Informatica - Area scientifica
Dipartimento di Scienze matematiche, informatiche e fisiche
Università di Udine

Progetto di Algoritmi (Seconda parte)

Agnoletti Pierre (150426) agnoletti.pierre@spes.uniud.it
Da Re Davide (141976) dare.davide@spes.uniud.it
Gazzola Elia (147575) gazzola.elia@spes.uniud.it
Moro Martina (147592) moro.martina001@spes.uniud.it
Pasin Angelica (149479) pasin.angelica@spes.uniud.it

Anno accademico 2020/2021

Indice

Introduzione

La seconda parte del progetto richiede l'implementazione e l'analisi dei tempi di esecuzione di operazioni di ricerca e inserimento in alberi binari di ricerca. Nello specifico viene richiesto di implementare le operazioni di ricerca e inserimento per tre tipi diversi di alberi binari di ricerca: alberi binari di ricerca semplici, di tipo AVL e di tipo Red-Black. Ogni nodo di un albero binario di ricerca contiene una chiave numerica (di tipo intero) e un valore alfanumerico (di tipo stringa).

La classe "Node" contiene tutte le proprietà di base comuni ad ogni tipologia di nodo, quali la chiave, il contenuto, i nodi adiacenti (parent, left e right) ed un booleano posto a true se il nodo in questione è NIL. Ad ognuna di queste proprietà sono associati due metodi, relativamente un getter e un setter.
Le classi rappresentanti le varie tipologie di nodi estendono appunto la classe Node, aggiungendo ad essa le caratteristiche tipiche di ogni tipologia di nodo. Fa eccezione il nodo dell'albero BST che non necessita di nessuna caratteristica aggiuntiva.

1 Algoritmi implementati

1.1 Alberi binari di ricerca semplici

Binary search tree (BST) Sono alberi binari in cui ogni nodo contiene una chiave intera tale che per ogni nodo x appartenente all'albero, tutte le chiavi che si trovano nel sottoalbero radicato a sinistra di x sono minori della chiave di x e tutte le chiavi che si trovano nel sottoalbero a destra di x sono maggiori della chiave di x.
Vengono implementate le seguenti funzioni:

- **BST():** Costruttore della classe, crea albero vuoto con root=NULL
- **insert(int key, String value):** Inserisce un nodo nella posizione corretta del BST
- **search(int key):** Cerca il nodo con chiave key all'interno del BST
- **preOrderVisit():** Restituisce una visita preOrder dell'albero
- **reset():** Resetta il BST ponendo a NULL il valore della radice dell'albero

1.2 Alberi binari di ricerca di tipo AVL

Alberi di tipo AVL Sono alberi binari che, oltre a soddisfare la proprietà di un albero di ricerca semplice, devono soddisfare anche la seguente proprietà: per ogni nodo x, le altezze dei sotto-alberi di sinistra e di destra nel nodo x differiscono al più di 1. Vengono implementate le seguenti funzioni:

- **AVL():** Costruttore della classe, crea albero vuoto con root=NULL (di tipo AVL)
- **unBalancedNodes(AvlNode avl):** Ritorna la differenza di altezza tra il ramo sinistro ed il ramo destro di un nodo (numero di nodi sbilanciati)
- **rotateRight(AvlNode avlR):** Esegue una rotazione a destra rispetto al nodo passato come parametro
- **rotateLeft(AvlNode avlL):** Esegue una rotazione a sinstra rispetto al nodo passato come parametro
- **getHeight(AvlNode avl):** Ritorna l'altezza di un nodo passato come parametro
- **insert(int key, String value):** Inserisce il nodo di chiave key con contenuto value nella posizione corretta

- **search(int key):** Cerca il nodo con chiave key all'interno dell'albero
- **preOrderVisit():** Restituisce una visita preOrder dell'albero
- **reset():** Resetta l'albero AVL ponendo a NULL il valore della radice dell'albero

1.3 Alberi binari di ricerca di tipo Red-Black

Red black tree (RBT) Sono dei BST in cui ogni nodo ha un campo "color" che può essere red o black in modo che:

1. Le foglie sono NIL e black
2. Ogni nodo Red ha due figli black
3. Per ogni nodo x lungo ogni cammino x-foglia si trova sempre lo stesso numero di nodi black
4. La radice è black

Vengono implementate le seguenti funzioni:

- **RBT():** Costruttore della classe, crea albero vuoto con root=NULL (di tipo RBT)
- **rotateRight(RBTNode red):** Esegue una rotazione a destra sul nodo passato come parametro
- **rotateLeft(RBTNode red):** Esegue una rotazione a sinistra sul nodo passato come parametro
- **insert(int key, String value):** Inserisce il nodo di chiave key con contenuto value nella posizione corretta
- **FixInsert(RBTNode z):** Tramite rotazioni e ricolorazioni risolve eventuali problemi sul nodo z
- **search(int key):** Cerca il nodo con chiave key all'interno dell'albero
- **preOrderVisit() :** Restituisce una visita preOrder dell'albero
- **reset() :** Resetta l'albero RBT ponendo a NULL il valore della radice dell'albero

2 Classi utili

2.1 Misurazione dei tempi

La classe TimesRecording, visibile nel modulo VPL dedicato, è composta da quattro metodi:

1. **timesOnFile():**
   La procedura "timesOnFile" ha lo scopo di iterare per 100 volte le misurazioni dei tempi per ogni tipologia di albero. Inizialmente vengono istanziati tre oggetti rappresentanti i tre tipi differenti di alberi, che verranno poi passati uno per volta a tre chiamate diverse alla funzione privata "timesRecorder" che si occuperà poi di eseguire le effettive misurazioni.
   Ad ogni iterazione del ciclo for viene inoltre calcolato il numero di interi che verranno poi cercati/inseriti nei vari alberi, tale valore viene generato tramite una funzione esponenziale in i, del tipo $A * (B^i)$.
   Il numero di operazioni viene scritto in una stringa denominata "textToFile" assieme al tempi ammortizzati e alle deviazioni standard relative ai tre alberi. Alla fine di ogni iterazione del ciclo for tale stringa viene scritta in un file di testo e viene poi pulita, per fare spazio ai valori successivi della prossima iterazione.

2. **timesRecorder(Trees tree, int op):**
   La procedura timesRecorder calcola il tempo di esecuzione ammortizzato e la deviazione standard relativi ad un albero in particolare, passato come parametro alla procedura.
   Viene generato un vettore di interi randomici, tali interi saranno poi oggetto delle operazioni di ricerca e inserimento che andremo ad effettuare nell'albero; la dimensione di tale vettore viene passata come parametro alla procedura.
   I tempi di esecuzione complessivi per le operazioni contenute nel vettore vengono rilevati per 50 volte, in modo da ottenere delle misurazioni più precise.
   Per ogni misurazione, le operazioni vengono ripetute finché il tempo rilevato non supera il tempo minimo misurabile dalla macchina, tenendo conto anche dell'errore massimo consentito.
   Per concludere vengono calcolati i valori del tempo ammortizzato e della deviazione standard, tali valori vengono poi inseriti nella stringa "textToFile" che verrà infine memorizzata in un file di testo tramite la funzione "writeFile".

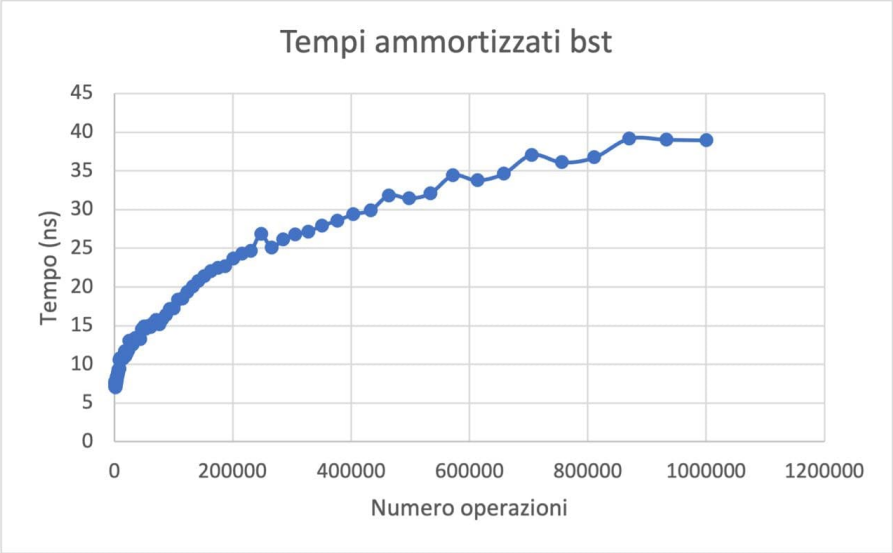
3. **writeFile(String path, String text, boolean create):**
   Scrive su file una stringa passata come parametro ed è in grado di creare il file se esplicitamente richiesto. Se il file è gia esistente, modifica il nome del file finchè non ne trova uno inutilizzato.

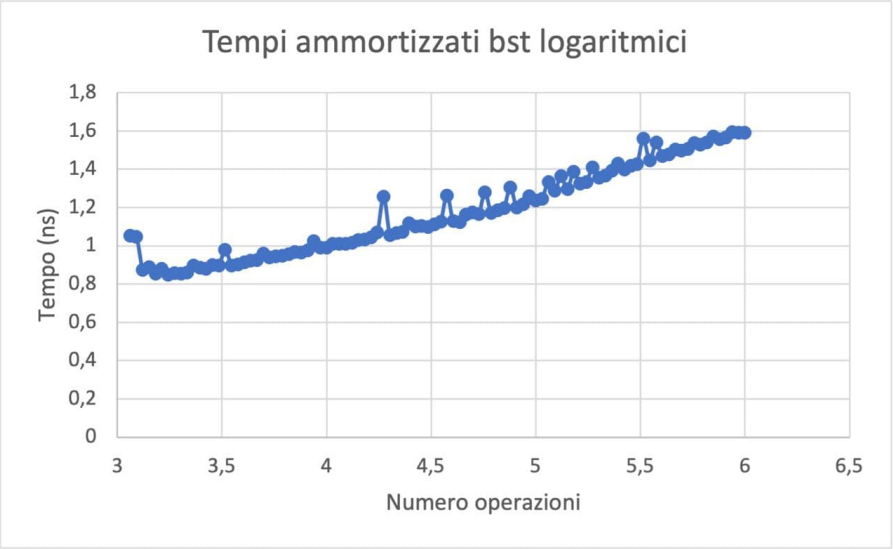
4. **getResolution():**
   Stima la risoluzione del clock di sistema utilizzando un ciclo while per calcolare l'intervallo minimo di tempo misurabile.

3 Analisi dei tempi

3.1 Analisi BST

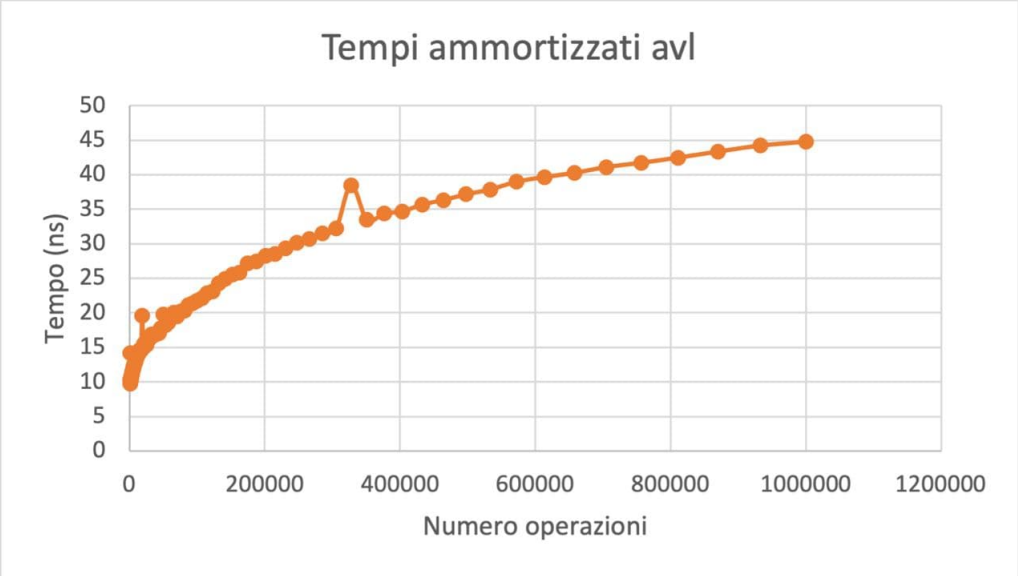
Di seguito vengono riportati i grafici dei BST sia in scala lineare che in scala doppiamente logaritmica.
Dai grafici si può notare l'andamento logaritmico dei tempi ammortizzati rispetto al numero di operazioni eseguite.

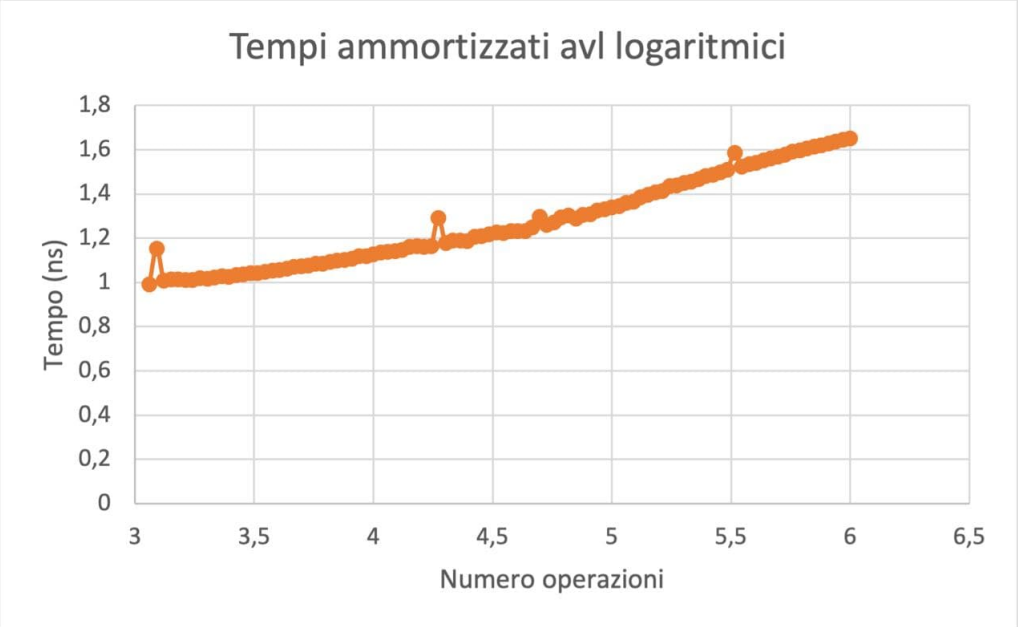




3.2 Analisi AVL

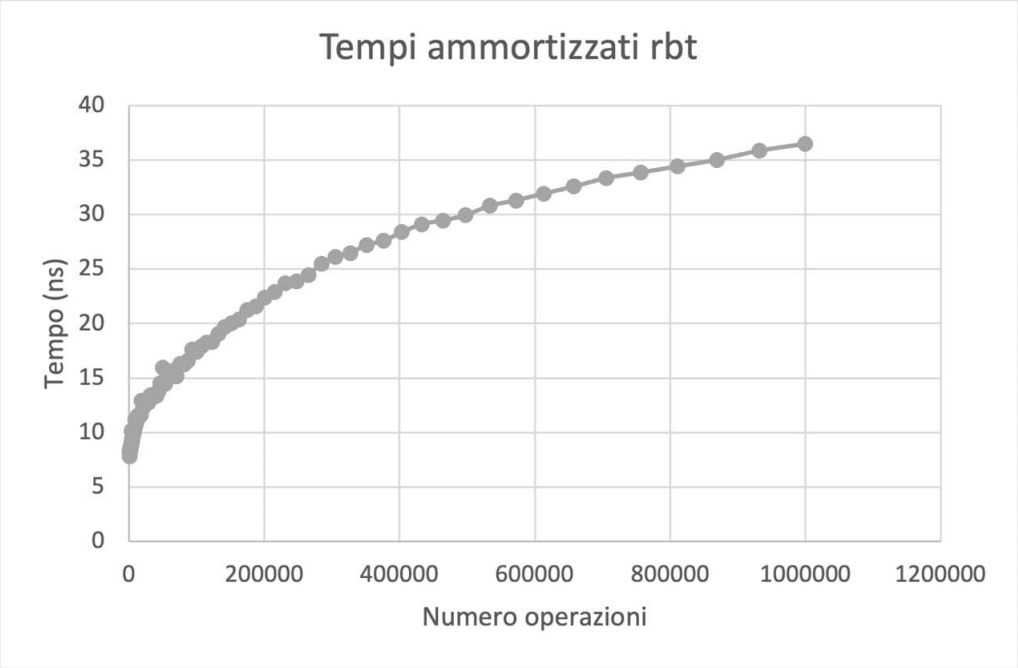
Di seguito vengono riportati i grafici dei AVL sia in scala lineare che in scala doppiamente logaritmica.
Dai grafici si può notare l'andamento logaritmico dei tempi ammortizzati rispetto al numero di operazioni eseguite ed il costo delle operazioni, pari a $O(logn)$.

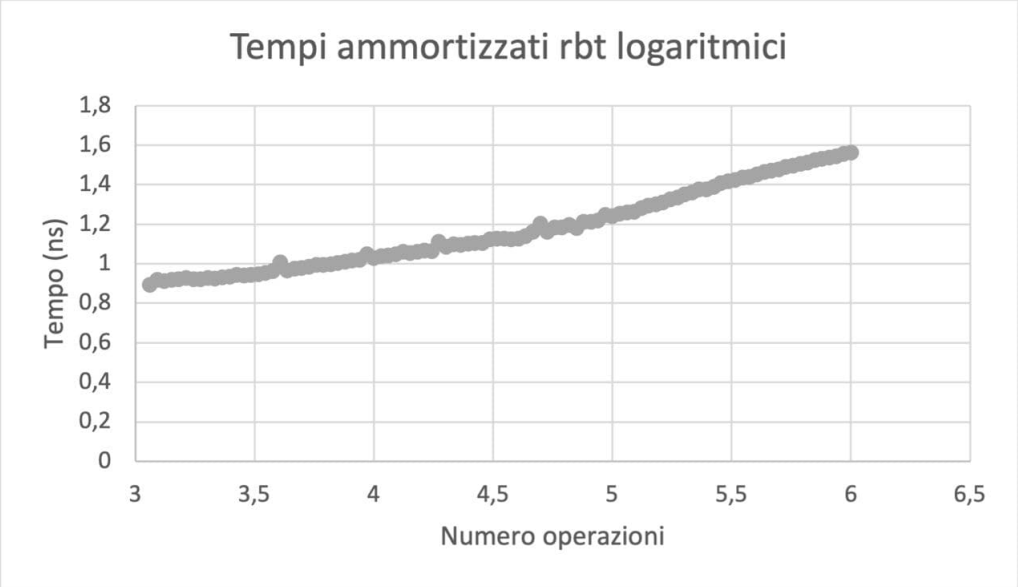




3.3 Analisi RBT

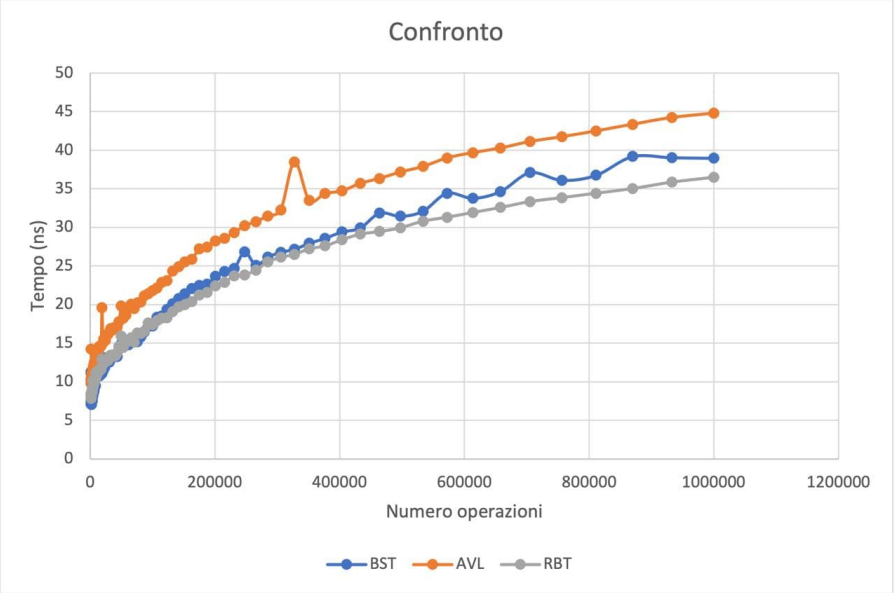
Di seguito vengono riportati i grafici dei RBT sia in scala lineare che in scala doppiamente logaritmica.
Dai grafici si può notare l'andamento logaritmico dei tempi ammortizzati rispetto al numero di operazioni eseguite ed il costo delle operazioni, pari a $O(logn)$.

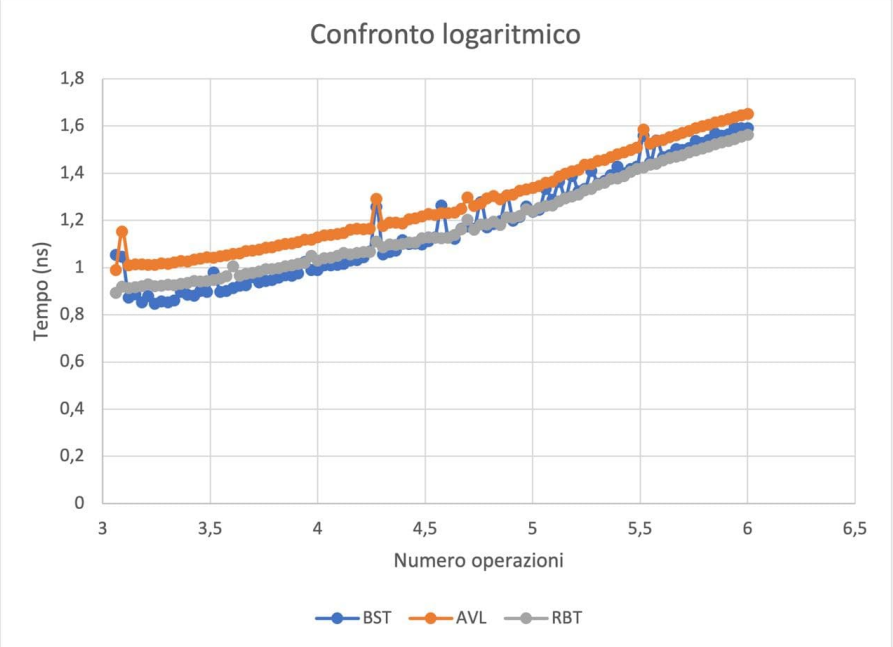


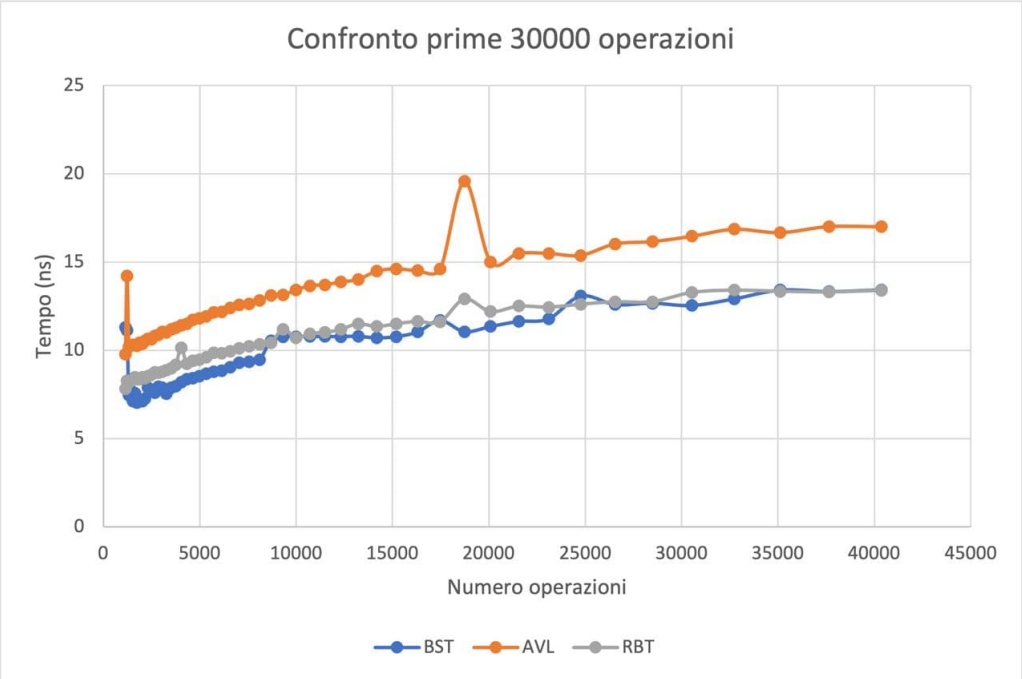


4 Conclusioni

Dai grafici comparativi si può notare come la miglior struttura dati sia l'albero RBT. Tuttavia, come si può vedere dal grafico di confronto delle prime 30000 operazioni, i BST sono più rapidi nello svolgere le n operazioni di ricerca e inserimento con $n < 25000$ circa.






Confronto prime 30000 operazioni
Tempo (ns)
25
20
15
10
5
0
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
Numero operazioni
BST
AVL
RBT