Laboratorio di ASD - seconda parte

Bazzana Lorenzo, 147569
bazzana.lorenzo@spes.uniud.it

D'Ambrosi Denis, 147681
dambrosi.denis@spes.uniud.it

Zanolin Lorenzo, 148199
zanolin.lorenzo@spes.uniud.it

30 maggio 2021

1 Introduzione

Richiesta E' stata richiesta l'implementazione delle operazioni di *find()* e *insert()* per tre tipi diversi di alberi binari di ricerca: i *BST*, gli *RBT* e gli *AVL*. Gli ultimi due sono alberi bilanciati, ossia adottano algoritmi aggiuntivi volti a mantenere una struttura ottimale per l'esecuzione di operazioni future.

L'analisi dell'efficacia di queste strutture dati riguardo alle due operazioni di ricerca e inserimento viene effettuata attraverso la misurazione dei tempi medi e ammortizzati per l'esecuzione di n operazioni di *find()* e m operazioni di *insert()*, con $m \leq n$.

2 Presentazione degli algoritmi

2.1 Descrizione delle strutture dati

Binary Search Tree: Albero binario in cui ogni nodo contiene una chiave intera e due puntatori *left* e *right* ai figli. L'invariante dei *BST* consiste nel fatto che per ogni nodo x appartenente all'albero tutte le chiavi che si trovano nel sottoalbero sinistro di x sono minori di x e tutte le chiavi nel sottoalbero destro sono maggiori di x, ossia:

$$\forall \text{ nodo } x, \forall y \in left(x), \forall z \in right(x) \Longrightarrow y.key < x.key < z.key$$

dove $left(x)$ indica il sottoalbero radicato nel figlio sinistro di x e $right(x)$ il sottoalbero radicato nel figlio destro. In particolare, nell'implementazione le foglie sono nodi che hanno entrambi i puntatori ai figli $NULL$.

AVL Tree: Albero BST in cui ogni nodo x è dotato del campo aggiuntivo *height*, che rappresenta l'altezza del nodo stesso, ossia la lunghezza del cammino più lungo x-foglia. L'invariante degli AVL che garantisce il bilanciamento del peso consiste nella proprietà che per ogni nodo x appartenente all'albero le altezze dei sotto-alberi di sinistra e di destra differiscono al più di 1, ovvero

$$\forall \text{ nodo } x,\ \left|height(left(x)) - height(right(x))\right| \leq 1$$

Grazie a questo accorgimento, l'altezza di un albero AVL è $\Theta(\log n)$, con n numero di nodi nell'albero.

Red Black Tree: Albero BST in cui ogni nodo ha associato un colore: rosso o nero. I colori sono fondamentali per definire una nuova grandezza necessaria per il bilanciamento: l'*altezza nera*, che consiste nel numero di nodi neri nel cammino tra x (escluso) e le foglie (incluse). Le proprietà degli RBT che permettono di mantenere l'altezza della struttura dati logaritmica rispetto al numero di nodi sono:

- Ogni nodo è rosso o nero
- La radice è nera
- Ogni foglia è nera
- Se un nodo è rosso, entrambi i suoi figli sono neri
- Tutti i discendenti di un nodo presentano la stessa altezza nera

Quindi l'altezza di un albero RBT è $\Theta(\log n)$, n numero di nodi nell'albero. Vale inoltre che

$$bh(T) \leq h(T) \leq 2bh(T)$$

dove $bh(x)$ rappresenta l'*altezza nera* del nodo x.

2.2 Inizializzazione dei nodi

La scelta implementativa presa per la generazione dei nodi è la seguente: si è deciso di istanziare un vettore che conterrà i nodi dei vari alberi, in cui le chiavi sono inizializzate tramite uno dei metodi descritti sotto. Gli altri campi inizializzati con queste procedure sono:

- *left, right, parent* a *NULL*;
- *color* a *red* per i RBT;
- *height* a 1 per gli alberi AVL.

L'inizializzazione preventiva dei nodi serve ad evitare operazioni inutili durante la misurazione dei tempi del programma, quali la gestione dell'eliminazione degli alberi e l'allocazione dinamica della memoria, che andrebbero a influenzare negativamente le performance. Successivamente, durante l'inserimento del nodo k si confronta semplicemente la sua chiave con quelle presenti nell'albero; una volta trovato il punto in cui inserirlo si imposta il puntatore (destro o sinistro) del genitore al nodo k.

Random Initialise (Caso medio): Fornito un array di n nodi già allocati, questa procedura li inizializza con chiavi pseudocasuali comprese tra 0 e $RANDMAX$ generate con la funzione della libreria standard C $rand()$. La sequenza numerica così creata avrà pochi valori duplicati per n piccoli rispetto a $RANDMAX$; di conseguenza, le procedure utilizzate per la misurazioni di tempi eseguiranno n operazioni $find()$ e un numero molto vicino a n di $insert()$.

Linear Initialise (Caso peggiore): Rispettate le stesse precondizioni della procedura precedente, *Linear_Initialise()* istanzia i nodi dell'array fornito con chiavi in ordine monotona crescente. In tal modo, le operazioni di $insert()$ fanno degenerare gli alberi BST in liste concatenate, con una conseguente degradazione delle performance di ricerca/inserimento da logaritmiche a lineari. Al contrario, le euristiche messe in pratico dagli alberi AVL e RBT permettono di garantirne il bilanciamento. Di conseguenza, questa procedura è utile per studiare le prestazioni di queste strutture dati nel caso peggiore.

Balanced Operations (Caso migliore): Contrariamente alla funzione precedente, questo algoritmo istanzia i nodi dell'array con chiavi "alternate", estraendo la mediana di intervalli sequenzialmente più piccoli fino ad arrivare a singoli valori. In tal modo, i BST rimangono bilanciati durante gli inserimenti ed è possibile valutare il costo aggiuntivo dei ri-bilanciamenti di AVL e RBT. Questa procedura, di conseguenza, è utile per studiare le prestazioni dei vari tipi di alberi nel caso migliore.

3 Misurazione dei tempi di esecuzione

Il tempo ammortizzato T_{amm} di un insieme di n operazioni è definito come il rapporto tra il tempo totale necessario per eseguire tutte le operazioni e la loro numerosità n: di conseguenza, T_{amm} rappresenta il tempo necessario a eseguire mediamente una singola operazione.

Il calcolo del tempo medio e ammortizzato per le operazioni di ricerca e inserimento viene effettuato tenendo conto del numero di nodi e dell'errore commesso: in particolare, per ogni dato numero di nodi n, l'algoritmo implementato ripete n operazioni di ricerca e m operazioni di inserimento, $m \leq n$ un numero di volte tale da assicurare un errore massimo relativo pari a $\epsilon_{max} = 0.01$, garantito da un tempo totale maggiore o uguale a T_{min}, calcolato come

$$T_{min} = R(\frac{1}{\epsilon} + 1)$$

in cui R è la risoluzione del clock.

Poiché per raggiungere l'errore massimo ϵ_{max} basta una sola iterazione (per la maggior parte dei valori di n), si è deciso di imporre un numero minimo di 20 ripetizioni del ciclo, al fine di ottenere tempi meno dipendenti dalla sequenza di chiavi generate. Per questo motivo si determina il tempo medio necessario a

ricercare e inserire n nodi come:

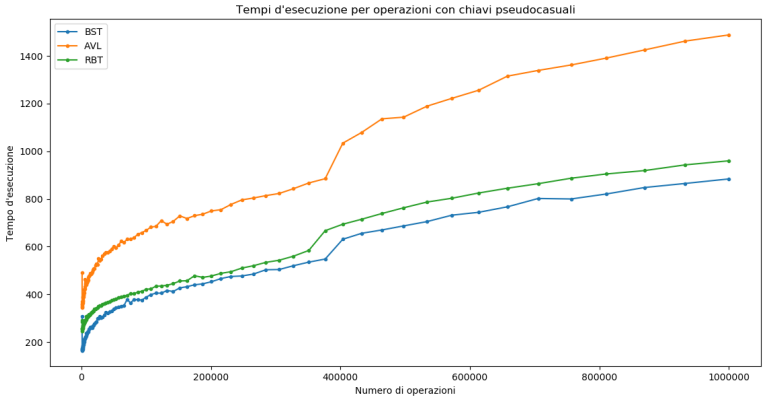
$$T_{medio} = \frac{tempo\ totale}{k}$$

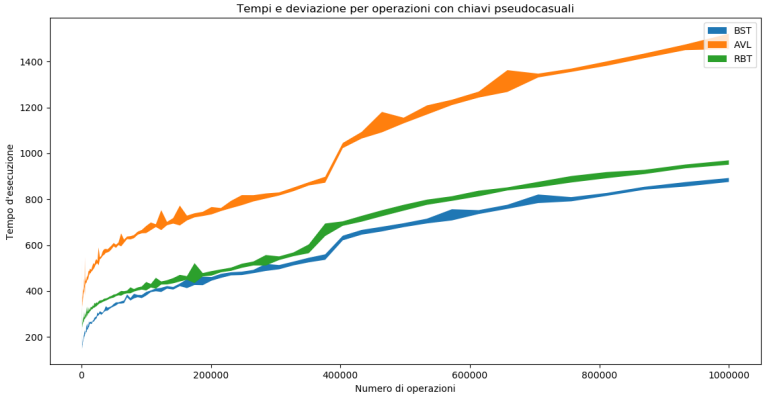
Dove $k \geq 20$ [1]. Il tempo ammortizzato viene poi calcolato dividendo il tempo medio per il numero di inserimenti:

$$T_{amm} = \frac{tempo\ totale}{n}$$

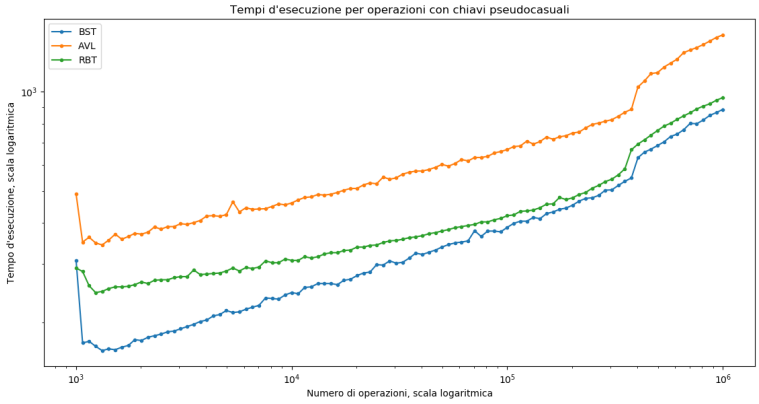
4 Analisi dei tempi di esecuzione

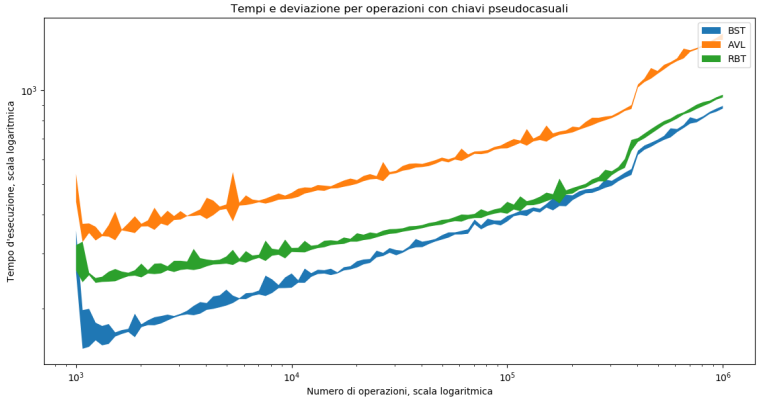
4.1 Caso Random





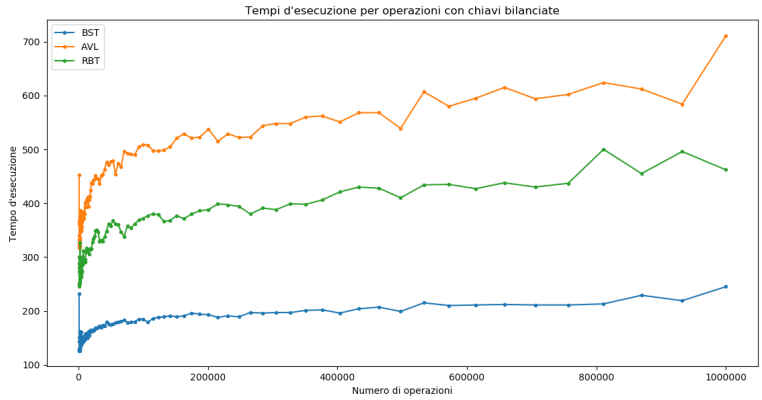
[1] In realtà con un errore ϵ_{max}=0.01 il numero di ripetizioni k eseguite è esattamente 20

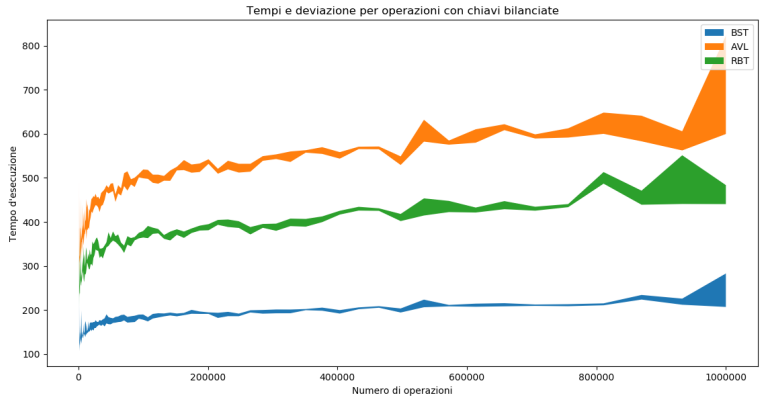


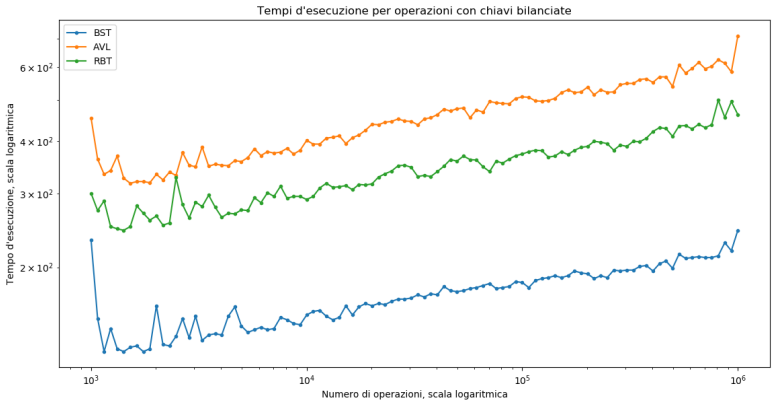


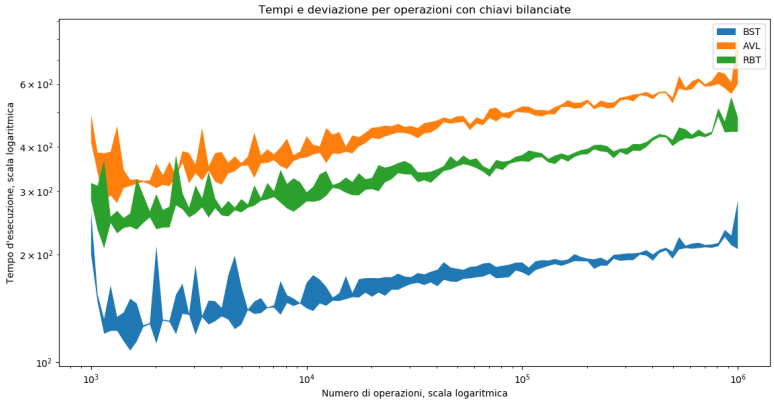
Nel caso di chiavi estratte in ordine pseudocasuale, le performance di tutti e tre gli alberi assumono ordine logaritmico. Questo comportamento è assicurato nel caso dei *AVL* e *RBT* dalle loro euristiche di bilanciamento che pongono il costo delle operazioni sulla struttura dati a $\Theta(\log n)$ a prescindere dall'ordine di inserimento delle chiavi. Al contrario, l'andamento logaritmico dei *BST* non è garantito da alcun accorgimento specifico della struttura dati, bensì dal fatto che estraendo pseudocasualmente un numero intero è equiprobabile che esso sia maggiore o minore rispetto a ciascun nodo che compone l'albero in quell'istante. Di conseguenza, il *BST* risulta *probabilmente* bilanciato e l'assenza di ulteriori procedure per la ridistribuzione dei nodi all'interno delle operazioni di inserimento determina migliori performance rispetto agli altri due tipi di alberi. Viceversa, gli accorgimenti per mantenere l'altezza logaritmica negli alberi *AVL* richiedono un numero elevato di operazioni, determinando il peggioramento nei tempi di esecuzione evidente nei grafici.

4.2 Caso notevole: Balanced





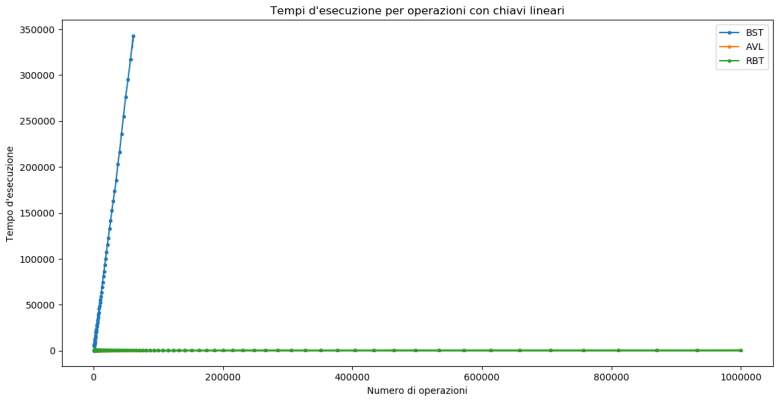


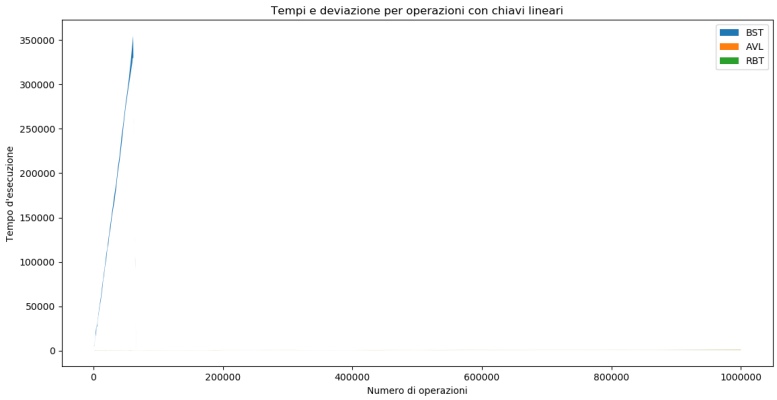


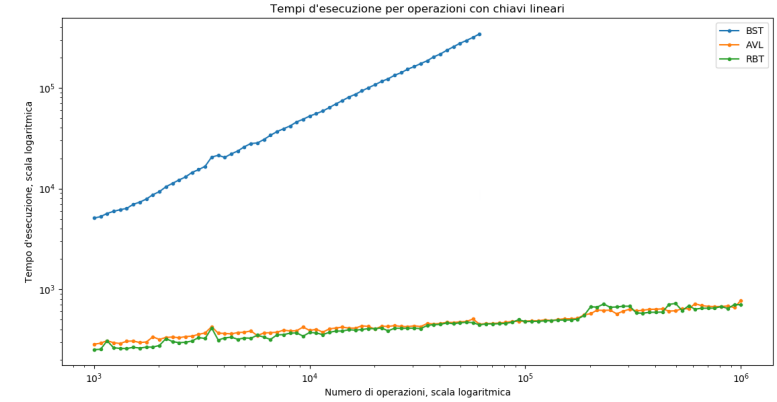
Utilizzando l'algoritmo *Balanced_Initialise()* per la generazione delle chiavi, i tempi registrati mostrano che il costo di un inserimento all'interno dei *BST* è logaritmico. Ciò è dovuto al fatto che le chiavi così estratte e inserite creano alberi necessariamente bilanciati: di conseguenza le performance dei *Binary Search Tree* rimangono nel caso migliore $\Theta(\log n)$. Si può notare quindi che le prestazioni migliori anche in questo caso si raggiungono con i *BST* e questo è dovuto al fatto che, a differenza degli *AVL* e *RBT*, questa struttura dati non adotta delle operazioni di bilanciamento, ottenendo un minor spreco di tempo durante le operazioni di inserimento.

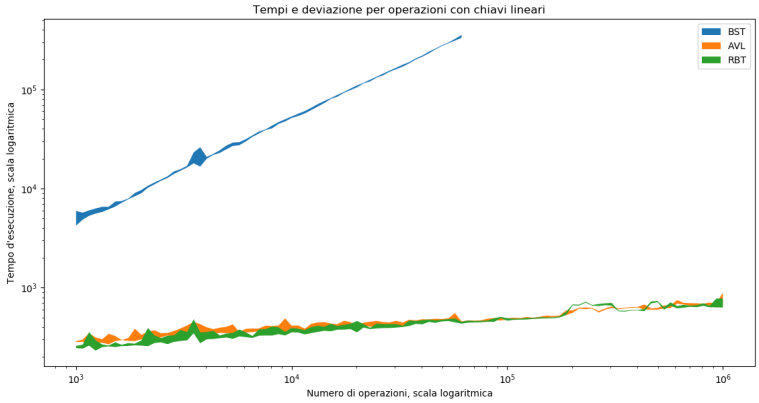
È interessante inoltre notare come questi grafici assomigliano molto a quelli del caso *Random_Initialise()* in quanto, come precedentemente affermato, l'estrazione pseudocasuale di chiavi porti a sequenze statisticamente *probabilmente* bilanciate.

4.3 Caso notevole: Linear









Generando le chiavi da inserire in modo monotono crescente si ricade nel caso peggiore per i *BST*, che degenerano in una lista concatenata; di conseguenza, la complessità per la ricerca e l'inserimento diventa lineare rispetto al numero di nodi che compongono l'albero nel determinato istante. Proprio per questo motivo, nel grafico non sono riportati i dati per esecuzioni oltre $n \approx 45000$, in quanto i tempi sarebbero diventati troppo lunghi da misurare. Questo problema non si presenta nel caso degli alberi *AVL* e *RBT*, perché mantengono il bilanciamento grazie alle loro particolari proprietà: a prescindere dall'ordine di inserimento, tali strutture dati hanno sempre altezza logaritmica.