare pochi avl, ma con un gran numero di elementi, quindi l'insert e il delete risulteranno essere più lenti in quanto ad ogni operazione può essere richiesto il bilanciamento della struttura dati. Per lo stesso ragionamento delle tripla con D grandi il search risulta essere più rapido.



Dizionario

Dopo aver effettuato i nostri test, il dizionario di python risulta essere più veloce di qualunque altro nostro test fatto, in quanto non richiede l'ausilio di strutture complicate per operazioni che, usando il dictionary, richiedono O(1).