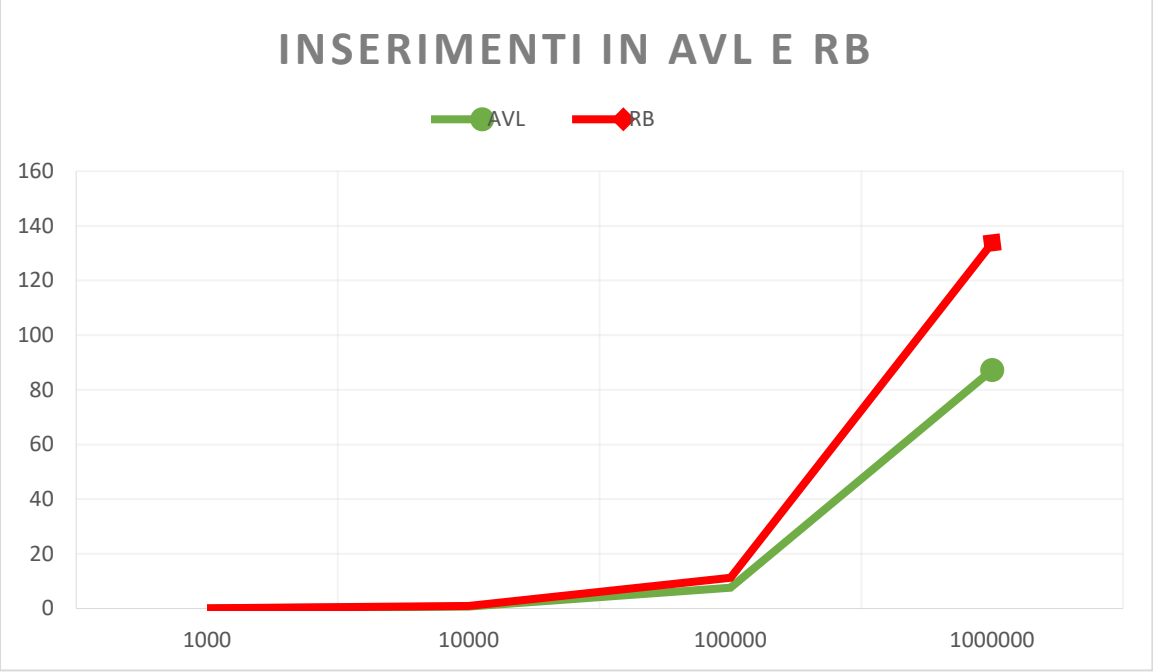
Progetto #2

Nell'implementare il primo esercizio relativo al secondo progetto, abbiamo creato un modulo verifica1.

In questo modulo vi è la funzione test_trees_insertion, la quale, preso in input il numero di elementi, genera una sequenza casuale di n parole, di lunghezza 10, queste poi sono inserite, secondo chiavi ordinate da 1 a n, rispettivamente, in un albero AVL e RB.

Viene poi stampato, il tempo totale dovuto agli inserimenti nell'albero AVL e quello RB, inoltre, viene stampato il rapporto del tempo impiegato dall'albero RB e quello AVL.

In un campione causale di rispettivamente 1000, 10000, 100000 e 1000000 elementi ordinati otteniamo i seguenti risultati:



n. elementi	1000	10000	100000	1000000
Tempo AVL	0.047	0.657	7.619	87.248
Tempo RB	0.085	0.93	11.275	133.778
Rapporto dei tempi	1.79	1.42	1.48	1.53

Entrambi gli algoritmi di inserimento hanno complessità temporale $O(\log n)$.

Va considerato però, che mentre l'altezza nel caso peggiore di un AVL è $\sim 1.44\ logn$, quella di un RB è $2logn$.

In un inserimento prima dobbiamo cercare il nodo dove inserire la chiave, e il tempo che questa operazione prende dipende dall'altezza dell'albero, poi aggiungiamo il nodo e poi applichiamo operazioni di bilanciamento, le quali ancora una volta richiedono un tempo dipendente dall'altezza dell'albero.

Questo è il motivo per cui, come notato dai precedenti grafici, all'aumentare degli elementi, la differenza di tempo negli inserimenti "si fa sentire".

La ricerca negli AVL, è certamente molto veloce, mentre nei RB la ricerca è meno veloce a causa dell'altezza dell'albero.

Per poter progettare MyTreeMap affinché after e before richiedano complessità temporale $O(1)$ senza andare ad intaccare quella degli altri metodi è stato necessario aggiungere informazioni aggiuntive nei nodi.

Nella classe _Node abbiamo aggiunto due attributi: successor e predecessor.

Abbiamo poi fatto l'override di after e before, facendogli restituire la position relativa al nodo puntato dai due attributi successor e predecessor.

Al fine di rendere compatibile, la nuova classe, così progettata, con la possibilità di restituire predecessore e successore è stato necessario modificare tutti i metodi che consentono l'update dell'albero, che sono i seguenti: _add_left, _add_right, _delete e _attach.

Questi sono gli unici metodi derivati dalla classe, LinkedBinaryTree che consentono di modificare l'albero, cancellare o aggiungere nodi, e sono stati cambiati in maniera tale da aggiornare sempre successore e predecessore a ogni modifica dell'albero.

In particolare, per poter modificare _attach e garantire la stessa complessità temporale che aveva prima, abbiamo aggiunto alla classe due

attributi: _max e _min, questi contengono rispettivamente il nodo con chiave massima e nodo con chiave minima.

Per quanto riguarda la seconda parte dell'esercizio e cioè implementare MyAVLTreeMap, garantendo le stesse complessità temporali di AVLTreeMap ed inoltre il corretto funzionamento di after e before, è stato necessario derivare MyTreeMap e AVLTreeMap nell'ordine corretto e implementarne i costruttori.

Abbiamo fatto anche l'override di __str__ in modo da poter usare il modulo drawtree, per stampare gli alberi in una maniera leggibile.

Per poter stampare gli alberi rosso neri e quindi avere visibili a schermo nodi rosso e neri, abbiamo dovuto modificare il modulo drawtree ed inoltre fare l'override di __str__ ad hoc.

Per implementare invece il quarto esercizio, il quale, richiedeva dato un albero binario generico di trovare il lowest common ancestor, abbiamo creato il metodo LCA.

Il metodo LCA, partendo dalla radice, confronta questa con la chiave delle due position, se la radice è maggiore di entrambe si scende a destra, se invece è minore di entrambe a sinistra.

Questo garantisce complessità $O(h)$, dal momento che nel caso peggiore dobbiamo scendere attraverso un numero di nodi pari all'altezza dell'albero. Il caso peggiore si verifica se LCA(p,p) e p è una foglia, con altezza massima.

Al fine di poter implementare il quarto esercizio è stato necessario aggiungere alla classe _Node due attributi: _left_size e _right size, questo ci permette quando effettuiamo la split di non ricalcolare le dimensioni dei due alberi che si vengono a creare.

A seguire è stato dunque necessario modificare la __delitem__al fine di aggiornare _left_size e right_size, stessa cosa per la _rotate.

In un albero red and black possiamo fare: rotazioni, ristrutturazioni e ricolorazioni.

Mentre le ricolorazioni non modificano _left_size e _right_size, le ristrutturazioni e rotazioni sì. Poiché le ristrutturazioni non sono altro che

due rotazioni, è stato dunque necessario modificare _rotate per aggiornare _left_size e _right_size.

Abbiamo a sua volta modificato la _rebalance_insert affinchè a seguito di ribilanciamento dovuto a un inserimento si aggiornassero ancora una volta la _left_size e _right_size.

È stato necessario creare due metodi di ausilio alla fusion e alla split, uno _black_depth il quale trova la profondità nera dell'albero e la restituisce questo metodo ha complessità proporzionale all'altezza dell'albero $O(logn)$.

Il secondo è _find_black_parent, il quale specificata una certa black depth come input, trova il nodo che si trova a quella profondità nera, il metodo può avere complessità $O(logn)$.

Il metodo _update_sizes, specificata un nodo e una position e una size in ingresso risale fino alla radice aggiornando i suoi parent con la corretta size quindi al più ha complessità $O(logn)$.

Nel quarto esercizio, abbiamo implementato prima la fusion, questo perché per poter fare la split dell'albero, è possibile usare la fusion.

Nella fusion, preso un albero in input, si ricerca prima l'elemento più grande dell'albero su cui è eseguito il metodo, questo ha complessità $O(logn)$.

Ricerchiamo poi il più piccolo elemento dell'albero preso in input che ai fini della trattazione denominiamo t1. La complessità di questo metodo è: $O(logm)$.

Dopo aver confrontato dunque il minimo di t1 col massimo di self siamo in grado di stabilire se l'albero passato ha chiavi maggiori dell'albero su cui stiamo eseguendo la fusion e in caso contrario lanciare la relativa eccezione.

Si calcola ora la profondità nera dei due alberi e questo ha complessità ancora una volta $O(logn) + O(logm)$.

Si calcola la dimensione di entrambi gli alberi e dell'albero risultante e questo ha complessità $O(1)$. Si elimina ora dall'albero t1 il nodo la cui chiave è minima, questo ha al più complessità $O(logm)$.

Ora possono verificarsi 2 casi:

1) Nel primo caso in cui abbiamo stessa black depth, il nodo che è stato eliminato da t1 lo si rende radice del nuovo albero, lo si colora di nero e a sinistra e a destra vi si collegano i due alberi precedenti.
2) Gli alberi hanno diversa black depth, a seconda di quale albero ha profondità nera maggiore, si trova il nodo al di sotto del quale abbiamo stessa black depth, si aggiunge opportunamente il minimo di t1, lo si colora di rosso e se necessario si ribilancia.

Il caso peggiore a livello di complessità temporale è il secondo, potrebbe essere necessario, a partire dal punto in cui gli alberi sono stati fusi, ribilanciare verso la radice, questo a seconda che il primo albero abbia profondità nera maggiore o minore può valere $O(logn)$ oppure $O(logm)$. Oltre al ribilanciamento è necessario anche aggiornare _left_size e _right_size che ancora una volta può avere complessità $O(logn)$ o $O(logm)$.

In definitiva la fusion ha complessità: $O(logn) + O(logm)$.

La split innanzitutto ricerca la position relativa alla chiave specificata, questo ha complessità O(logn), si lancia un errore se la position non esiste.

A questo punto si creano due alberi, che contengono il sottoalbero sinistro della position cercata e il sottoalbero destro.

A partire dalla position si risale verso la radice, e a seconda che si risale a un parente da destra o da sinistra, effettuiamo una fusion, usando uno dei due sottoalberi, prima istanziati, col sottoalbero destro/sinistro del nodo parente.

La split ha dunque complessità: $O(logn)$.