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

Progetto di Algoritmi e Strutture Dati e Laboratorio

Seconda parte

Biasi Yuri (142121)
Galvan Matteo (142985)
Mocchi Alberto (144552)
Vendrame Gabriel (128311)

Anno accademico 2019/2020

Indice

Capitolo 1

Introduzione

La seconda parte del progetto richiede l'implementazione e l'analisi dei tempi di l'esecuzione di operazioni di inserimento e di ricerca in alberi binari di ricerca. Nello specifico, si richiede di implementare le operazioni di ricerca e inserimento per tre tipi diversi di alberi binari di ricerca: alberi binari di ricerca semplici (BST), di tipo AVL e di tipo Red-Black.
Per ogni tipologia di algoritmo sono state esplicitate le funzioni con i rispettivi input, output e con la loro implementazione.
Sono presenti le implementazioni dei programmi per il calcolo dei tempi medi, oltre ai grafici di ogni algoritmo di ordinamento al variare della dimensione del vettore e del tempo per eseguirlo.
È presente infine un grafico riassuntivo contenente la comparazione dei tempi tra i vari algoritmi all'aumentare del numero di valori all'interno del vettore.

Capitolo 2

Binary Search Tree (BST)

Abbiamo considerato gli alberi binari di ricerca in cui ogni nodo di un albero binario di ricerca contiene una chiave numerica (di tipo intero) e un valore alfanumerico (di tipo stringa). Un albero binario di ricerca semplice deve soddisfare la seguente proprietà: per ogni nodo x che non sia una foglia e per ogni nodo y nel sotto-albero sinistro (rispettivamente, destro) di x, la chiave associata a x è strettamente maggiore (rispettivamente, minore) della chiave associata a y.

L'algoritmo implementa le seguenti funzioni:

- BST: costruttore che inizializza la radice.
- insert: dato un nodo, il valore (la chiave del nodo) e la stringa (valore del nodo), lo inserisce nell'albero.
- search: dato un valore che è la chiave del nodo esegue la ricerca.
- reset: funzione che crea una nuova radice, cancellando l'albero preesistente.

Capitolo 3

Adelson-Velsky and Landis Tree (AVL)

Abbiamo considerato gli alberi binari di ricerca in cui ogni nodo di un albero binario di ricerca contiene una chiave numerica (di tipo intero) e un valore alfanumerico (di tipo stringa). Si ricorda che un albero binario di ricerca di tipo AVL, oltre a soddisfare la proprietà di un albero di ricerca semplice, deve 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. Info in più

L'algoritmo implementa le seguenti funzioni:

- AVL: costruttore che inizializza la radice.
- unBalancedNodes: dato in input un nodo dell'albero ritorna il numero di nodi sbilanciati del sottoalbero, quindi di quanto il sottoalbero sinistro differisce da quello destro.
- getHeight: dato un nodo in input ritorna l'altezza del nodo.
- rotateRight: dato un nodo in input effettua la rotazione destra su quel nodo.
- rotateLeft: dato un nodo in input effettua la rotazione sinistra su quel nodo.
- insert: dato il valore (la chiave del nodo) e la stringa (valore del nodo), lo inserisce nell'albero e ritorna true.
- search: dato un valore che è la chiave del nodo esegue la ricerca.
- reset: funzione che crea una nuova radice, cancellando l'albero preesistente.

Capitolo 4

Red Black Tree (RBT)

Negli alberi binari di ricerca di tipo Red-Black ogni nodo ha associato un colore, che può essere rosso o nero. L'altezza *nera* del sottoalbero radicato in un nodo x è definita come il massimo numero di nodi neri lungo un possibile cammino da x a una foglia. Un albero binario di ricerca Rosso-Nero deve soddisfare anche la seguente proprietà: per ogni nodo x, le altezze nere dei sotto-alberi di sinistra e di destra nel nodo x coincidono.

L'algoritmo implementa le seguenti funzioni:

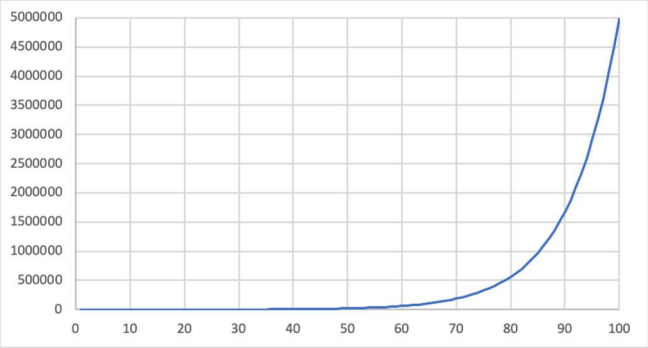
- RBT: costruttore che inizializza la radice.
- rotateRight: dato un nodo in input effettua la rotazione destra su quel nodo.
- rotateLeft: dato un nodo in input effettua la rotazione sinistra su quel nodo.
- insert: dato il valore (la chiave del nodo) e la stringa (valore del nodo), lo inserisce nell'albero.
- FixInsert: dato in input un nodo, corregge gli errori di riposizionamento e di colorazione, modificando il colore della radice ed effettuando eventuali ricolorazioni.
- search: dato un valore che è la chiave del nodo esegue la ricerca.
- reset: funzione che crea una nuova radice, cancellando l'albero preesistente.

Capitolo 5

Analisi dei tempi

La presa dei tempi è stata raccolta all'interno del main, in modo da condividere la stessa struttura per tutti e tre gli alberi. Nello specifico nella prima parte vengono definite le variabili stabilendo il numero di campioni (100), numero di ripetizioni (1000, 1000000) e le ripetizioni da eseguire per una migliore accuratezza dei tempi (50) con un errore costante massimo di 0.01.

Con la chiamata a **getMachineBasedTimeResolution** viene restituito il valore minimo teorico della macchina che servirà in **getTime** per un calcolo del tempo più veritiero. Vengono quindi nidificati alcuni cicli partendo dai campioni complessivi; successiva la definizione della grandezza del vettore per la generazione randomica seguendo la seguente formula:



$$base = e^{[\ln(maxSize) - \ln(minSize)]/(samples-1)}$$

$$minSize * base^{sample[i]}$$

Poi per ogni albero viene chiamata **insertOperation,** la quale dopo aver generato l'array di numeri casuali data la grandezza ricevuta in input, ritorna il tempo di inserimento negli alberi (se non presente) di ogni k del vettore casuale.

Con i dati a disposizione per ogni ripetizione di inserimento, viene inoltre calcolata la deviazione standard. Viene in questo caso calcolato anche il tempo ammortizzato dato dal tempo totale diviso il numero di operazioni di ricerca ed eventuale inserimento.

- **getMachineBasedTimeResolution**: funzione che restituisce il tempo di risoluzione in nano secondi di un'operazione predefinita, basandosi sul numero di clock della macchina.
- **insertOperation :** restituisce i tempi di inserimento di ogni operazione.

L'output generato è stato salvato direttamente in un file csv seguendo il seguente formato:

N T1 D1 T2 D2 T3 D3 , dove N è il numero di numeri da inserire. Ti (i=1,2,3) è una stima del tempo ammortizzato medio di inserimento nell'i-esimo albero e Di (i=1,2,3) è la relativa deviazione standard.

Grafico Binary Search Tree

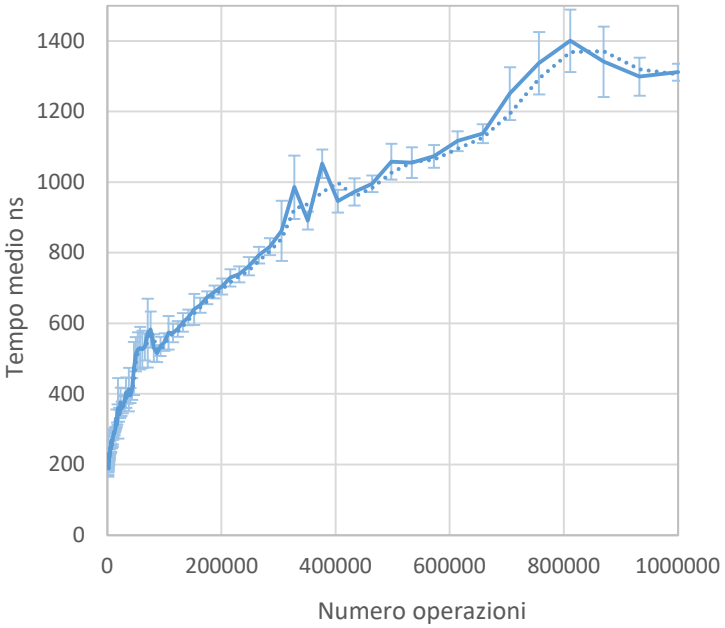
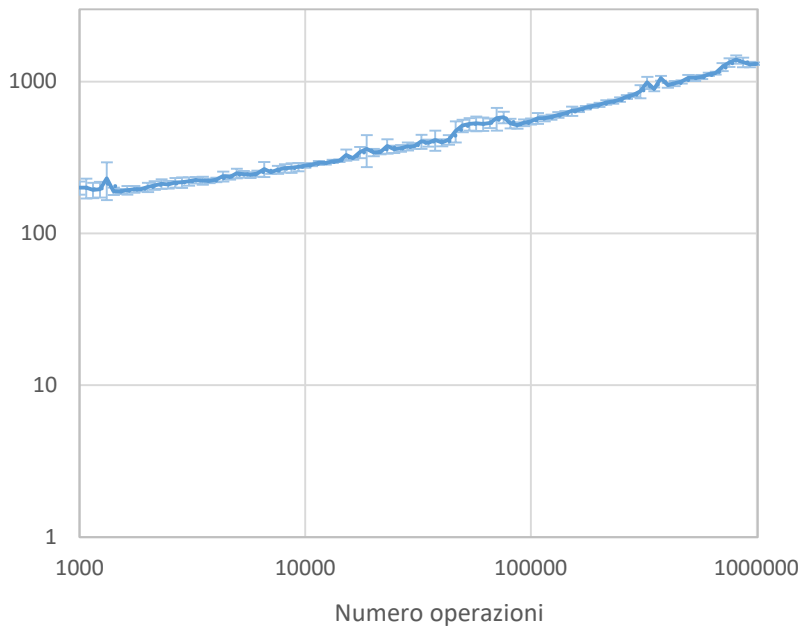


Grafico Binary Search Tree logaritmico



Possiamo notare dal grafico che le operazioni di ricerca e inserimento costano al più O(n).

Grafico AVL

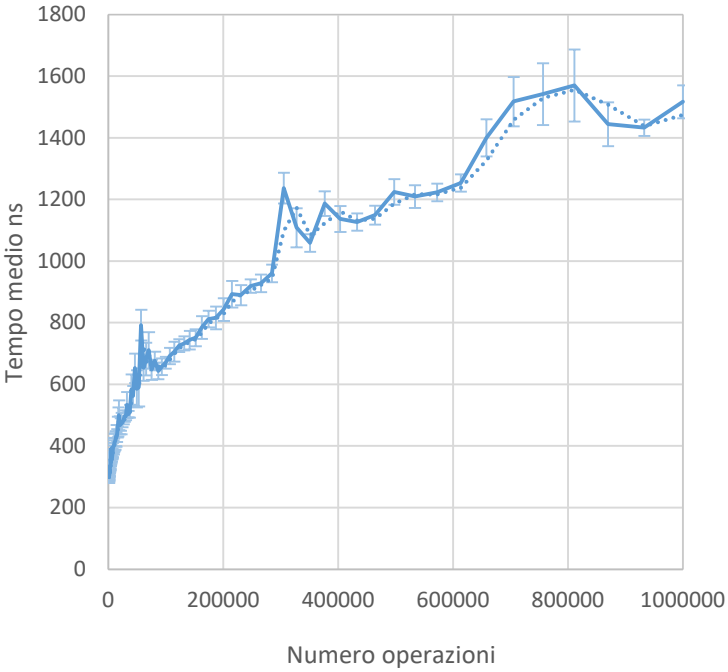
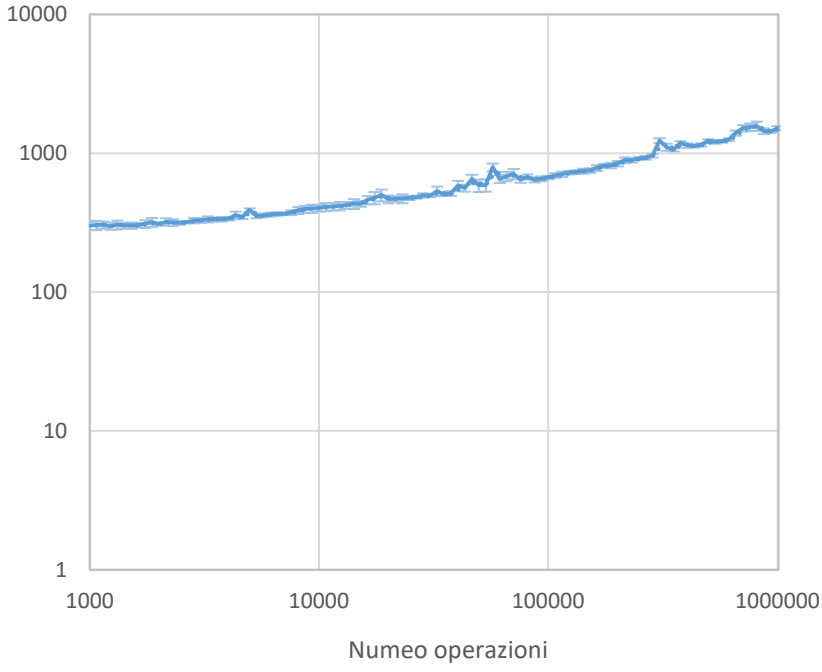


Grafico AVL logaritmico



Da questo grafico possiamo notare che le operazioni di ricerca e inserimento costano O(logn).

Grafico Red Black Tree

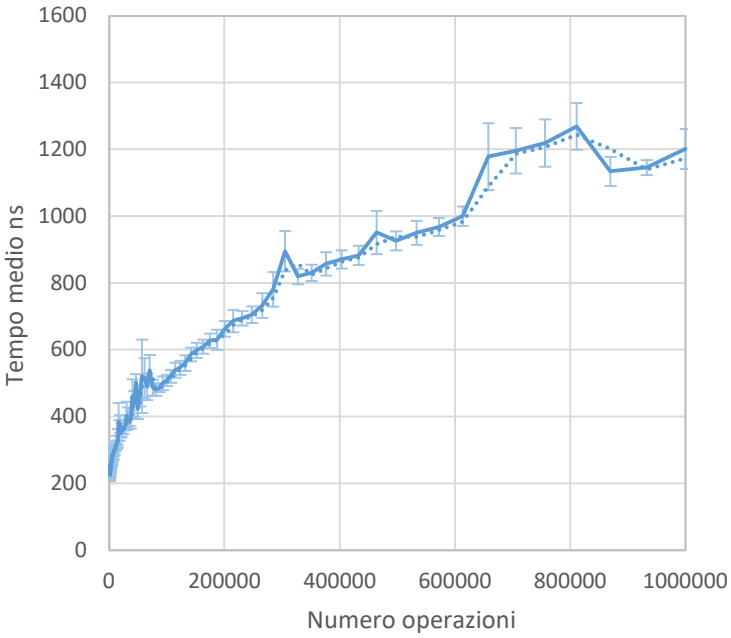
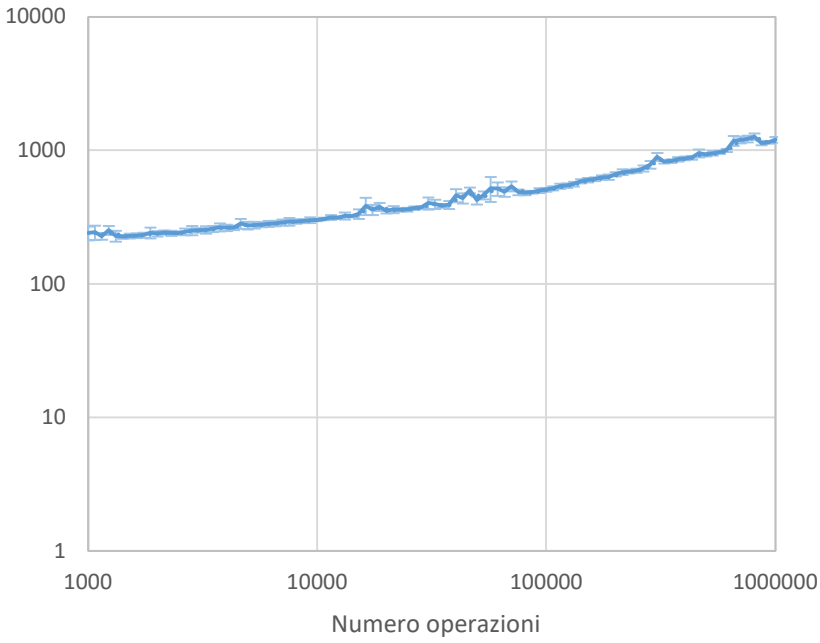


Grafico Red Black Tree logaritmico



Da questo grafico possiamo notare che le operazioni di ricerca e inserimento costano O(logn)

Capitolo 6

Conclusioni

Grafico riassuntivo

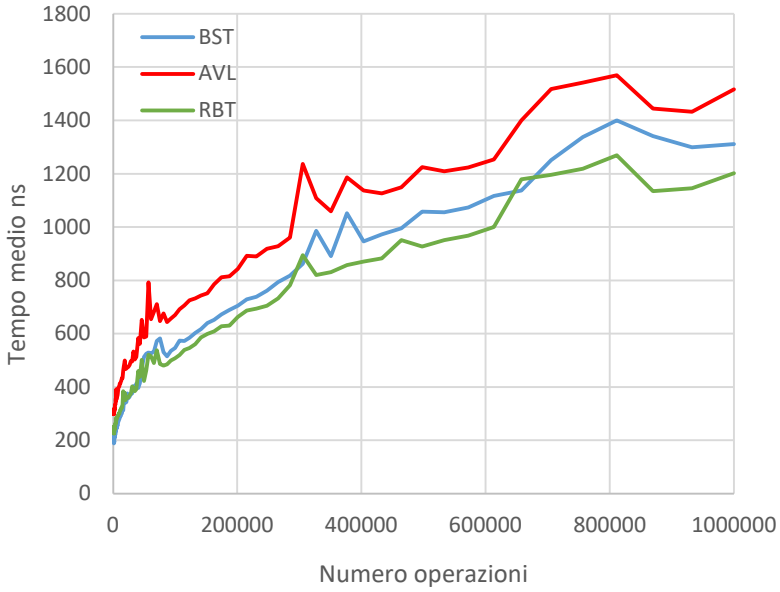
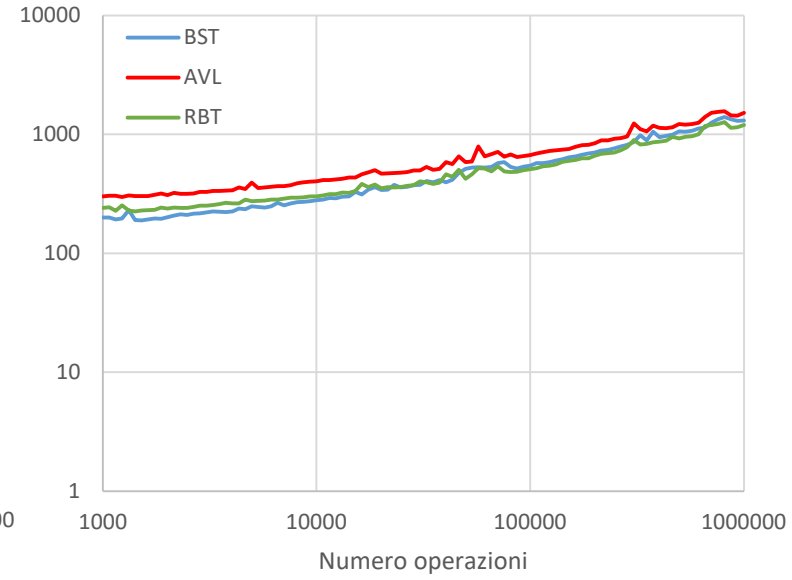


Grafico riassuntivo logaritmico



Dettaglio

Dal grafico comparativo si può notare che fino a 20.000 operazioni la struttura dati migliore è il Binary Search Tree (BST) seguito da RBT e da AVL. Da 50.000 operazioni la migliore struttura dati è il Red Black Tree (RBT).
Come si può notare dal grafico tutte e tre le strutture dati hanno un andamento logaritmico.